One VM to Rule Them All

Thomas Würthinger* Christian Wimmer* Andreas Wöß† Lukas Stadler†
Gilles Duboscq† Christian Humer† Gregor Richards§ Doug Simon* Mario Wolczko*

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria §S³ Lab, Purdue University
{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
{woess, stadler, duboscq, christian.humer}@ssw.jku.at gr@purdue.edu

Abstract

Building high-performance virtual machines is a complex and expensive undertaking; many popular languages still have low-performance implementations. We describe a new approach to virtual machine (VM) construction that amortizes much of the effort in initial construction by allowing new languages to be implemented with modest additional effort. The approach relies on abstract syntax tree (AST) interpretation where a node can rewrite itself to a more specialized or more general node, together with an optimizing compiler that exploits the structure of the interpreter. The compiler uses speculative assumptions and deoptimization in order to produce efficient machine code. Our initial experience suggests that high performance is attainable while preserving a modular and layered architecture, and that new high-performance language implementations can be obtained by writing little more than a stylized interpreter.

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

Keywords Java; JavaScript; dynamic languages; virtual machine; language implementation; optimization

1. Introduction

High-performance VM implementations for object-oriented languages have existed for about twenty years, as pioneered by the Self VM [25], and have been in widespread use for fifteen years. Examples are high-performance Java VMs such as Oracle's Java HotSpot VM [28] and IBM's Java VM [30], as well as Microsoft's Common Language Runtime (the VM of the .NET framework [42]).

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2472-4/13/10/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509578.2509581

These implementations can be characterized in the following way:

- Their performance on typical applications is within a small multiple (1-3x) of the best statically compiled code for most equivalent programs written in an unsafe language such as C.
- They are usually written in an unsafe, systems programming language (C or C++).
- Their implementation is highly complex.
- They implement a single language, or provide a bytecode interface that advantages a narrow set of languages to the detriment of other languages.

In contrast, there are numerous languages that are popular, have been around for about 20 years, and yet still have no implementations that even approach this level of performance. Examples are PHP, Python, Ruby, R, Perl, MATLAB, Smalltalk, and APL. The *computer language benchmarks game* [20] provides some insights into the performance of various languages.

JavaScript is in an intermediate state. Performance was lackluster until 2008, but significant effort has since been invested in several competing implementations to make performance respectable. We believe that the recent history of JavaScript explains why the other languages have lesser performance: until industrial-scale investment becomes available, the complexity of a traditional high-performance VM is too high for small-scale efforts to build and maintain a high-performance implementation for multiple hardware platforms and operating systems.

We present a new approach and architecture, which enables implementing a wide range of languages within a common framework, reusing many components (especially the optimizing compiler). This brings excellent performance for all languages using our framework. Our approach relies on a framework that allows nodes to rewrite[1] themselves during interpretation, a speculatively optimizing compiler that can

[1] We use "node rewriting" in a sense that is distinct from rewriting in the context of the lambda calculus, formal semantics, or term rewriting systems.

exploit the structure of interpreters written using this framework, and deoptimization to revert back to interpreted execution on speculation failures.

In this paper, we describe the overall approach and the structure of our implementation, with particular attention to the interaction between the interpreter, the compiler, and deoptimization. Additionally, we describe the variety of language implementations we are undertaking, with specific reference to their unique attributes and the challenges posed. Our focus is on dynamically typed, imperative programming languages such as JavaScript, Ruby, and Python; as well as languages for technical computing such as R and J. Section 6 presents details on the languages.

This paper presents a forward-looking viewpoint on work in progress. We have a working prototype implementation of the interpreter framework and the compilation infrastructure. A detailed description of the node rewriting appears elsewhere [72]; a brief summary of that paper appears at the beginning of Section 3. The AST and machine code of the example presented in the subsequent sections are produced by our actual system. We are therefore convinced that the approach can be successful. However, more implementation work is necessary to get larger industry-standard benchmarks running for multiple languages.

Our language implementation framework, called *Truffle*, as well as our compilation infrastructure, called *Graal*, are available as open source in an OpenJDK project [45]. We encourage academic and open source community contributions, especially in the area of new innovative language features and language implementations. Language implementations that are started now, using our framework, will automatically profit from our compilation infrastructure when it is finished, so we envision multiple language implementations being developed by third parties simultaneously to our efforts. In summary, this paper contributes the following:

- We present a new VM architecture and our implementation, which can be used to construct high-performance implementations of a wide range of languages at modest incremental cost.
- We show how the combination of node rewriting during interpretation, optimizing compilation, and deoptimization delivers high performance from an interpreter without requiring a language-specific compiler.
- We show how features of a variety of programming languages map on our framework.
- We describe how our system supports alternative deployment strategies; these enhance developer productivity and separate language-specific from language-independent components.

2. System Structure

High-performance VM implementations are still mostly monolithic pieces of software developed in C or C++. VMs offer many benefits for applications running on top of them, but they mostly do not utilize these benefits for themselves. In contrast, we want the VMs for many different *guest language*s written in a managed *host language*. Only the guest-language specific part is implemented anew by the language developer. A core of reusable host services are provided by the framework, such as dynamic compilation, automatic memory management, threads, synchronization primitives, and a well-defined memory model. Figure 1 summarizes our system structure.

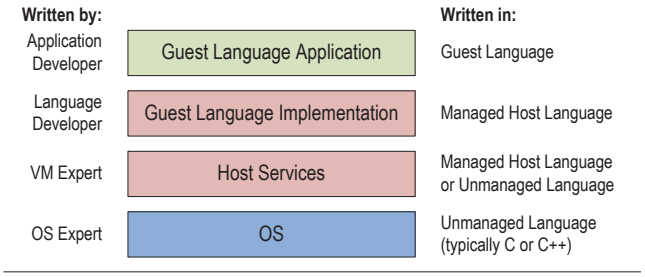

Figure 1. System structure of a guest language implementation utilizing host services to build a high-performance VM for the guest language.

The host services can either be written in the managed host language or in an unmanaged language; our approach does not impose restrictions on that. Section 5 presents different approaches for providing the host services.

For the concrete examples in this paper, the guest language is JavaScript and the host language is a subset of Java, i.e., we have a guest JavaScript VM as well as the host services implemented in Java. However, we want to stress that the idea is language independent. Section 6 shows how to map key features of many guest languages to our system. We have designed our system in the expectation that the host language is statically typed. This allows us to express type specializations, i.e., to explicitly express the semantics of a guest language operation on a specific type, as well as to explicitly express type conversions in the (possibly dynamically typed) guest language.

Our layered approach simplifies guest language implementation. Host services factor out common parts found in every high-performance VM, allowing guest language developers to focus on required execution semantics. However, the benefits of the layering must not sacrifice peak performance of the guest language. Dynamic compilation of guest language code to machine code is therefore essential. Figure 2 and 3 illustrate the key steps of our language implementation and optimization strategy:

- The guest language implementation is written as an AST interpreter. We believe that implementing an AST interpreter for a language is a simple and intuitive way of describing the semantics of many languages. For example, the semantics for an addition are well encapsulated and intuitively described in one place: the addition node.

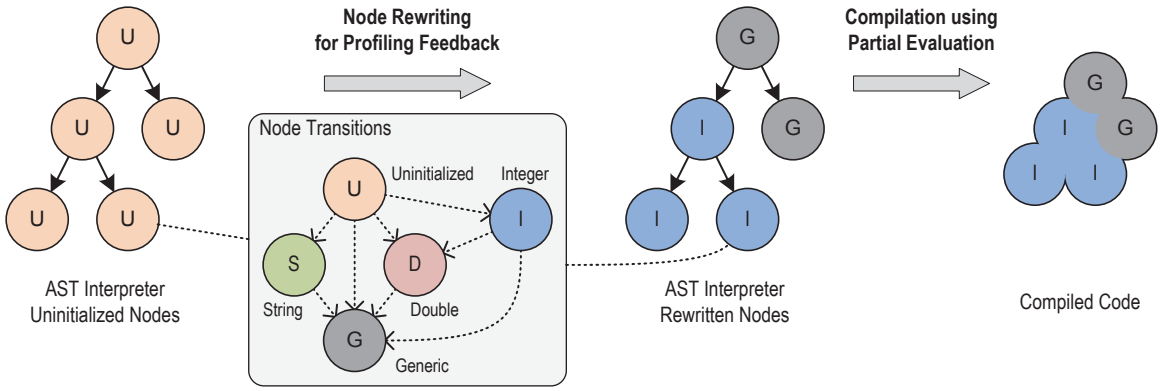

Figure 2. Node rewriting for profiling feedback and partial evaluation lead to aggressively specialized machine code.

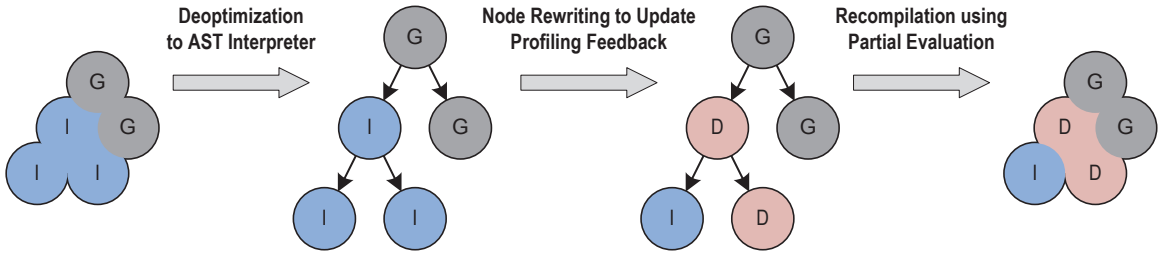

Figure 3. Deoptimization back to the AST interpreter handles speculation failures.

