Practical Partial Evaluation for High-Performance Dynamic Language Runtimes

Thomas Würthinger[*] Christian Wimmer[*] Christian Humer[*] Andreas Wöß[*]
Lukas Stadler[*] Chris Seaton[*] Gilles Duboscq[*] Doug Simon[*] Matthias Grimmer[†]

[*]Oracle Labs [†]Institute for System Software, Johannes Kepler University Linz, Austria
{thomas.wuerthinger, christian.wimmer, christian.humer, andreas.woess, lukas.stadler, chris.seaton, gilles.m.duboscq, doug.simon}@oracle.com matthias.grimmer@jku.at

Abstract

Most high-performance dynamic language virtual machines duplicate language semantics in the interpreter, compiler, and runtime system. This violates the principle to not repeat yourself. In contrast, we define languages solely by writing an interpreter. The interpreter performs specializations, e.g., augments the interpreted program with type information and profiling information. Compiled code is derived automatically using partial evaluation while incorporating these specializations. This makes partial evaluation practical in the context of dynamic languages: It reduces the size of the compiled code while still compiling all parts of an operation that are relevant for a particular program. When a speculation fails, execution transfers back to the interpreter, the program re-specializes in the interpreter, and later partial evaluation again transforms the new state of the interpreter to compiled code. We evaluate our approach by comparing our implementations of JavaScript, Ruby, and R with best-in-class specialized production implementations. Our general-purpose compilation system is competitive with production systems even when they have been heavily optimized for the one language they support. For our set of benchmarks, our speedup relative to the V8 JavaScript VM is 0.83x, relative to JRuby is 3.8x, and relative to GNU R is 5x.

CCS Concepts **• Software and its engineering → Runtime environments**

Keywords dynamic languages; virtual machine; language implementation; optimization; partial evaluation

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062381

1. Introduction

High-performance virtual machines (VMs) such as the Java HotSpot VM or the V8 JavaScript VM follow the design that was first implemented for the SELF language [25]: a multi-tier optimization system with adaptive optimization and deoptimization. The first execution tier, usually an interpreter or fast-compiling baseline compiler, enables fast startup. The second execution tier, a dynamic compiler generating optimized machine code for frequently executed code, provides good peak performance. Deoptimization transitions execution from the second tier back to the first tier, i.e., replaces stack frames of optimized code with frames of unoptimized code when an assumption made by the dynamic compiler no longer holds (see Section 5.2 for details).

Multiple tiers increase the implementation and maintenance costs for a VM: In addition to a language-specific optimizing compiler, a separate first-tier execution system must be implemented [2, 4, 23]. Even though the complexity of an interpreter or a baseline compiler is lower than the complexity of an optimizing compiler, implementing them is far from trivial [48]. Additionally, they need to be maintained and ported to new architectures. But more importantly, the semantics of the language need to be implemented multiple times in different styles: For the first tier, language operations are usually implemented as templates of assembler code. For the second tier, language operations are implemented as graphs of language-specific compiler intermediate representation. And for the runtime system (code called from the interpreter or compiled code using runtime calls), language operations are implemented in C or C++.

We implement the language semantics only once in a simple form: as a language interpreter written in a managed high-level host language. Optimized compiled code is derived from the interpreter using partial evaluation. This approach and its obvious benefits were described in 1971 by Y. Futamura [15], and is known as the *first Futamura projection*. To the best of our knowledge no prior high-performance language implementation used this approach.

We believe that a simple partial evaluation of a dynamic language interpreter cannot lead to high-performance compiled code: if the complete semantics for a language operation are included during partial evaluation, the size of the compiled code explodes; if language operations are not included during partial evaluation and remain runtime calls, performance is mediocre. To overcome these inherent problems, we write the interpreter in a style that anticipates and embraces partial evaluation. The interpreter specializes the executed instructions, e.g., collects type information and profiling information. The compiler speculates that the interpreter state is stable and creates highly optimized and compact machine code. If a speculation turns out to be wrong, i.e., was too optimistic, execution transfers back to the interpreter. The interpreter updates the information, so that the next partial evaluation is less speculative.

This paper shows that a few core primitives are sufficient to communicate information from the interpreter to the partial evaluator. We present our set of primitives that allow us to implement a wide variety of dynamic languages. Note that we do not claim that our primitives are necessary, i.e., the only way to make partial evaluation of dynamic language interpreters work in practice.

Closely related to our approach is PyPy [6, 8, 39], which also requires the language implementer to only write an interpreter to express language semantics. However, PyPy does not use partial evaluation to derive compiled code from the interpreter, but meta-tracing: a trace-based compiler observes the running interpreter (see Section 9 for details).

In summary, this paper contributes the following:

- We present core primitives that make partial evaluation of dynamic language interpreters work in practice. They allow a language-agnostic dynamic compiler to generate high-performance code.
- We present details of the partial evaluation algorithm and show which compiler optimizations are essential after partial evaluation.
- We show that our approach works in practice by comparing our language implementations with the best production implementations of JavaScript, Ruby, and R.

2. System Structure

We implement many different *guest language*s in a managed *host language*. Only the interpreter and the runtime system for a guest language is implemented anew by the language developer. Our framework and the host language provide a core set of reusable host services that do not need to be implemented by the language developer, such as dynamic compilation (the focus of this paper), automatic memory management, threads, synchronization primitives, and a well-defined memory model.