- Node rewriting in the interpreter captures profiling information, e.g., dynamic type information (see Section 3 for details). During interpretation, a node can replace itself at its parent with a different node. This allows a node to specialize on a subset of the semantics of a particular guest language operation. This rewriting incorporates profiling feedback in the AST. Counters measure execution frequencies and the rate of node rewrites.
- When the compiler is invoked to compile a part of the application, it uses the extant ASTs (with profiling and frequency information) to perform an automatic partial evaluation of the AST interpreter (see Section 4 for details). For this, the AST is assumed to be in a *stable* state where subsequent rewrites are unlikely, although not prohibited. Partial evaluation, i.e., compilation with aggressive method inlining, eliminates the interpreter dispatch code and produces optimized machine code for the guest language.
- The parts of the interpreter code responsible for node rewriting are omitted from compilation. Branches that perform rewriting are not compiled, but instead cause *deoptimization* [27]. This results in machine code that is aggressively specialized for the types and values encountered during interpretation.
- In case that a specialization subsequently fails, *deoptimization* discards the optimized machine code and reverts execution back to the AST interpreter. During interpretation, the nodes then perform the necessary rewrites to incorporate revised type information into the AST, which can be compiled again using partial evaluation.

Note that at no point was the dynamic compiler modified to understand the semantics of the guest language; these exist solely in the high-level code of the interpreter and runtime. A guest language developer who operates within our interpretation framework gets a high-performance language implementation, with no need to understand dynamic compilation. Subsequent sections illustrate what kinds of transformations are required to enable the compiler to generate efficient machine code.

The example AST in Figure 2 consists of 5 nodes. The first snapshot on the left side shows the tree before execution, where no type information is available yet. During execution, the nodes are replaced with typed nodes according to types seen at run time. The type transitions shown in the figure model the guest language (JavaScript) data types using host language (Java) types. JavaScript numbers are represented by the Java type *double*, with the optimization that the Java type *integer* can be used for computations that do not overflow the 32-bit signed integer range. Observing that three nodes are typed to *integer*, compilation by using partial evaluation generates *integer*-specialized code for these nodes. Note that we use type transitions only as an intuitive example here. Node rewriting is used for profiling feedback in general and not restricted to type transitions.

Figure 3 continues the example. Assume that a computation overflows the *integer* range. The code compiled cannot handle this case, so deoptimization reverts execution back to the AST interpreter. During interpretation, the *integer* nodes rewrite themselves to *double* nodes. After this change, the AST is compiled again, this time producing *double*-specialized code.

3. Self-optimization

A guest language developer using our system writes a specific kind of AST interpreter in a managed language. A guest language function has a *root node* with an `execute` method returning the result of the function. Every node in the AST has a list of *child nodes*. All nodes except the root node also have a single associated *parent node*. The parent node calls `execute` methods of its child nodes to compute its own result. Non-local control flow (e.g., break, continue, return) in the guest language is modeled using host language exceptions.

3.1 Node Rewriting

During execution, a node can replace itself at its parent with a different node. This allows a node to specialize on a subset of the semantics of a particular guest language operation. If its own handling of the operation cannot succeed for the current operands or system environment, a node replaces itself and lets the new node return the appropriate result.

Node replacement depends on the following conditions. The guest language developer is responsible for fulfilling them, there is no automatic enforcement or verification.

Completeness: Although a node may handle only a subset of the semantics of a guest language operation, it must provide rewrites for *all* cases that it does not handle itself.

Finiteness: After a finite number of node replacements, the operation must end up in a state that handles the full semantics without further rewrites. In other words, there must be a generic implementation of the operation that can handle all possible inputs itself.

Locality: A rewrite may only happen locally for the current node while it is executing. However, as part of this rewrite a node may replace or change the structure of its complete subtree.

When a parent node calls the `execute` method of a child node, the parent node may provide information about the expected constraints on the return value. The child node may either fulfill the expected constraints or provide the result value via an exception. After catching such an exception, a parent node must replace itself with a new node that no longer expects the child node to comply with these constraints. This allows a parent node to communicate to its child node its preferred kind of returned values (see Section 3.4 for an example).

Node replacement allows the AST interpreter to automatically incorporate profiling feedback while executing. This profiling feedback is expressed as the current state of the AST. Concrete possibilities include but are not limited to:

Type Specialization: Operators in dynamic languages can often be applied to a wide variety of operand types. A full implementation of such an operator must therefore apply several dynamic type checks to the operands before choosing the appropriate version of the operator to execute. However, for each particular operator instance in a guest language program, it is likely that the operands always have the same types. The AST interpreter can therefore speculate on the types of the operands and only include the code for one case. If the speculation fails, the operator node replaces itself with a more general version.

Polymorphic Inline Caches: Our system supports polymorphic inline caches [26] by chaining nodes representing entries in the cache. For every new entry in the cache, a new node is added to the tree. The node checks whether the entry matches. Then it either proceeds with the operation specialized for this entry or delegates the handling of the operation to the next node in the chain. When the chain reaches a certain predefined length (e.g., the desired cache size), the whole chain replaces itself with one node responsible for handling the fully megamorphic case.

Resolving Operations: Using rewriting, a guest language operation that includes a resolving step upon first execution replaces itself with a resolved version of the operation. This way, it avoids subsequent checks, a technique that is related to bytecode quickening [11].

Rewriting splits the implementation of an operation into multiple nodes. In general, this should be considered if the following two conditions are met: the implementation covers only a subset of the operator's semantics; and that subset is likely sufficient for the application of the operation at a specific guest language program location.

As a consequence of rewriting, the actual host-language code that is executed changes dynamically when the guest-language interpreter executes a guest-language method—usually towards a faster, more specialized version on successive executions. Because of this, our interpreter has good performance compared to other interpreter implementations [72]. However, we cannot compete with the performance of compiled code. The main overhead is due to dynamic dispatch. We address this problem in Section 4

3.2 Boxing Elimination

The standard `execute` method of a node always returns an `Object` value. However, a node may, e.g., offer a specialized `executeInt` method, whose return type is the primitive *integer* type. A parent node can call this method if it wants to receive an *integer* value. The child node can then either

```
function sum(n) {
  var sum = 0;
  for (var i = 1; i < n; i++) {
    sum += i;
  }
  return sum;
}
```

Figure 4. JavaScript code of example method.

return the value as an *integer*, or if this this not possible, box the value in an object and wrap it in an exception. The parent node catches this exception and replaces itself with a node that no longer calls the `executeInt` method.

This technique can be used to avoid any sorts of boxing in case a child node always produces a primitive value. The parent node no longer needs a type check, type cast, or unboxing operation after receiving the value of the child on the normal path of execution. The exceptional behavior is moved to a catch block.

3.3 Tree Cloning

One problem with dynamic runtime feedback is the pollution of the feedback due to different callers using a method in different ways. In our system, every caller can potentially cause a rewrite of AST nodes to a less specialized version. An operation has to handle the union of the semantics required by every caller. This works against our goal of having every node handle only the smallest, required subset of the semantics of a full operation.

We use cloning of the AST to mitigate this effect. Every node has the ability to create a copy of itself. By cloning the AST of a guest language method for a call site, we avoid the problem of profiling feedback pollution for that method. The cloning increases the number of AST nodes in the system, so we have to be selective in choosing the targets for cloning. For this decision, we use heuristics that include the current specialization state of the AST as well as the invocation count of a method.

3.4 Example

In order to demonstrate the abstract ideas in this paper, we use a concrete JavaScript method as a running example throughout the paper. Although the example shows only a simple subset of JavaScript, it suffices to explain many of our concepts. Figure 4 shows the JavaScript source code of our example method. It adds up all non-negative integers below a provided maximum and returns the result.

Figure 5 shows the AST of the method after parsing. The loop and other operations from the source code are clearly visible. Uninitialized operations are shown with no type prefix in the node operation. Only constants are typed at this state, since it is known whether or not they fit into the value range of *integer*. Although the JavaScript language only defines floating point numbers (represented in our example by the prefix D), it is faster and more space efficient to

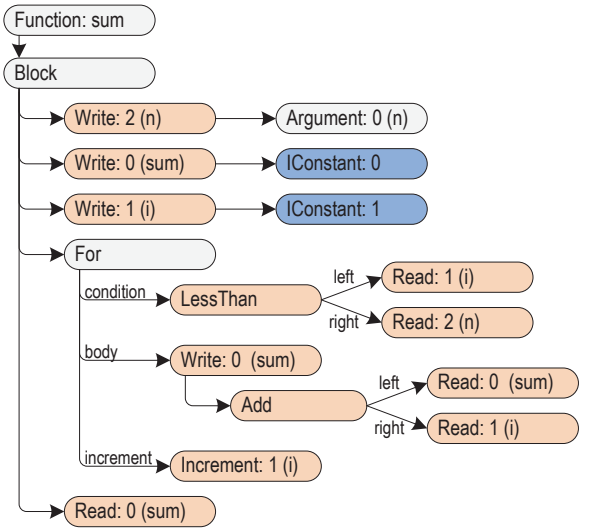

Figure 5. Example AST after parsing.

perform computations that fit into the value range of 32-bit signed integers as *integer* operations (represented in our example by the prefix I).