For the concrete examples and the evaluation of this paper, we implemented multiple guest languages (JavaScript, Ruby, and R) in the host language, Java. We require only one implementation for every guest language operation. Since the interpreter and the runtime system are written in the same managed host language, there is no strict separation between them. Regular method calls are used to call the runtime from the interpreter.

Optimized compiled code is inferred from the interpreter using partial evaluation. *Partial evaluation* (PE) is the process of creating the initial high-level compiler intermediate representation (IR) for a guest language function[1] from the guest language interpreter methods (code) and the interpreted program (data). The interpreted program consists of all the data structures used by the interpreter, e.g., interpreter instructions organized as an abstract syntax tree. The language implementer does not have to provide any language specific input to the language-independent optimizing compiler. Section 5.1 provides details of our PE process.

In dynamic languages, the full semantics of seemingly simple operations are complex. For example, addition in JavaScript can be as simple as the addition of two numbers. But it can also be string concatenation as well as involve the invocation of complex conversion functions that convert arbitrary object inputs to numbers or strings. Addition in Ruby can again be just the addition of two numbers, but also the invocation of an arbitrary addition method that can be redefined at any time. Addition in R can be just the addition of two vectors of numbers, but also involve a complex method lookup based on the classes of its arguments. We claim that a simple PE that always captures the full semantics of every operation in compiled code is infeasible: it either leads to code explosion if all code for the operation is compiled, or mediocre performance when only parts of the operation are compiled and runtime calls remain in the compiled code for the full semantics.

We solve this problem by making the interpreter aware of PE using core primitives. The remainder of this section is a high-level description of the most important primitives, with forward references to Section 3, which defines all core primitives in detail.

Figure 1 illustrates the interactions of interpreter code, compiled code, and runtime code. The interpreter methods a to d call each other, and call the runtime methods e to i. Two guest language functions `x` and `y` are compiled. The *core primitive to initiate PE* (see Section 3.1) starts PE at the same interpreter method a, but then processes different interpreter methods that are in different specialization states. We denote the different specialization states using subscripts, e.g., b_{x1} denotes one specialization state of b for function x. The compiled code does not contain calls to any interpreter methods anymore. PE stops at calls to the runtime, i.e., calls to methods that are annotated with the *core primitive* `PEBoundary` (see Section 3.4).

[1] To distinguish the languages in this paper, *function* always refers to the guest language while *method* always refers to the host language

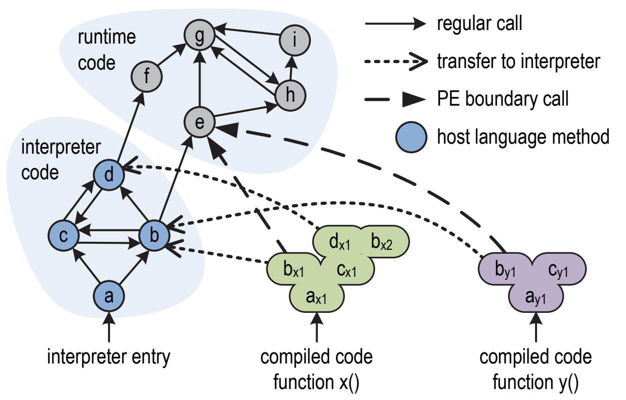


Figure 1: Interactions of interpreter code, compiled code, and runtime code.

While interpreting an operation, the interpreter specializes the instructions, e.g., collects type information or profiling information. In other words, for each instruction the interpreter stores which subset of the semantics is actually used. PE uses the specialization information. For this, the instructions are assumed to be in a *stable* state where subsequent changes are unlikely, although not prohibited. PE eliminates the interpreter dispatch code access of stable state: PE eliminates the load of a field annotated with the *core primitive* `PEFinal` by replacing the field load with a constant for the read (see Section 3.2).

The parts of the interpreter code responsible for specialization and profiling are omitted from compilation: The *core primitive* `transferToInterpreter` (see Section 3.2) is a function that causes a transfer from optimized machine code back to the interpreter. This results in machine code that is aggressively specialized for the types and values encountered during interpretation. The transfer back to the interpreter discards the optimized machine code. During interpretation, the interpreter updates the specialization, e.g., incorporates revised type information. The function can be compiled again using PE, with the new specializations considered stable by the compiler. In the example in Figure 1, methods like b_{x1}, b_{y1}, and d_{x1} are specialized in the compiled code, i.e., the compiled code omits some parts of methods b and d. Execution transfers to the interpreter in case these parts are executed at run time. PE is an expensive operation, therefore waiting for a stable state before initiating PE is important to avoid repeated PE of the same function.

Optimized compiled code can also be specialized on global information maintained by the runtime. When such information changes, the compiled code is discarded using the *core primitive* `Assumption` (see Section 3.3). Therefore, no code needs to be emitted by the compiler to check the validity of an assumption.

Our *transfer to interpreter* is implemented in the VM via *deoptimization* [25]. Deoptimization replaces the stack frames of optimized code with interpreter stack frames (see Section 5.2). It is provided by our framework to the guest language implementer as an *intrinsic method*, i.e., a method of the host language that is handled specially by the compiler.

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, written in the host language. The guest language developer gets a high-performance language implementation, but does not need to be a compiler expert.

The language implementer is responsible for *completeness* and *finiteness* of specializations. In a particular specialization state, an operation may handle only a subset of the semantics of a guest language operation. However, it must provide re-specialization for the complete semantics of the operation. After a finite number of re-specializations, the operation must end up in a state that handles the full semantics without further updates. In other words, there must be a state that can handle all possible inputs. Otherwise, the process of deoptimization, re-specialization, and compilation would go on forever. Identifying and fixing repeated deoptimization issues is challenging. To minimize the possibility of errors, we provide a higher-level API that uses the core primitives and is guaranteed to stabilize, for example, a domain-specific language to express specializations [26].

3. Core Primitives for the Interpreter

We believe that an interpreter needs to be aware of PE so that the compiler can create concise and efficient machine code. We identified a small number of core primitives that are sufficient to make the interpreter aware of PE.

3.1 Initiate Partial Evaluation

PE combines a method (code) with data. Figure 2 illustrates how PE is initiated for a guest language function. The interpreter data structures, e.g., the interpreted instructions, are reachable from fields of a `Function` object. Calling the instance method `execute` interprets the function. The input to PE is a reference to this function (code), and a concrete `Function` object (data). See Section 5.1 for details on how the PE algorithm traverses the method call graph and optimizes access of the data objects. The result of PE is a handle to optimized machine code for the guest language function. It can be invoked like a static method that takes the same arguments (the `args` array) as the `execute` method. Note that Figure 2 is simplified and not valid Java code because Java requires the use of reflection to search for a method.

A compilation policy decides which guest language functions are compiled, based on execution count and other properties of a function. Since most dynamic languages work well with the same compilation policy, a default implementation is provided by our framework to language implementers, i.e., language implementers do not need to initiate PE themselves.

```
Object interpret(Function function, Object[] args) {
  return function.execute(args);
}

Object compiled(Function function, Object[] args) {
  MethodHandle code = partialEvaluation(
                   Function::execute, function);
  return code.invoke(args);
}
```

Figure 2: Initiate partial evaluation.

3.2 Method-Internal Invalidation of Compiled Code

The interpreter collects information while interpreting a function, stores this information in the instructions (i.e., instructions are a mutable data structure), and makes the information available to the compiler. Compiled code speculates that the information is stable, i.e., speculates that the information is not going to change in future executions. The compiled code contains a check that the information is still correct, but no code to handle incorrect information. Instead, execution continues in (i.e., deoptimizes to) the interpreter. The interpreter then updates the information and stores the new value in the instructions. When the function is compiled again, the new information leads to code which is less speculative.

The annotation `PEFinal` on a field instructs PE that the field is stable. PE treats the field like a `final` field, i.e., reads the value, replaces the field load with a constant for the read value, and performs constant folding. The interpreter is allowed to write the field, but compiled code must not write to it. Every write must be preceded by a call to the intrinsic method `transferToInterpreter`. This intrinsic transfers execution from optimized compiled code to the interpreter and invalidates the compiled code. Execution does not continue in compiled code after the call to `transferToInterpreter`, so dead code elimination removes all code dominated by the method call.

Since compiled code is not allowed to write a `PEFinal` field, PE could treat a write implicitly as a transfer to the interpreter. We decided to use an explicit intrinsic method because we believe it makes the interpreter code more readable. In addition, code that computes the new value of the `PEFinal` field can also be excluded from PE, i.e., the call to `transferToInterpreter` can be well before the write to a `PEFinal` field.

Figure 3 shows an example for method-internal invalidation of compiled code. We want to optimize the negation operation of a JavaScript-like dynamic language. Any value can be negated, but usually only numbers are. To create concise and fast code, the compiler needs to omit the code for the unlikely path. This reduces the code size, but more importantly improves type information for subsequent operations: since the result of the negation is always a number, the compiler can remove type checks on the result value.

```
@PEFinal boolean objectSeen = false;

Object negate(Object v) {
  if (v instanceof Double) {
    return -((double) v);
  } else {
    if (!objectSeen) {
      transferToInterpreter();
      objectSeen = true;
    }
    // slow-case handling of all other types
    return objectNegate(v);
  }
}
```

Figure 3: Method-internal invalidation of compiled code.

```
if (v instanceof Double)
  return -((double) v);
else
  deoptimize;
```

(a) `objectSeen` is `false`.

```
if (v instanceof Double)
  return -((double) v);
else
  return objectNegate(v);
```

(b) `objectSeen` is `true`.

Figure 4: Compiled parts of the example in Figure 3.

In the common case, the interpreter only encounters numbers, i.e., the value of the `PEFinal` field `objectSeen` is `false` during PE. PE replaces the field load with the constant `false`. Constant folding and dead code elimination remove the call to `objectNegate`. The boxing and unboxing operations between primitive `double` values and boxed `Double` objects can be removed by the compiler because the exact type of boxed values is known. Figure 4a shows the code that gets compiled when `objectSeen` is `false`. The call to `transferToInterpreter` is intrinsified to a `deoptimize` runtime call that rewrites the stack and therefore does not return to the compiled code.

If `negate` is called with a non-number argument, the machine code cannot compute the result. Instead, the `transferToInterpreter` invalidates the machine code, and execution continues in the interpreter. The interpreter updates `objectSeen` to `true` and performs the correct negation operation. When `negate` is compiled again, the call to `objectNegate` remains reachable. Figure 4b shows the code that gets compiled when `objectSeen` is `true`.