Assume that the method `sum` is first called with a small value of the parameter `n` so that `sum` fits in the range of *integer*. During the first execution of `sum`, the nodes replace themselves with nodes specialized for the type *integer*. For example, the `Add` node first evaluates its `left` child (the read of the local variable `sum`) and `right` child (the read of the local variable `i`), using the `execute` methods of the children. Both `execute` methods return a boxed `Integer` object. Therefore, the `Add` node replaces itself with an `IAdd` node. This rewriting process involves instantiating a new tree node, i.e., a new object where the `execute` method is implemented differently. The `IAdd` node will be used for subsequent executions. For the current execution, the `Add` node unboxes the `Integer` object to primitive *integer* values, performs the addition, and returns a boxed `Integer` object.

The new `IAdd` node can only add *integer* numbers and triggers another replacement if a child returns a differently typed value. It communicates its expectation to receive an *integer* value to its children by calling a specialized `executeInt` method instead of the generic `execute` method. This method has a primitive *integer* return type, i.e., subsequent executions do not require boxing. If a child cannot comply by returning an *integer* value, it must wrap the return value in an exception and return it on this alternative path. This `UnexpectedResult` exception triggers a replacement of the `IAdd` node.

Figure 6 shows the implementation of the `IAdd` node. Note that the `Math.addExact` method is a plain Java utility method added to the upcoming JDK 8; it throws an `ArithmeticException` instead of returning an overflowed result. The implementation of the `rewrite` method, which

```
class IAddNode extends BinaryNode {
  int executeInt(Frame f) throws UnexpectedResult {
    int a;
    try {
      a = left.executeInt(f);
    } catch (UnexpectedResult ex) {
      throw rewrite(f, ex.result, right.execute(f));
    }

    int b;
    try {
      b = right.executeInt(f);
    } catch (UnexpectedResult ex) {
      throw rewrite(f, a, ex.result);
    }

    try {
      return Math.addExact(a, b);
    } catch (ArithmeticException ex) {
      throw rewrite(f, a, b);
  }
}
```

Figure 6. Implementation of *integer* addition node.

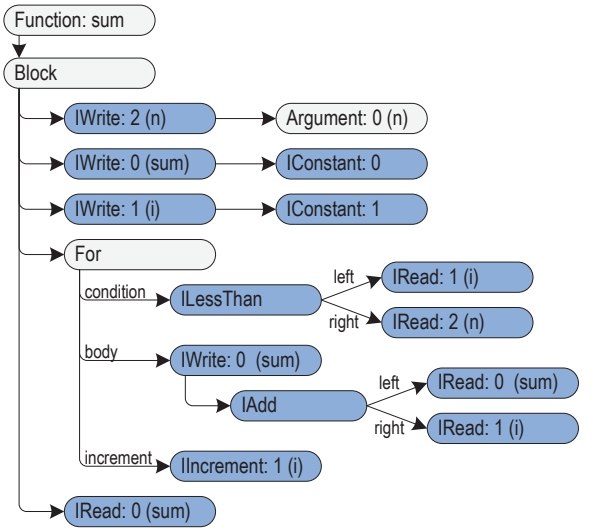

Figure 7. Example AST specialized to *integer*.

is not shown in the figure, creates a new node, replaces the `IAdd` node with this new node, performs the addition, and returns the non-*integer* result of this addition wrapped in a new `UnexpectedResult` exception.

Figure 7 shows the AST after the execution of the method, with all arithmetic operations typed to *integer*. The tree is in a stable state and can be compiled (see Section 4.6).

Assume now that the method `sum` is called with a larger value of `n`, still in the range of *integer*, but causing `sum` to overflow to *double*. The `IAdd` node detects this overflow, but is not able to perform the computation using type *double*. Instead, it replaces itself with a `DAdd` node. Subsequently, the nodes that write and read the local variable `sum` also replace themselves. Figure 8 shows the AST after all the type changes have been performed. The tree is in a stable state again and can be compiled.

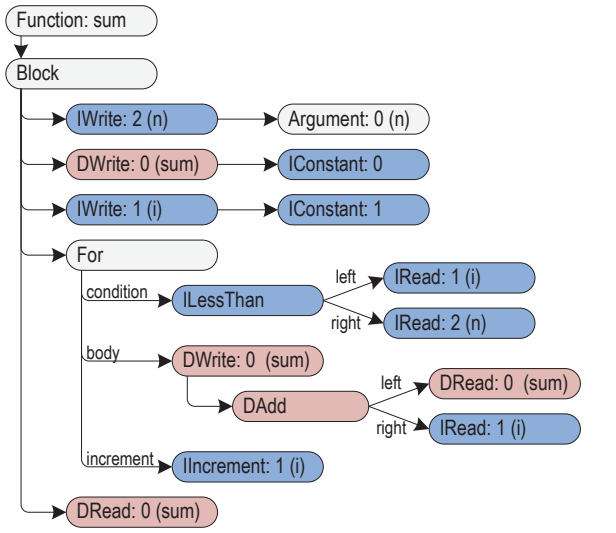

Figure 8. Example AST with local variable `sum` specialized to *double*.

3.5 DSL for Specializations

We continue using the example from Figure 4. Instead of looking at the whole loop, we want to concentrate on the `Add` operation. The `Add` operation replaces itself with more specialized forms when executed. Implementing such behavior in the host language is a repetitive and redundant task if performed for many operations. Most code shown in Figure 6 simply handles the rewriting and does not depend on guest language semantics.

A domain-specific language (DSL) enables us to specify guest language semantics in a more compact form. The DSL is integrated into the host language, i.e., it is an internal DSL without a separate syntax. In our implementation with Java as the host language, the DSL is expressed as Java annotations for classes and methods. Java annotations are an elegant way to attach metadata to Java code that can be processed both during compilation and at run time. We use a Java annotation processor at compile time to generate additional Java code from the annotations.

Figure 9 illustrates the development process: Our Java annotations are used in the source code (step 1). When the Java compiler is invoked on the source code (step 2), it sees the annotations and calls the annotation processor (step 3). The annotation processor iterates over all of our annotations (step 4) and generates Java code for the specialized nodes (step 5). The annotation processor notifies the Java compiler about the newly generated code, so it is compiled as well (step 6). The result is the executable code that combines the manually written and the automatically generated Java code (step 7).

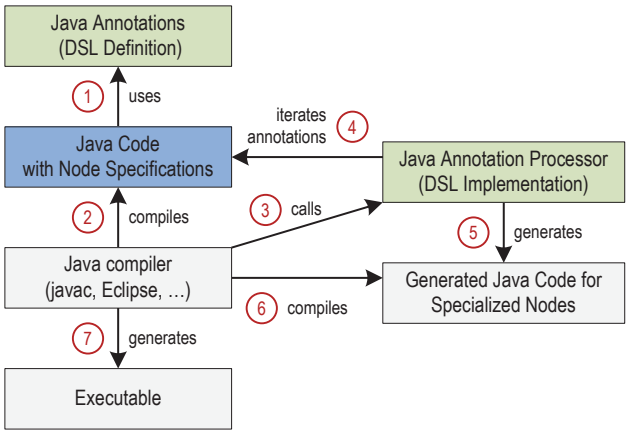

Figure 9. Development and build process for the annotation-based DSL.

```
@Specialization(rewriteOn=ArithmeticException.class)
int addInt(int a, int b) {
  return Math.addExact(a, b);
}

@Specialization
double addDouble(double a, double b) {
    return a + b;
}

@Generic
Object addGeneric(Frame f, Object a, Object b) {
  // Handling of String omitted for simplicity.
  Number aNum = Runtime.toNumber(f, a);
  Number bNum = Runtime.toNumber(f, b);
  return Double.valueOf(aNum.doubleValue() +
           bNum.doubleValue());
}
```

Figure 10. Type specializations defined using the annotation-based DSL.

Figure 10 presents how the `Add` node specializations are expressed by using annotations. Each method annotated by a `@Specialization` annotation represents one specialization. The method `addInt` implements the `IAdd` specialization behavior and analogously `addDouble` implements the `DAdd` specialization behavior. The `@Generic` annotation indicates the generic implementation of the operation. Owing to the complicated language semantics of JavaScript, the conversion of arbitrary values to numbers can even trigger the execution of user-defined conversion methods, i.e., additional JavaScript methods.

One node class is generated for each annotated method. The generated implementation follows the type transition order as indicated in Figure 2. An uninitialized version of the node is created by the annotation processor without further definition. We omit the handling of the `String` type for simplicity.

Method signatures can be used to infer which types are expected for the `left` and `right` child nodes. Calls to the typed execute methods are generated depending on that inference. If an `UnexpectedResult` exception is thrown by the typed execute methods, the node rewrites itself to a more generic state. The annotation processor knows from the specialization signatures how generic a specialization is. Therefore it can order them to achieve the desired specialization transitions. If a specialization signature is ambiguous, this order can also be defined manually. In the definition of the specializations, the transition can be triggered by exceptions as well as by explicitly specified guards. The latter is defined as a method that returns `false` if the node should be replaced.