In summary, the `PEFinal` field `objectSeen` is always constant folded, i.e., there is no read of the field in compiled code. The write is after a call to `transferToInterpreter`, so there is no write of the field in compiled code. Note that `objectSeen` is an instance field. Every negation operation in an application has a separate instruction object for the negation, so each negation is specialized independently.

3.3 Method-External Invalidation of Compiled Code

The runtime system of a language collects global information about loaded guest language code, stores this information in global data structures, and makes the information

```
@PEFinal Assumption notRedefined = new Assumption();

int add(int left, int right) {
  if (notRedefined.isValid()) {
    return left + right;
  }
  ... // complicated code to call user-defined add
}

void redefineFunction(String name, ...) {
  if (name.equals("+")) {
    notRedefined.invalidate());
    ... // register user-defined add
  }
}
```

Figure 5: Method-external invalidation of compiled code.

available to the compiler. The compiler speculates that the information is stable, i.e., optimizes the function as if the information is not going to change in future executions. In contrast to the core primitive of the previous section, the compiled code does not contain a check that the information is still correct. Since the information in the global data structure can be changed at any time, there is no single point within a function where a check could be placed. Performing a check repeatedly would lead to slow code. Instead, the compiled code can be invalidated externally. When the global information is changed, all compiled code that depends on this information is invalidated. All stack frames of all affected compiled functions are deoptimized.

Our implementation uses a class `Assumption` with an intrinsic method `isValid`. Every assumption has a boolean field that stores whether the assumption is still valid. In the interpreter, `isValid` returns this field so that the interpreter can react to invalidated assumptions. The compiler intrinsifies `isValid` to a no-op, i.e., no code is emitted for it. Instead, the compiler adds the compiled code to a dependency list stored in the assumption object. If the assumption is invalidated using `invalidate`, all dependent compiled code is invalidated.

Figure 5 shows an example for method-external invalidation of compiled code. We want to optimize addition of a Ruby-like dynamic language. Arithmetic operations on all types can be changed by the application at any time, but usually applications do not change such basic operations. But to guarantee correct language semantics, compiled code must be prepared for a change to happen at any time, including while evaluating the arguments for the addition. It is therefore not enough to check for redefinition once at the beginning of the compiled code. Emitting machine code for the check before every arithmetic operation would lead to an unacceptable slowdown. Therefore, we define an `Assumption` that the addition has not been redefined. If the user has not redefined addition, the compiler replaces the intrinsic method `isValid` with the constant `true`. Constant folding and dead code elimination simplify the `if` statement, so only the simple and straight-line code `left + right` is compiled. No compiled code is emitted to check the validity of the assumption. Instead, the compiler registers the optimized code as depending on the assumption.

The runtime code that loads and registers new functions checks if a newly defined function redefines the default addition operation. In this case, the runtime invalidates the assumption. All compiled code that contains an addition is invalidated, and execution continues in the interpreter. The next time an addition is compiled, the intrinsic method `isValid` is replaced with the constant `false`. Only the call to the user-defined new addition operation is emitted by the compiler.

```
Object parseJSON(Object value) {
  String s = objectToString(value);
  return parseJSONString(s);
}

@PEBoundary
Object parseJSONString(String value) {
  ... // JSON parsing code
}
```

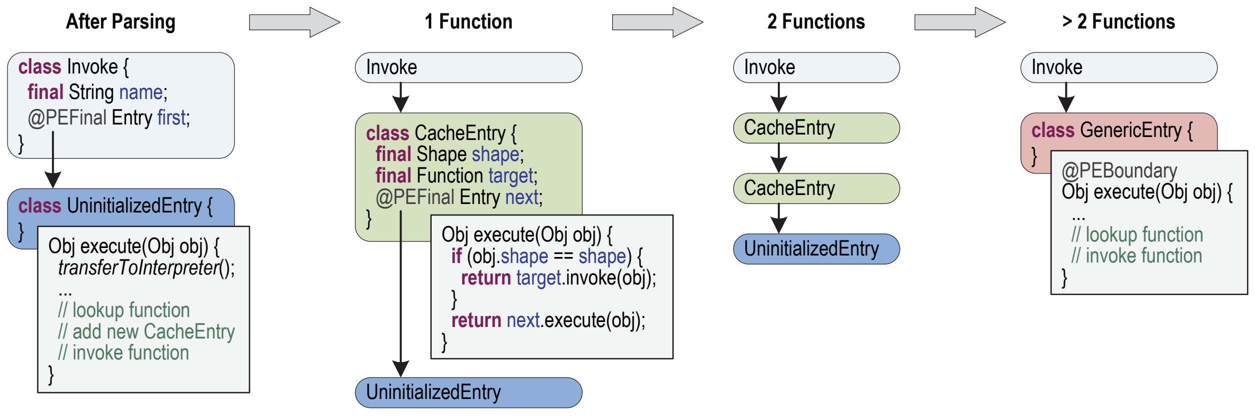
Figure 6: Explicit boundary for partial evaluation.

3.4 Explicit Boundary for Partial Evaluation

PE traverses the call graph of the interpreter and includes all interpreter methods reachable for a particular guest language function. At some point traversal must stop, otherwise unimportant slow-path code would be compiled and the size of the compiled code would explode. The stopping point separates the interpreter from the language runtime: everything that is part of the interpreter is processed by PE, while calls to the language runtime remain as calls. Remember that in our system the interpreter and the language runtime are implemented in the same host language. The language implementer can freely decide where the boundary is, and move the boundary easily by marking a method explicitly as the boundary. We use the annotation `PEBoundary` to explicitly mark boundaries.

Figure 6 shows an example for the boundary annotation. A JavaScript-like language has a built-in function for parsing JSON into an object graph. The function accepts any value as an argument, but converts the argument to a string first before invoking the JSON parser on the string. In our example, the PE boundary is placed after the to-string conversion, but before the actual JSON parsing. This allows speculation and type propagation for the to-string conversion: it is likely that the compiler can infer types and optimize the conversion. But the actual JSON parsing code remains behind a call.

It might be possible to find out boundaries automatically, i.e., eliminate the need for the annotation. We experimented with different heuristics and algorithms to detect boundaries, but none of them resulted in intuitive and predictable results. We therefore removed all heuristics again and reverted to explicit boundaries. In our example, the language imple-



```
class Invoke {
  final String name;
  @PEFinal Entry first;
}
```

```
class UninitializedEntry {
}
Obj execute(Obj obj) {
  transferToInterpreter();
  ...
  // lookup function
  // add new CacheEntry
  // invoke function
}
```

```
class CacheEntry {
  final Shape shape;
  final Function target;
  @PEFinal Entry next;
}
Obj execute(Obj obj) {
  if (obj.shape == shape) {
    return target.invoke(obj);
  }
  return next.execute(obj);
}
```

```
class GenericEntry {
}
@PEBoundary
Obj execute(Obj obj) {
  ...
  // lookup function
  // invoke function
}
```

Figure 7: Sketch for a polymorphic inline cache.

menter might find out, by analyzing real usages of JSON parsing, that the to-string conversion cannot be optimized by the compiler. Then the PE boundary can be moved to the first method of the example. Or the language implementer might find out that the actual JSON parsing code itself can benefit from type information and speculation. Then the `PEBoundary` can be removed completely or moved inside of the JSON parsing code. The explicit annotation gives the language implementer fine-grained control to incorporate knowledge how the language is used, i.e., knowledge that is not present in the language implementation itself.

3.5 Distinguishing Interpreted Execution

Profiling code in the interpreter must be excluded during compilation. The intrinsic method `inInterpreter` always returns `true` in the interpreter, but is intrinsified to `false`. Constant folding and dead code elimination remove all code that is guarded using `inInterpreter`. Figure 8 in a later section shows an example.

3.6 Modifying the Interpreter State of Caller Frames

Many dynamic languages have built-in functions that reflect on the call stack, e.g., print the stack trace in case of exceptions or change local variables in activation frames down the stack. The few frames that are written externally must be allocated on the heap instead of on the stack, and the compiled code of these functions cannot speculate on local variable types since local variables are changed externally. However, the language runtime does not know ahead of time which frames will be written because arbitrary optimized frames can be the subjects of inspection.

For example, Ruby's regular expression match function stores the result in a local variable named `$~` in the caller frame. This occurs in practice, e.g., when running the Discourse web forum application in its benchmark configuration each request involves around 14k reads from and 11k writes to a caller frame. To support this behavior, other implementations of Ruby either store all function activation frames on the heap so that they can always be accessed (MRI and Rubinius), or try to analyze before compilation if a frame will be written to (JRuby) and then use heap-based frames only for such methods. In the case of JRuby this static analysis is demonstrably unsound because functions can be aliased and local variables accessed using meta-programming, defeating the static analysis.

We provide the language implementer with an interpreter-level view of all stack frames, with full support to read and write local variables; but without any overhead for frames that are not written to. The language implementer always sees the interpreter-level state, i.e., a heap-based frame object. By default, compiled code uses stack-based frames. The same information that allows transferring to the interpreter also allows inspecting optimized frames. If only read access is necessary, the heap-based frame is temporarily restored in a side data structure and read operations are performed as if it was the real frame. This allows reflecting on the executed function and reads without impacting the performance: when control returns to the optimized code, the compiled frame is still on the stack.

Write access to an optimized frame requires a transfer to the interpreter for that particular activation. When control returns to that function, execution continues in the interpreter using the heap-based frame that was written to. Profiling information stores that the frame was written to from outside, so that the next version of compiled code uses a heap-based frame that can be written to externally.

4. Example Usages of the Core Primitives

This section uses the core primitives to implement higher-level concepts used by many dynamic languages.

4.1 Polymorphic Inline Caches

Polymorphic inline caches optimize function and property lookup in dynamic languages and are usually implemented using assembler code and code patching [24]. Our system supports them by chaining host-language objects representing cache entries. For every new entry in the inline cache, a new entry is added to the list. The cache entry 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 entry in the chain. When the chain reaches a certain predefined length, i.e., the desired maximum cache size, the whole chain is replaced with one entry responsible for handling the fully megamorphic case.