The addition example highlights how we can define specializations for binary nodes using metadata. But this approach is not limited to the definition of binary nodes. It can be used to generate any kind of specializing nodes with any number of child nodes. We use this approach in a number of ways to reduce redundancy in guest language implementations.

Our code generator is tightly integrated with the source compiler and development environment for the host language. This enables using the same development tools and integrated development environment (e.g., Eclipse or NetBeans) to browse the generated code. The generated Java code is also available during debugging.

4. High Performance

The main overhead in our interpreter comes from dynamic dispatch between nodes. However, the targets of those dynamic dispatches are constant except when rewriting occurs. We count the number of invocations of a tree and reset the counter in the event of a node replacement. When the number of invocations on a stable tree exceeds a threshold, we speculate that the tree has reached its final state. We then start a special compilation for a specific tree where we assume every node in the tree remains unmodified. This way, the virtual dispatch between the `execute` nodes can be converted to a direct call, because the receiver is a constant. These direct calls are all inlined, forming one combined unit of compilation for a whole tree.

Because every node of the tree is assumed constant, many values in the tree can also be treated as constants. This helps the compiler produce efficient code for nodes that have constant parameters. Examples include the actual value of a constant node, the index of a local variable access (see Section 4.2), and the target of a direct call. This special compilation of the interpreter by assuming the nodes in the tree to be constants is an application of partial evaluation to generate compiled code from an interpreter definition [21].

Every control flow path leading to a node replacement is replaced with a deoptimization point. Those points invalidate the compiled code and continue executing the tree in the interpreter mode. A rewrite performed immediately after

```
if (x) {
  code block
} else {
  rewrite node
}
```

Figure 11. Injecting static information by node rewriting.

deoptimization resets the invocation counter of the tree, so that it is recompiled only after additional invocations cross the compilation threshold again.

The result after partial evaluation is represented in the compiler's high-level intermediate representation (IR). It embodies the combined semantics of all current nodes in the tree. This IR is then given for further optimizations to the host language optimizing compiler. The compiler then performs additional inlining, and in particular also global optimizations of the whole IR such as sharing of common operations between nodes or global code motion. The compiler can perform more global optimizations than node rewriting, since node rewrites are always local to one node. This means that a rewrite would only optimize a node based on the local information available at the specific node.

The compilation process need not begin at the root node. It can also be started only for a subtree, i.e., we compile often-executed loop structures before their containing methods. For this purpose, we have an invocation counter for a specific subtree that is also reset in case of node replacement.

4.1 Injecting Static Information

Speculative optimization allows a node to inject additional static information into the compilation process, yet safely fall back to deoptimized code in the rare case when this information is falsified. Figure 11 illustrates the prototypical form of such a node. When this node is interpreted, the condition `x` is dynamically checked on every node invocation. In the case where `x` is true, additional compiler optimizations (inside the `if`-branch) can be applied, yielding a faster (but specialized) implementation of the node's operation. When the node is compiled, the `else` block with the node rewrite is converted into a deoptimization point. This means that for any code that follows from this node, the compiler assumes `x` to be true. The compiler only inserts a conditional trigger of deoptimization in case the value of `x` is false. The dynamic check on `x` in the interpreter is transformed into static information preceded by a deoptimization point in the compiler. If `x` is an expression instead of a simple boolean variable, the compiler can still extract and use static information, e.g., a limited value range of an *integer* variable.

4.2 Local Variables

Reading and writing local variables is performed by guest languages via an index into a `Frame` object that contains a frame array holding the values. Local variable access nodes use node replacement in order to specialize on the type of a local variable. This allows for dynamic profiling of variable types while executing in the interpreter.

The performance of local variable access is critical for many guest languages. Therefore, it is essential that a local variable access in the compiled code after partial evaluation is fast. We ensure this by forcing an *escape analysis* (see for example [36]) of the array containing the values of local variables. This eliminates every access to the array and instead connects the read of the variable with the last write. This implicitly creates a static single assignment (SSA) form [16] for the local variables of the guest language. After conversion to SSA form, guest-language local variables have no performance disadvantage compared to host language local variables. In particular, the SSA form allows the host compiler to perform standard compiler optimizations such as constant folding or global value numbering for local variable expressions without a data flow analysis for the frame array. The actual frame array is never allocated in the compiled code, but only during deoptimization.

In essence, this optimization for local variables is just a targeted and well-defined application of escape analysis to the compiler IR resulting from partial evaluation. This guarantees predictable high performance for guest-language local variables. Use of these frame facilities for local variables is optional; guest-language implementations are not forced to use them.

4.3 Branch Probabilities

An optimizing dynamic compiler uses the probabilities of the targets of a branch to produce better performing machine code. The execution frequency of a certain block controls optimizations such as method inlining or tail duplication. Additionally, an optimized code layout can decrease the number of branch or instruction cache misses.

Branch probability information from executing the baseline execution of the host system is available in the compiler IR after partial evaluation. However, for a particular node, this information is the average of all executions of that kind of node. This can differ significantly from the information that would have been gathered for the particular node the partial evaluator is currently adding to the combined compiler IR.

Figure 12 shows an example AST node that implements the semantics of a conditional operation of the guest language. Depending on the evaluation of `condition`, either the value obtained by executing the `thenPart` or the `elsePart` node is returned. This is implemented using an `if` statement of our host language.

The branch probability for this `if` would be the average of all guest language conditional nodes, because they all share the same code and profiling is usually method based in the host system. A similar problem arises when guest language loop constructs are implemented using host language loop constructs. The host system facilities for loop frequency pro-

```
Object execute() {
  if (condition.execute() == Boolean.TRUE) {
    return thenPart.execute();
  } else {
    return elsePart.execute();
  }
}
```

Figure 12. Example of a conditional node that needs guest-level branch probabilities.

filing are no longer sufficient as they would always produce the average of all loops of a specific guest language operation.

Therefore, our system supports the injection of branch probabilities and loop frequencies by the guest language interpreter. Those values overwrite the existing values derived from the host system during partial evaluation. The code for measuring the values is only present in the interpreter and removed in the compiled code. This is an example where the interpreter has to go beyond only implementing the language semantics, but do additional profiling for the optimizing compiler. However, this additional profiling is optional, and the interface for providing the probability to the optimizing compiler is language agnostic.

Figure 13 shows the example conditional node extended with guest-level branch probabilities. It invokes static methods that are compiler directives, i.e., they behave differently in the interpreter and in compiled code. The method `inInterpreter` returns always `true` in the interpreter, but always `false` in compiled code. Therefore, the counter increments are only performed during interpretation. The method `injectBranchProbability` is a no-op in the interpreter, but attaches the provided probability to the `if`-node in the compiler.

4.4 Assumptions

Some guest languages need to register global assumptions about the system state in order to execute efficiently. Examples of such assumptions are: the current state of a Java class hierarchy, and the redefinition of well-known system library objects for JavaScript. Traditionally, every VM has a language-specific system of registering such dependencies of the compiled code. Our system provides a language-agnostic way of communicating assumptions to the runtime system.

A guest language interpreter can request a global variable object from the runtime that encapsulates a stable boolean value. The initial value is `true` and the variable can only be modified a single time to be `false`. In the interpreter, a node can check such a variable dynamically for its value. During partial evaluation, we assume the value of the variable to be stable. We replace the access to the variable with the current constant and register a dependency of the compiled code on the value. If subsequently the value changes, any compiled code relying on the value is deoptimized. This way, we

```
Object execute() {
  if (injectBranchProbability(
        thenCounter / (thenCounter + elseCounter),
        condition.execute() == Boolean.TRUE)) {

    if (inInterpreter()) {
      thenCounter++;
    }
    return thenPart.execute();
  } else {
    if (inInterpreter()) {
      elseCounter++;
    }
    return elsePart.execute();
  }
}
```

Figure 13. Example conditional node extended with guest-level branch probabilities.

ensure that the check of the stable variable has no overhead in compiled code and the system still executes correctly in all cases.

4.5 Flexible Runtime Call Inlining

After inlining the `execute` methods of nodes and the escape analysis of the frame, there can still be calls remaining in the compiler IR. In particular, for implementing more complex semantics, a node can call helper methods from its own methods. Those calls are equivalent to runtime calls in traditional dynamic compilation systems. One specific advantage of our system is that the code behind those runtime calls is neither native code nor hand-written assembler code, but again just code written in the host language. The host language optimizing compiler can inline those calls.

Whether to inline such a runtime call can be decided individually for every operation in the system. This allows us to use probability information to guide that decision. A runtime call on a particularly hot path is more likely to be inlined than a runtime call on a cold path. The decision of how much of the code of an operation is inlined into the current compilation is highly flexible.

4.6 Compilation Example

This section continues the example of Section 3.4 and shows how the method is compiled to optimized machine code. Recall that we showed the AST of a simple JavaScript method (see Figure 4) at two stable states: nodes specialized to type *integer* (see Figure 7) and *double* (see Figure 8).