Figure 7 sketches a polymorphic inline cache for guest language function invocation. The `Invoke` interpreter instruction stores the name of the invoked function and the head of the entry list. Before the first execution, the only entry in the list is the `UninitializedEntry`. Upon execution, it adds a new cache entry to the list, but remains at the end of the list. The linked list uses `PEFinal` fields: the interpreter can change them, but PE sees them as constant. This removes the dispatch overhead between cache entries. Each `CacheEntry` caches the invoked function for a receiver object type (usually called *shape* or *hidden class*). Checking for a cache hit can be compiled to one memory load and one comparison because `final` and `PEFinal` fields are constant folded by the compiler.

When the cache exceeds the maximum desired size, the whole list is replaced with a single `GenericEntry` that performs a slow-path lookup of the function. This is usually an expensive and complicated lookup, so it is a runtime call behind a `PEBoundary`. It does not invalidate optimized code. In contrast, the `UninitializedEntry` uses `transferToInterpreter` to invalidate optimized code upon execution. This ensures that the newly added cache entries are included in the next compilation.

4.2 Typed Local Variables

Reading and writing local variables is performed by guest languages via an index into a `Frame` object that contains an array holding the values. Local variable access instructions specialize on the type of a local variable. This allows for dynamic profiling of variable types 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 PE is fast. Escape analysis (see Section 5.3) eliminates every access to the array and instead connects the read of the variable with the last write. Guest-language local variables have no performance disadvantage compared to host language local variables. The host compiler can perform standard compiler optimizations such as constant folding or global value numbering for guest language 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.

4.3 Compilation of Loops

For long running loops it is beneficial to switch from the interpreter to compiled code during the execution of the loop. This kind of behavior is often referred to as on-stack-replacement (OSR) for loops and usually requires non-trivial changes to the optimizing compiler and the runtime system [23]. Since PE can start at any place in the interpreter, we implement compilation of loops with a few lines of code in the interpreter.

```
class DoWhileLoop {
  MethodHandle code = null;

  void executeLoop() {
    int loopCount = 0;
    do {
      if (inInterpreter()) {
        loopCount++;
        if (code == null && loopCount > THRESHOLD) {
          code = partialEvaluation(
              DoWhileLoop::executeLoop, this);
        }
        if (code != null) {
          code.invoke();
          return;
        }
      }

      body.execute();
    while (condition.execute());
  }
}
```

Figure 8: Compilation of frequently executed loops.

Figure 8 shows how to invoke PE for a `do-while` loop. The `do-while` loop instruction is implemented in the interpreter using the `do-while` loop of the host language. The loop `body` is executed as long as evaluating the loop `condition` returns `true`, but at least once. If the loop execution count exceeds an arbitrary but fixed threshold, PE is initiated for the `executeLoop` method that is currently running in the interpreter, i.e., with the current `this` pointer as the data for PE. The intrinsic method `inInterpreter` is intrinsified to `false` during PE, so the counting code is excluded from compilation. The resulting compiled code is invoked immediately and executes the remaining loop iterations. Note that the interpreter frame remains on the stack since the interpreter invokes the compiled code, which is different to other OSR implementations.

5. Implementation Details

5.1 Partial Evaluation of the Interpreter

PE builds the initial high-level compiler IR that is then the input to the compiler. The input of PE is code and data. Code denotes the interpreter methods written in the host language, which are the same for all guest language functions. Data denotes the interpreter data structures for the guest language function that is compiled: the actual interpreted instructions together with profiling information collected by the interpreter.

Our PE algorithm follows the standard approach for *online PE* [27]. It starts with the first instruction of the interpreter method passed to PE. This method has two kinds of arguments: 1) the object that references all the interpreter data structures, and 2) the actual user arguments. For PE of a function, the first kind of argument is a constant `Function` object for that particular function. Based on this initial constant, virtual method calls performed by the interpreter can be de-virtualized, i.e., converted from indirect calls to direct calls with a known exact target method.

PE follows all direct calls: instead of emitting a call instruction, the called method is processed recursively. Memory loads are constant folded immediately during PE: if the receiver of a memory load is a constant object and the accessed field or memory element is considered as immutable, the read is performed during PE and the result of the read is another constant. Constant folding replaces arithmetic and comparison operations with constants when all inputs of the operation are constant. Dead code elimination replaces `if`-statements whose condition is a constant with a branch to the single reachable successor. The dead branch is not visited during PE, i.e., instead of parsing all branches into IR and then later on deleting the branches again, dead branches are not parsed in the first place. This makes PE linear in time and space regarding the size of the built IR, and not linear in the size of interpreter code or potentially reachable code. Constant folding of memory reads is not just performed for always-immutable `final` fields, but also for fields annotated as `PEFinal`.

Intrinsic methods are intrinsified during PE. For example, the method `transferToInterpreter` is intrinsified to a compiler IR node that triggers deoptimization. Since control never returns to compiled code after deoptimization, all code dominated by it is dead code. Dead code elimination ensures that all code dominated by the deoptimization instruction is not parsed, i.e., the deoptimization instruction is a control flow sink that has no successors.

PE stops at methods annotated with `PEBoundary`. Calls to such annotated methods always result in a call instruction in the compiler IR, i.e., PE does not recurse into the annotated method. Indirect calls that cannot be de-virtualized during PE also result in call instructions. However, they are often an indicator that PE could not replace the receiver of the call with a constant, so we emit a warning message to the language implementer.

Our experience shows that all code that was not explicitly designed for PE should be behind a PE boundary. We have seen several examples of exploding code size or even nonterminating PE due to infinite processing of recursive methods. For example, even the seemingly simple Java collection method `HashMap.put` is recursive and calls hundreds of different methods that might be processed by PE too.