Assume that the example method is called frequently with a small value for argument `n`, i.e., the AST specialized to *integer* is executed often without being rewritten. Even though the interpreter is optimized and specialized, the interpreter dispatch overhead remains. It is not possible to reach excellent peak performance without compilation. Partial evaluation eliminates the interpretation overhead, resulting in the Intel x86 machine code shown in Figure 14. The figure shows all of the machine code for the loop, but omits the pro-

```
    access argument 0 (JavaScript variable n)
    type check that n is of type Integer
      deoptimize if check fails
    unbox n into register esi
    mov       eax, 1     // JavaScript variable i
    mov       ebx, 0     // JavaScript variable sum
    jmp       L2
L1: mov       edx, ebx
    add       edx, eax   // Writes the overflow flag
    jo        L3    // Jump if overflow flag is true
    incl      eax
    safepoint            // Host-specific yield code
    mov       ebx, edx
L2: cmp       eax, esi
    jlt       L1
    box ebx (sum) into Integer object
    return boxed sum

L3: call      deoptimize
```

Figure 14. Example machine code specialized to *integer*.

```
ForNode.execute [bci: 39]
  local 0 (this) = const ForNode@1005245720
  local 1 (frame) = vobject 0
BlockNode.execute [bci: 20]
  local 0 (this) = const BlockNode@169916747
  local 1 (frame) = vobject 0
  local 2 (i) = const 3
FunctionNode.execute [bci: 5]
  local 0 (this) = dead
  local 1 (frame) = dead
OptimizedCallTarget.executeHelper [bci: 15]
  local 0 (this) = dead
  local 1 (arguments) = dead
  local 2 (frame) = dead

vobject 0 Frame
  arguments = r8
  primitiveLocals = vobject 1
  objectLocals = vobject 2
vobject 1 long[]
  0 = ebx   // JavaScript variable i
  1 = eax   // JavaScript variable sum
  2 = esi   // JavaScript variable n
vobject 2 Object[]
  0 = const null
  1 = const null
  2 = const null
```

Figure 15. Deoptimization information to restore AST interpreter frames and allocate escape-analyzed objects.

logue and epilogue of the loop. The prologue accesses the first method argument, which is passed in as a boxed object. It checks that the argument is of the box type `Integer`; if not, it deoptimizes to the AST interpreter in order to change the type specialization. The unboxed primitive *integer* value is stored in register `esi`.

The loop performs an addition to accumulate the `sum` variable. It is specialized to *integer*, so the normal integer `add` instruction is used. However, the normal behavior of the Intel instruction set is to wrap around to negative numbers

```
    access argument 0 (JavaScript variable n)
    type check that n is of type Integer
      deoptimize if check fails
    unbox n into register esi
    mov       eax, 1     // JavaScript variable i
    xorpd     xmm0, xmm0 // JavaScript variable sum
    jmp       L2
L1: cvtsi2sdl xmm1, eax
    addsd     xmm0, xmm1
    incl      eax
    safepoint            // Host-specific yield code
L2: cmp       eax, esi
    jlt       L1
    box xmm0 (sum) into Double object
    return boxed sum
```

Figure 16. Example machine code specialized to *double*.

if the addition overflows. Therefore, we need a conditional jump after the addition; if an overflow happens, we deoptimize. Deoptimization is a call to a runtime function, i.e., there is no need to emit any more machine code than a single `call` instruction. Metadata associated with the call is used to restore the AST interpreter stack frames, as described later in this section. The increment of the loop variable `i` is also specialized to *integer*, but does not need an overflow check since the lower and upper bound are known to be *integer* values. After the loop, the accumulated `sum` is converted to a boxed `Integer` object and returned.

The machine code for the loop is short and does not contain any type checks, object allocations, or pointer tagging operations. Some of the `mov` instructions may seem superfluous at the first glance, but exist to allow for deoptimization. Since deoptimization needs to restore the AST interpreter stack frames with values *before* the overflowing operation, the input values of the addition need to be kept alive for potential use during deoptimization.

Figure 15 shows the metadata recorded for deoptimization. It is associated with the program counter of the call instruction, i.e., a metadata table lookup with this program counter returns the deoptimization information serialized in a compressed form (see for example [37, 54]). Four interpreter stack frames need to be restored: the helper method where compilation was started and that is responsible for allocating the AST interpreter stack frame; and three `execute` methods of the AST. Even though deoptimization happens nested more deeply during the execution of the addition node, it is sufficient to restart interpretation at the beginning of the current loop iteration. This keeps the metadata small.

The AST interpreter operates on a heap-allocated `Frame` object. For improved performance, we apply escape analysis of this object during compilation (see Section 4.2). This `Frame` object, together with the actual data arrays that store the value of local variables, need to be allocated during deoptimization. The necessary information for this allocation is stored in the `vobject` sections of the metadata.

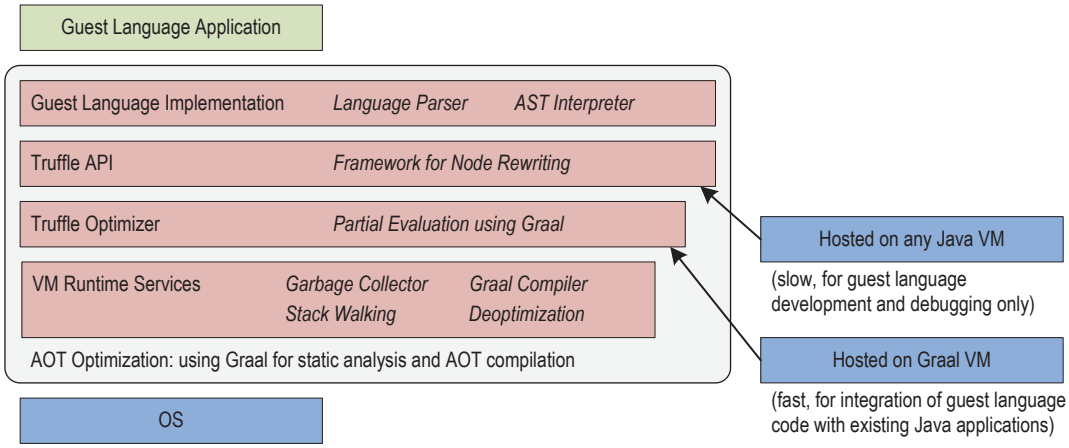

Figure 17. Detailed system structure of our Java-based prototype, with additional deployment strategies on any Java VM and our Graal VM.

The AST nodes, i.e., the receiver objects of the `execute` methods, are provided as literal constants in the metadata. Since the machine code does not access the nodes at all, it is not necessary to keep them in a register. The values of local variables in `execute` methods that are no longer accessible by the AST interpreter need not be restored during deoptimization. This is represented as `dead` in the figure.

Note that all primitive JavaScript local variables are stored in a `long[]` array (the `vobject 1` in the figure), regardless of their actual type (*integer* or *double* in the case of JavaScript). The AST nodes that access the frame object are all typed, i.e., all reads and writes are specialized to the same type. This ensures that the raw bits stored into the `long[]` array are always interpreted as the correct type. References cannot be mixed with primitive data because the garbage collector needs to distinguish them, so we have a separate `Object[]` array, with the same length and indexed with the same local variable numbers.

Should the example optimized machine code be called with a high value for `n`, the `add` instruction sets the processor's overflow flag to `true`. The succeeding conditional jump triggers deoptimization, which invalidates the machine code so that it is no longer entered on subsequent executions of this method. Execution of the AST continues in the interpreter, which replaces the `IAdd` node with a `DAdd`. Later, when this version of the AST is considered stable, a new compilation of the guest language method with the new AST is triggered.

Figure 16 shows the machine code for the method type-specialized to *double*. In contrast to Figure 14, it no longer contains an *integer* addition that can overflow. The `addsd` instruction that performs the *double* addition always succeeds. The prologue code before the loop is unchanged. The parameter `n` is still assumed to be of type *integer*, so deoptimization can happen in the prologue. In the epilogue, a boxed `Double` object is created to return the result.

5. Implementation and Deployment

Our prototype implementation of guest languages and the host services is written in a subset of Java. To allow ahead-of-time (AOT) compilation, we do not use Java features such as reflection and dynamic class loading that would prevent a whole-program static analysis determining the methods and fields in use. Since we use Java as a systems programming language [19], omitting these features is not too restrictive. AOT compilation is performed by the Graal compiler, i.e., we use the same compiler for ahead-of-time compilation that we also use for dynamic compilation. This process produces a runtime environment that is able to run the languages for which an implementation was provided during AOT compilation. This is similar to the bootstrapping process of metacircular VMs such as Maxine [69] or Jikes [2]. However, we want to note that our system is not a metacircular Java VM, since we are not able to load and execute new Java bytecode at run time. We are only able to load and execute guest language source code, as defined by the guest language implementation.

Figure 17 shows a refinement of the system structure presented in Figure 1. The *host services* are split up into three layers:

- Truffle API: This is the API to implement the AST interpreter of guest languages. It provides the handling of guest language frames and local variables, as well as means for nodes to rewrite themselves. This is the only public API that a guest language implementation needs; the lower levels are invisible to the guest language implementation.
- Truffle Optimizer: this includes the partial evaluation, and is implemented on top of the API that the Graal compiler [45] provides.
- VM Runtime Services: This layer provides the basic VM services such as garbage collection, exception handling,

and deoptimization. It also includes Graal, our aggressively optimizing compiler for Java, written in Java.

This layered approach allows two alternative deployment scenarios for guest language implementations. In both cases, no AOT compilation is performed since the guest language implementation runs as an application on a Java VM.

1. Hosted on any Java VM: Since all of our implementation is pure Java code, the guest language implementation can run on any unmodified Java VM. However, such a Java VM does not provide an API to its dynamic compiler, so partial evaluation is not possible and the guest language is only interpreted. Still, this scenario is useful for development and debugging of the guest language implementation since it minimizes the amount of framework code that is in use. Additionally, it can be used for deployment on legacy systems where the underlying Java VM cannot be changed.
2. Hosted on Graal VM: Graal VM is a modification of the Java HotSpot VM that uses Graal as its dynamic compiler. Additionally, it provides an API to access the compiler, so guest language code can run at full speed with partial evaluation. This scenario is especially useful when integrating guest language code with existing Java applications, i.e., to invoke guest language code from Java using the existing Java scripting API [35].

6. A Wide Variety of Languages

The crucial question that we need to answer is: "Which languages are amenable to the Truffle approach?" In order to answer this question, we must undertake a variety of implementations for disparate languages. This approach cannot yield an upper bound of generality, but with each implementation the known lower bound is raised. If a new language has a totally new feature or is of a radically different paradigm from existing languages, then it may need new compiler optimizations to get good performance, but for features close to those in languages already supported, a direct implementation should be able to reuse the existing machinery. As such, we aim to implement diverse languages, creating in the process a broad toolkit of techniques.

Existing languages frequently fall into a number of families, and as such share significant features. Implementing them in the same framework allows significant parts of language implementations to be shared. Even when for pragmatic reasons they cannot be shared, existing techniques can inform the implementation of new languages, expediting the implementation process. We discuss in this section the particular challenges and features that are required (and, where applicable, implemented) for each of these languages.

6.1 JavaScript

JavaScript is an excellent language to use as a testbed, since there exist several correct, high-performance implementations and some fairly comprehensive test suites. Truffle's flexibility allows our implementation of JavaScript to be mostly straightforward: JavaScript objects are implemented as a particular class of Java objects; all JavaScript values are representable as subtypes of Java's `Object` type; and most of the of type checks required for basic JavaScript operations are implemented using the `instanceof` operator.

Rewriting is used in two ways: type specialization of generic operations and inline caching. Our type specialization technique is discussed in Section 4.6. Inline caching (see Section 3.1) optimizes operations that are frequently performed on objects of a specific type. Each object's memory layout is represented as a *shape* object, which is shared by all similarly-shaped objects. Inline caching involves replacing a member access node with a version that is specialized for a specific shape. The specialized node checks the actual shape against the expected shape and, on success, executes the simplified access for this shape. Because cached shapes are immutable in these generated nodes, machine code generated by Graal is highly efficient, requiring no dynamic lookups.

This general inline caching technique is extended for JavaScript by making it aware of prototypes. In JavaScript, an object's members are inherited from other objects in a chain of *prototypes*. This system is derived from Self [65], but simpler, as an object may have only one prototype. The simplest way of supporting this system would be to cache each level of the prototype chain: A cached check of the object itself would reveal that a member is not there, and so move on to a cached check of its prototype, etc. We expedite this by moving the prototype relationship into the shapes themselves; any two objects with the same shape are assured to have the same prototypes, and so a cache over an object's shape is equivalent to a cache over its entire prototype chain.

Truffle also provides the possibility for a novel technique for dealing with `eval`, a function provided by many languages to execute strings as code. Especially for JavaScript, it is widely used and often poorly understood [48]. Although many JavaScript implementations cache particular *values* of evaluated strings, rewriting may additionally be used to specialize on particular *patterns* of evaluated strings. Since the vast majority of `eval` call sites encounter strings of a small number of simple patterns, this technique can allow for a highly efficient implementation of `eval` in common cases.

6.2 Ruby

Ruby's design was strongly influenced by Smalltalk. It shares many of the performance challenges of Smalltalk (and Self). Almost every operation in Ruby is the result of a method invocation. Hence, efficient method invocation is key, and the best kind of invocation is no invocation, i.e., the target is inlined into the caller. Truffle's inline caching, along with optimistic movement of the shape checks and method inlining by Graal, allows us to achieve this.

In Ruby, any method can be redefined, including basic arithmetic operations. The way we handle uncommon paths

based on global assumptions via deoptimization (see Section 4.4) allows us to handle redefinitions by registering a dependence on the existing definition of a basic method. If the method is redefined, then the compiled code is invalidated and execution resumes in the interpreter.

The Ruby language includes a feature called *fibers*, which is equivalent to one-shot continuations or coroutines (storing and restoring the current execution context). There are implementations of these concepts in Java that rely on bytecode instrumentation [4], but these incur a significant runtime overhead. In prior work we have developed native VM support for continuations [62] and coroutines [63] in the HotSpot VM, which can be leveraged by our system to provide fiber operations with $O(1)$ complexity.

6.3 Python

Similar to other dynamic languages, Python provides a range of integrated collection types (lists, sets, arrays, etc.). It supports multiple inheritance, and subclassing even of primitive data types. The shape concept, as introduced for JavaScript, fits the requirements of a Python implementation well.

Python allows a method to access the call stack via `sys._getframe(n)` and access to local variables of the caller frame via `sys._getframe(n).f_locals`. Implementing access to caller frames in the AST interpreter is simple: the frame objects can be chained by passing in the caller frame in a method call. Accessing the caller frame is a feature that we think does not need to be fast, i.e., supported in a compiled method. However, simply not compiling a method that uses `_getframe` is not enough; it can access frames of compiled methods already on the stack. Since the frame object is elided by escape analysis during compilation, it does not exist in memory and cannot be part of a frame chain. However, we can use the deoptimization metadata to reconstruct the frame object. Accessing (and possibly changing) the local variables of a method on the call stack requires the following steps: walk the Java stack to the corresponding compiled method activation frame; reconstruct the frame object using the deoptimization metadata for the compiled method; access and change the local variable value in the frame object; mark the compiled method as deoptimized, so that the execution continues in the AST interpreter when control returns to the method whose frame object was accessed. In summary, accessing the frame of a compiled method requires support from the VM runtime services. However, it does not require more metadata and infrastructure than deoptimization, i.e., all information is already readily available.

6.4 Technical Computing: J and R

There is no widely accepted definition of "technical computing", but the one we use is the application of numerical techniques to business and technical problems by domain experts. It includes languages such as R and J. As pointed out by Morandat et al. [43], R has been underserved by the implementation community. It presents many implementation challenges, some of which are unique to the language, and many of which are common to most or all technical computing languages. Its handling of whole-array operations, influenced strongly by the design of APL [32], is one such area. Many optimizations were devised for APL in the 1970s and 1980s that are largely unknown outside the APL community. Additionally, this style of computation is amenable to large-scale parallelization [60]. The convenient expression of parallelism is an industry-wide challenge, and much work has been expended in this area since the early work on APL optimization. The early APL work also did not anticipate the radical change in memory hierarchy of the intervening decades. Combining the more recent work on parallelization and memory exploitation with the earlier array optimization work seems like a fertile combination for technical computing.

To gain insight into this area, we have undertaken an implementation of J, an APL-derived language [29]. Although not as widely used, J encapsulates the array-processing style with minimal additional complication. An implementation of J serves as an experimentation lab for array processing techniques for technical computing.

The R programming language incorporates many of the concepts of array programming, but additionally has an object-oriented and list-processing nature with influences from Smalltalk and Scheme. Among the challenges facing an R implementation are: avoiding unnecessary copying of vectors; complex method invocation semantics; reflection (especially on activations); efficient interfaces to legacy libraries in C and Fortran 77; and partly-lazy, partly-functional semantics [43]. However, long-running computations in R (and technical computing applications in other languages) offer the possibility of extensive run-time analysis and optimization without concern for interactive performance. We believe that deep inlining in hot loops expose the connections needed by the compiler to optimize these challenging features. For example, whole-loop analysis can be used to transform operation sequences to eliminate unnecessary array copies [67].

Graal is also being used within the context of OpenJDK Project Sumatra [46], to generate code for GPUs. A GPU backend for array languages such as J and R, and for array-processing libraries for other languages (e.g., River Trail [50]) offers the potential of high-performance parallel execution and is something we intend to pursue in the near future.

6.5 Functional Languages

Our approach may be suited to the implementation of pure, functional languages but we have not pursued this in detail. Some of the elements of these languages are present in R, which is mostly side-effect-free at the level of methods, and is partly lazy [43]. Challenges are in the area of tail calls and continuations. We believe that both can be implemented in

plain Java at the AST interpreter level (using for example exceptions to unwind the stack for tail calls), but that it is easier and more robust with support from the host VM services. Both tail calls [57] and continuations [62] have been prototyped for the Java HotSpot VM, but to the best of our knowledge no efforts are underway to add them to the Java specification.