5.2 VM Support for Deoptimization

Deoptimization is an integral part of our approach. The deoptimization algorithm first presented for SELF [25] and now implemented in many VMs such as the Java HotSpot VM or the V8 JavaScript VM is sufficient for our purposes, i.e., we do not make a novel contribution to deoptimization. We briefly summarize the relevant parts here.

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

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

5.3 Compiler Support

The high-level IR that is created by PE is compiled using a standard compilation pipeline. Our approach makes only a few demands on the compiler:

- The compiler needs to support deoptimization as a first-class primitive. It needs to produce the necessary metadata for deoptimization (see Section 5.2). Additionally, it needs to provide a high-level IR instruction for deoptimization, used by the intrinsification of `transferToInterpreter`.
- The compiler needs to be able to track assumptions as well as communicate to the VM or directly to our `Assumption` objects the final list of assumptions on which the compiled code depends.

The compiler performs all standard compiler optimizations. However, the importance of compiler optimizations for PE is altered compared to a standard compilation. For example, method inlining in the compiler is not important at all and can even be disabled because PE already processes all relevant methods. We identified escape analysis as the most important optimization for the performance of our language implementations.

Escape analysis determines the dynamic scope and the lifetime of allocated objects. If an object does not escape the current compilation unit, *scalar replacement* eliminates the allocation altogether and replaces the fields of the object with local variables. If an object does escape, e.g., is returned from the method, *partial escape analysis* allocates the object just before the escape point and replaces fields with local variables for all code before the escape point [52]. Deoptimization instructions are not escape points. Instead, information about the virtual state of an object is stored in the *scope descriptor* so that the object can be re-allocated in the unlikely case that deoptimization happens [28].

6. Limitations

Our approach significantly reduces the implementation effort for a dynamic language by decoupling the language semantics and the optimization system. However, this comes at the cost of longer warmup times compared to a runtime system specialized for a single language. Our measurements show that warmup times are an order of magnitude longer than in a specialized runtime. This makes our approach unsuited for systems where peak performance must be reached within seconds. However, the evaluation in Section 7.2 shows that our system reaches peak performance in about a minute, which is sufficient for server-side applications. We are currently exploring ways to reduce warmup times by doing ahead-of-time optimization of our interpreter and compiler, so that those components are already warmed up and optimized at startup [59].

Writing language interpreters for our system is simpler than writing specialized optimizing compilers. However, it still requires correct usage of the primitives and a PE mindset. It is therefore not straightforward to convert an existing standard interpreter into a high-performance implementation using our system. On the other hand, building an interpreter from scratch for our system is simplified by available high-level language-agnostic primitives.

Guest languages often provide language features that are not available in our host language Java, e.g., Ruby's continuations and fibers. However, we were able to implement all language features of JavaScript, Ruby, and R. Continuations and fibers are implemented using threads. This implementation has the correct behavior but is much less efficient than it could be. An efficient implementation would require adding continuations [50] or coroutines [51] to our host language.

Note that tail calls are also not provided by Java, but can be correctly implemented in our system: In the interpreter, stack frames of the caller are removed using exceptions. The compiler detects that pattern and emits a tail call without the Java runtime system noticing.

7. Evaluation

We evaluate our approach with the implementation of three dynamic languages: JavaScript, Ruby, and R. JavaScript allows us to compare our approach with highly tuned virtual machines that were designed for only one language, i.e., have a language-specific dynamic compiler. Ruby offers sophisticated meta-programming facilities that are difficult to optimize [34]. R is a language for statistical computations that has been shown to be difficult to optimize because it is highly dynamic and reflective [33]. Because of the differences between the languages, they have been implemented by independent teams that together shaped the core primitives of our approach.

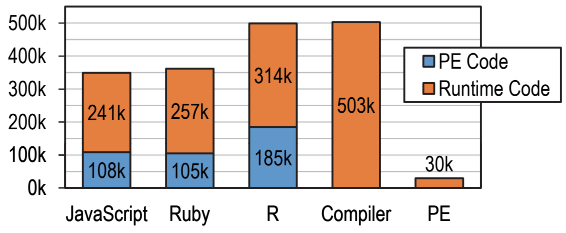


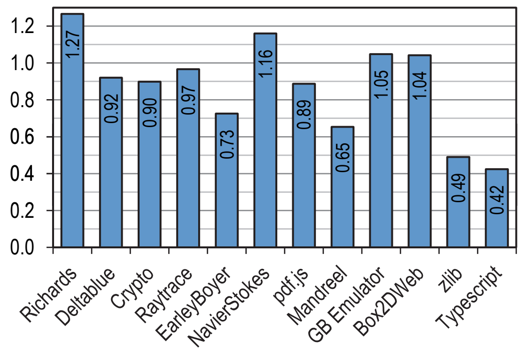
Figure 9: Code size (size of bytecode) of language implementations and compiler.

Our implementations support large parts of the standard libraries and standard test suites: Our JavaScript implementation passes 93% of the ECMAScript2016 standard test suite [54]. Our Ruby implementation passes 98% (language) and 94% (core library) of the *Ruby spec* standard test suite. No standard specification is available for R, but our implementation is tested using thousands of tests for compatibility with the de-facto standard GNU R implementation.