7. Related Work

7.1 PyPy

Bolz and Rigo [6] suggested that VMs for dynamic languages should be generated from simple interpreters written in a high-level but less dynamic language. The idea is to write the language specification as an interpreter that serves as the input for a translation toolchain that generates a custom VM and tracing dynamic compiler using partial evaluation techniques. The result is either a custom VM implemented in C or a layer on top of an object-oriented VM. The PyPy project puts this meta-programming approach into practice and strives for the goal of easing efficient language implementation by optimizing a bytecode interpreter written in a high-level language. The heart of PyPy is the RPython language, an analyzable restricted subset of Python that is statically typed using type inference, and can be compiled to efficient low-level code [3]. PyPy implements a Python VM written in RPython. Both the Python interpreter and the RPython VM itself are written in RPython. The RPython toolchain translates the interpreter and all necessary support code to C code. It also generates a tracing compiler, for which the interpreter author must insert hints in form of run-time calls and annotations [8, 49]. Some of those hints are required by compiler generator to identify program counter variables and possible entry and exit points for the tracer. Other hints expose constant folding opportunities and can be used to implement run-time feedback [9]. The tracing compiler performs online partial evaluation on traces as a form of escape analysis to remove allocations, including boxed representations of primitive values [10]. PyPy was explicitly designed with multi-language support in mind. Currently, there are VMs for Python, Ruby, Converge, and Prolog [7]; other language implementations are in development.

Bolz and Rigo [6] and later Bolz and Tratt [7] stated that in general-purpose object-oriented VMs, the compiler is optimized for the language, or group of languages, it was designed for. If a language's semantics are significantly different and thus do not fit the VM well, it will perform poorly—despite a highly optimized underlying VM. Various language implementations running atop the Java HotSpot VM and Microsoft's .NET CLR seem to confirm this. Although the introduction of *invokedynamic* [51] brought considerable performance improvements, highly optimized hand-crafted VMs written in a low-level language still have the edge over Java-based runtimes [7]. Our system tries to close the performance gap through run-time partial evaluation of an interpreter optimized for the target language. VM extensions allow us to efficiently support language features such as tail calls and continuations.

7.2 Self-Optimizing Interpreters

Most interpreters use some form of bytecode to represent the executed program, mainly because of concerns about execution speed. New techniques for efficiently executing applications written in dynamic programming languages, however, increasingly rely on modifications to the internal representation of the running code. This is possible to a certain degree with bytecode interpreters [11, 12]. However, we argued in [72] that AST interpreters allow for much more extensive modifications.

There have been various techniques devised to improve interpretation performance while preserving the simplicity, directness and portability of an interpreter. Examples include the work of Casey et al. [13], Ertl and Gregg [17], Gagnon and Hendren [22], Piumarta and Riccardi [47], Shi et al. [59], Thibault et al. [64]. However, because a compiler analyzes a much larger fragment of the program (once inlining has been applied) it can perform many global optimizations that are beyond the reach of interpreters.

7.3 Partial Evaluation

Partial evaluation has a long history and has been extensively studied for functional languages. Using partial evaluation to derive compiled code from an interpreter and the source code of the application was conceived by Futamura [21] and is called the first Futamura projection. Repeating the specialization process results in a compiler (second Futamura projection) and a tool to generate compilers (third Futamura projection). The technique used in Truffle can be conceived as a special form of the first Futamura projection, where the interpreter is already specialized for the source code, and compiled code is derived from the interpreter only for hot and stable code.

Partial evaluators can be classified as offline and online [33]. The offline strategy performs binding time analysis before partial evaluation that annotates the program with specialization directions. Inputs are declared as either known at specialization time (static) or unknown (dynamic). Ideally, the analysis would also guarantee that the specialization always terminates. Automatic partial evaluators usually use this strategy. The online strategy makes decisions on what to specialize during the specialization process. This approach can produce better results [33, 53] but is usually slower and has termination difficulties if the program contains iterative or recursive loops [33, 40]. Therefore, online partial evaluators usually rely on programmer-supplied annotations to direct the specialization.

Since we perform online partial evaluation at run time, we want to guarantee that our algorithm always terminates. We achieve this by clearly limiting the scope of partial evaluation to the AST and by prohibiting recursion and changes

to the AST state in partially evaluated code (unless guarded by deoptimization). Our main use for partial evaluation is to eliminate the biggest optimization barrier in our system, the virtual dispatch.

Partial evaluation as a means to specialize Java programs has been pursued by several research efforts. Schultz et al. decided to translate Java code to C code that serves as the input to an offline partial evaluator. The residual code is either compiled by a C compiler [55] or translated back to Java bytecode [56]. So-called specialization classes declare optimization opportunities for object-oriented design patterns such as visitor and strategy. They reported a significant speedup on selected benchmarks. Masuhara and Yonezawa [41] proposed automatic run-time bytecode specialization for a non-object-oriented subset of Java with an offline strategy. Affeldt et al. [1] extended this system to include object-oriented features with a focus on correctness. The speedups achieved by this system were significant for non-object-oriented code but less substantial for object-oriented code. Shali and Cook [58] implemented an offline-style online partial evaluator in a modified Java compiler. Their partial evaluator derives residual code from invariants manually specified using source code annotations. It strictly differentiates between compile-time and run-time variables: compile-time variables are completely eliminated from residual code. Generated code is as efficient as that of Schultz et al. [56] They illustrated how their partial evaluator can be used to optimize regular expressions when the pattern is known at compile time. While we also have compile-time variables that are eliminated from the code, we can undo this specialization at any time thanks to recorded deoptimization information. Therefore we do not need to specialize all possible branches that depend on compile-time values.

Partial evaluators targeting Java suffer from the fact that Java bytecode cannot fully express all optimizations [55]. The Truffle stack gives us more control because we do not need to re-generate bytecodes for partially evaluated guest language methods; instead, we only work on Graal compiler IR.

7.4 Other Approaches to High Performance

The use of speculation and deoptimization as a performance-enhancing technique was introduced in Self [27], to allow debugging of optimized code. Since then it has been applied to array-bound check elimination [71], escape analysis and stack allocation [36], object fusing [68], boxing elimination [15] and partial redundancy elimination [44].

JRuby [34] is an implementation of Ruby that compiles to Java bytecode, run on a Java VM. The challenge here is to map the semantics of a dynamically-typed language onto a statically-typed instruction set and get good performance. The Java VM knows nothing of the semantics of Ruby and there is no mechanism to communicate optimization information through Java bytecode. The recent addition of *invokedynamic* to the Java bytecode set [51] has made the implementation of method dispatch more efficient. Challenges remain in eliminating the boxing of numbers, efficient handling of method redefinition of basic operations, and elsewhere.

Another approach is to add support for dynamic languages to an existing high-performance static-language VM [14, 31].

A number of projects have attempted to use LLVM [38] as a compiler for high-level managed languages, such as Rubinius and MacRuby for Ruby [39, 52], Unladen Swallow for Python [66], Shark and VMKit for Java [5, 23], and McVM for MATLAB [24]. These implementations have to provide a translator from the guest languages' high-level semantics to the low-level semantics of LLVM IR. In contrast, our approach requires only an AST interpreter; our system can be thought of as a *High-Level Virtual Machine* (HLVM).

7.5 Multi-Language Systems

Wolczko et al. [70] described an approach for high performance implementations of multiple languages (Self, Smalltalk, Java) atop the Self VM. The Self VM was written in C++, and the guest languages were implemented in Self, either by translation to Self source (Smalltalk) or bytecode (Java). This approach relied on the minimality and flexibility of Self, and the deep inlining performed by the Self VM's inlining compiler. A similar approach is taken by the Java implementation in Smalltalk/X [61]. The Virtual Virtual Machine incorporated an architecture for multi-lingual VMs [18].

7.6 Metacircular VMs

In traditional VMs, the host (implementation) and guest languages are unrelated, and the host language is usually lower-level than the guest language. In contrast, metacircular VMs are written in the guest language, which allows for sharing of components between host and guest systems. The "boot image" used to start the system is constructed ahead of time, usually using an alternate implementation of the language.

The Jikes Research VM [2] and the Maxine VM [69] are examples for metacircular Java VMs. Both are focused mainly at the research community. We use many ideas from metacircular VMs for our all-Java prototype of VM runtime services. However, our goal is not a metacircular Java VM since we are not able to load and execute Java applications. We are only able to load and execute guest language source code, as defined by the guest language implementation.

8. Conclusions

We presented a new approach to VM construction based on a combination of node rewriting during AST interpretation, an optimizing compiler, and deoptimization. The compiler exploits the structure of the interpreter, in effect partially evaluating the interpreter when generating code. Using this approach we are implementing a variety of languages that to

date have mostly not had optimizing compilers. Each implementation consists of a language-specific AST interpreter; the compiler is reused for all languages. We have adopted a layered implementation approach, using Java as an implementation language. Our interpreters run faithfully, but perhaps with modest performance, on any compliant Java VM. When combined with the Graal compiler, we observe significant performance improvements. A language implementation consisting of a Truffle interpreter, Graal compiler, and associated runtime can also be compiled ahead of time with Graal to realize a standalone language VM, not requiring a Java VM except during bootstrapping. In such a VM, Graal is also used as a dynamic compiler.

We achieve high performance from a combination of techniques:

- Node rewriting specializes the AST for the actual types used, and can result in the elision of unnecessary generality, e.g., boxing, complex dispatch.
- Compilation by automatic partial evaluation leads to highly optimized machine code without the need of writing a language-specific dynamic compiler.
- Deoptimization from machine code back to the AST interpreter handles speculation failures.

Source code for the Graal compiler, the Truffle interpretation framework, and sample language implementations is available at the OpenJDK Project Graal site [45].

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs, the Institute for System Software at the Johannes Kepler University Linz, and our collaborators at Purdue University, TU Dortmund, and UC Irvine for their support and contributions. We especially thank Peter Kessler, Michael Van De Vanter, Chris Seaton, Stephen Kell, and Michael Haupt for feedback on this paper.

The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

References

[1] R. Affeldt, H. Masuhara, E. Sumii, and A. Yonezawa. Supporting objects in run-time bytecode specialization. In *Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 50–60. ACM Press, 2002. doi: 10.1145/568173.568179.

[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. doi: 10.1147/sj.442.0399.

[3] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the Dynamic Languages Symposium*, pages 53–64. ACM Press, 2007. doi: 10.1145/1297081.1297091.

[4] Apache Commons. Javaflow, 2009. URL http://commons.apache.org/sandbox/javaflow/.

[5] G. Benson. Zero and Shark: a zero-assembly port of OpenJDK, 2009. URL http://today.java.net/pub/a/today/2009/05/21/zero-and-shark-openjdk-port.html.

[6] C. F. Bolz and A. Rigo. How to not write virtual machines for dynamic languages. In *Proceedings of the Workshop on Dynamic Languages and Applications*, 2007.

[7] C. F. Bolz and L. Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 2013. doi: 10.1016/j.scico.2013.02.001.

[8] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM Press, 2009. doi: 10.1145/1565824.1565827.

[9] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 9:1–9:8. ACM Press, 2011. doi: 10.1145/2069172.2069181.

[10] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 43–52. ACM Press, 2011. doi: 10.1145/1929501.1929508.

[11] S. Brunthaler. Efficient interpretation using quickening. In *Proceedings of the Dynamic Languages Symposium*, pages 1–14. ACM Press, 2010. doi: 10.1145/1869631.1869633.

[12] S. Brunthaler. Inline caching meets quickening. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 429–451. Springer-Verlag, 2010. doi: 10.1007/978-3-642-14107-2_21.

[13] K. Casey, M. A. Ertl, and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems*, 29(6), 2007. doi: 10.1145/1286821.1286828.

[14] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 195–212. ACM Press, 2012. doi: 10.1145/2384616.2384631.

[15] Y. Chiba. Redundant boxing elimination by a dynamic compiler for Java. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 215–220. ACM Press, 2007. doi: 10.1145/1294325.1294355.

[16] 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. doi: 10.1145/115372.115320.

[17] M. A. Ertl and D. Gregg. Combining stack caching with dynamic superinstructions. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators*, pages 7–14. ACM Press, 2004. doi: 10.1145/1059579.1059583.

[18] B. Folliot, I. Piumarta, and F. Riccardi. A dynamically configurable, multi-language execution platform. In *Proceedings of the European Workshop on Support for Composing Distributed Applications*, pages 175–181. ACM Press, 1998. doi: 10.1145/319195.319222.

[19] 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. doi: 10.1145/1508293.1508305.

[20] B. Fulgham. The computer language benchmarks game, 2013. URL http://benchmarksgame.alioth.debian.org/.

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

[22] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Proceedings of the International Conference on Compiler Construction*, pages 170–184. Springer-Verlag, 2003. doi: 10.1007/3-540-36579-6_13.

[23] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: A substrate for managed runtime environments. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 51–62, 2010. doi: 10.1145/1735997.1736006.

[24] L. Hendren, J. Doherty, A. Dubrau, R. Garg, N. Lameed, S. Radpour, A. Aslam, T. Aslam, A. Casey, M. C. Boisvert, J. Li, C. Verbrugge, and O. S. Belanger. McLAB: enabling programming language, compiler and software engineering research for MATLAB. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 195–196. ACM Press, 2011. doi: 10.1145/2048147.2048203.

[25] 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. doi: 10.1145/178243.178478.

[26] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.

[27] 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. doi: 10.1145/143095.143114.

[28] HotSpot JVM. Java version history (J2SE 1.3), 2013. URL http://en.wikipedia.org/wiki/Java_version_history.

[29] R. K. W. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney. APL\? In *Conference Proceedings on APL 90: for the future*, pages 192–200. ACM Press, 1990. doi: 10.1145/97811.97845.

[30] IBM. Java 2 Platform, Standard Edition, 2013. URL https://www.ibm.com/developerworks/java/jdk/.

[31] K. Ishizaki, T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 169–180. ACM Press, 2012. doi: 10.1145/2151024.2151047.

[32] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962. ISBN 0-471430-14-5.

[33] N. D. Jones and A. J. Glenstrup. Program generation, termination, and binding-time analysis. In *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 1–31. Springer-Verlag, 2002. doi: 10.1007/3-540-45821-2_1.

[34] JRuby. JRuby: The Ruby programming language on the JVM, 2013. URL http://jruby.org/.

[35] JSR 223. JSR 223: Scripting for the Java™ platform, 2006. URL http://www.jcp.org/en/jsr/detail?id=223.

[36] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 111–120. ACM Press, 2005. doi: 10.1145/1064979.1064996.

[37] 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. doi: 10.1109/CGO.2007.34.

[38] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86. IEEE Computer Society, 2004. doi: 10.1109/CGO.2004.1281665.

[39] MacRuby. MacRuby, 2013. URL http://macruby.org/.

[40] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, 1992.

[41] H. Masuhara and A. Yonezawa. A portable-approach to dynamic optimization in run-time specialization. *New Generation Computing*, 20(1):101–124, 2002. doi: 10.1007/BF03037261.

[42] E. Meijer and J. Gough. Technical overview of the Common Language Runtime, 2000. URL https://research.microsoft.com/en-us/um/people/emeijer/papers/clr.pdf.

[43] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language—Objects and functions for

data analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 104–131. Springer-Verlag, 2012. doi: 10.1007/978-3-642-31057-7_6.

[44] R. Odaira and K. Hiraki. Sentinel PRE: Hoisting beyond exception dependency with dynamic deoptimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 328–338. IEEE Computer Society, 2005. doi: 10.1109/CGO.2005.32.

[45] OpenJDK. Graal project, 2013. URL http://openjdk.java.net/projects/graal.

[46] OpenJDK. Project Sumatra, 2013. URL http://openjdk.java.net/projects/sumatra/.

[47] 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.

[48] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 52–78. Springer-Verlag, 2011. doi: 10.1007/978-3-642-22655-7_4.

[49] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 944–953. ACM Press, 2006. doi: 10.1145/1176617.1176753.

[50] River Trail. Parallel EcmaScript (River Trail) API, 2013. URL http://wiki.ecmascript.org/doku.php?id=strawman:data_parallelism.

[51] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, 2009.

[52] Rubinius. Rubinius: Use Ruby, 2013. URL http://rubini.us/.

[53] E. Ruf and D. Weise. Opportunities for online partial evaluation. Technical report, Stanford University, 1992.

[54] D. Schneider and C. F. Bolz. The efficient handling of guards in the design of RPython's tracing JIT. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, pages 3–12. ACM Press, 2012. doi: 10.1145/2414740.2414743.

[55] U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 367–390. Springer-Verlag, 1999. doi: 10.1007/3-540-48743-3_17.

[56] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. In *ACM Transactions on Programming Languages and Systems*, pages 452–499. ACM Press, 2003. doi: 10.1145/778559.778561.

[57] A. Schwaighofer. Tail call optimization in the Java HotSpot VM. Master's thesis, Johannes Kepler University Linz, 2009.

[58] A. Shali and W. R. Cook. Hybrid partial evaluation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 375–390. ACM Press, 2011. doi: 10.1145/2048066.2048098.

[59] 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. doi: 10.1145/1328195.1328197.

[60] J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, 1991.

[61] Smalltalk/X, 2013. URL http://live.exept.de/doc/online/english/programming/java.html.

[62] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose. Lazy continuations for Java virtual machines. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 143–152. ACM Press, 2009. doi: 10.1145/1596655.1596679.

[63] L. Stadler, T. Würthinger, and C. Wimmer. Efficient coroutines for the Java platform. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 20–28. ACM Press, 2010. doi: 10.1145/1852761.1852765.

[64] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Journal on Higher-Order and Symbolic Computation*, 13(3):161–178, 2000. doi: 10.1023/A:1010078412711.

[65] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242. ACM Press, 1987. doi: 10.1145/38765.38828.

[66] Unladen Swallow. unladen-swallow, 2009. URL http://code.google.com/p/unladen-swallow/.

[67] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, 1992. doi: 10.1145/114005.102806.

[68] C. Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object fusing. *ACM Transactions on Architecture and Code Optimization*, 7(2), 2010. doi: 10.1145/1839667.1839669.

[69] 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. doi: 10.1145/2400682.2400689.

[70] M. Wolczko, O. Agesen, and D. Ungar. Towards a universal implementation substrate for object-oriented languages. In *Proceedings of the Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, 1999.

[71] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming*, 74(5-6), 2009. doi: 10.1016/j.scico.2009.01.002.

[72] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the Dynamic Languages Symposium*, pages 73–82. ACM Press, 2012. doi: 10.1145/2384577.2384587.