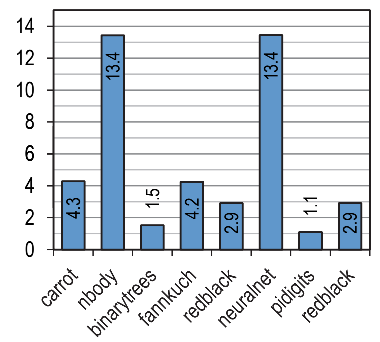
All measurements were performed on a dual-socket Intel Xeon E5-2699 v3 with 18 physical cores (36 virtual cores) per socket running at 2.30 GHz, 384 GByte main memory, running Red Hat Enterprise Linux Server release 6.5 (kernel version 2.6.32-431.29.2.el6.x86_64). All our language implementations are available online as part of the GraalVM download [36]. The source code of the language implementation framework (Truffle), the language-independent compiler (Graal), the Ruby implementation (TruffleRuby), and the R implementation (FastR) are available as open source [17].

7.1 Size of the Language Implementations

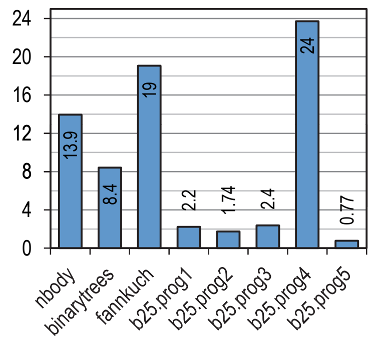
Figure 9 shows the approximate Java bytecode size of our compiler and language implementations. For each language implementation, we distinguish code that can be part of a PE (interpreter code) and code that is always behind a PE boundary (runtime code). As noted in previous sections, the boundary between interpreter and runtime is flexible and can be changed easily by moving annotations. Code that is dominated by a `transferToInterpreter`, i.e., code that changes specializations or collects profiling information, is also considered part of the runtime system because it is not reachable during PE. A large part of the language implementations (around 30%) can be part of PE. This substantial



(a) JavaScript relative to V8



(b) Ruby relative to JRuby.



(c) R relative to GNU R.

Figure 10: Speedup of our system relative to best production VM (higher is better).

amount of code would be implemented multiple times in a traditional VM.

The Graal compiler [13, 14, 49] is independent of our PE approach. Graal compiles Java bytecode to optimized machine code, and can serve as a replacement for the Java HotSpot server compiler [37]. We extended it with a front-end that performs PE, and added intrinsic methods that support our core primitives. These PE parts are only a fraction of the system: the compiler is about 500k bytecode; PE is only 30k bytecode. This suggests that the approach is transferable with modest effort to other optimizing compilers.

7.2 Peak Performance

We measure the peak performance of our system for each language. The baseline is the best performing language implementation system for that language: the V8 JavaScript VM (version 5.3.332.6), JRuby (version 9.1.6.0 revision `aba18f3` with `invokedynamic` support enabled, running on a Java HotSpot VM 1.8.0_102), and GNU R (version 3.3.2). We take the arithmetic mean of the execution time for 10 runs and compute the peak throughput obtained after warmup. The warmup time is between 30 seconds and 90 seconds per benchmark, depending on language and benchmark. After warmup, every benchmark has reached a steady state such that subsequent iterations are identically and independently distributed.

Figure 10a shows the speedup of our JavaScript implementation relative to the V8 JavaScript VM. We use peak performance benchmarks from the Octane suite, which is the benchmark suite published by the V8 authors. We exclude code loading and latency related benchmarks (where we are significantly worse as explained in Section 6) and GC throughput related benchmarks (where we are better because of the parallel, server-class GC algorithms provided by the Java HotSpot VM). While on average we are slightly slower than the highly optimized V8 VM, we are in the same range and can even outperform V8 on some benchmarks. It shows that our set of core primitives is sufficient for achieving the performance of a highly optimized specialized language implementation.

Figure 10b shows the speedup of our Ruby implementation relative to JRuby. We use the *optcarrot* benchmark [35] (recently chosen for measuring improvements to the main Ruby implementation, i.e., representing the patterns they believe to be important) as well as some benchmarks from the computer language benchmarks game [18]. JRuby runs hosted on a Java VM and generates Java bytecode for Ruby methods. The bytecode is dynamically compiled by the Java HotSpot server compiler like any other bytecode originating from Java code. Like our approach, JRuby also leverages an existing complex runtime system and garbage collector. The performance is, however, significantly slower because of the semantic mismatch between Java bytecode and Ruby language semantics. JRuby cannot solve this semantic mismatch because standard Java bytecode does not provide a mechanism like our core primitives to communicate information to the dynamic compiler.

Figure 10c shows the speedup of our R implementation relative to the GNU R bytecode interpreter. We use a set of benchmarks from the computer language benchmarks game [18] and the *programmation* subsection of the R benchmarks 2.5 [56]. They include diverse code patterns like scalar, vector and matrix operations, arithmetics, statistical functions, loops and recursion. For the set of benchmarks, the performance differences vary greatly. For some R benchmarks that perform operations on small operands, we obtain large speedups due to the reduction in interpreter overhead. For others that perform large vector operations, we obtain smaller performance gains.

Overall, all our language implementations are competitive with the best specialized runtimes, which have been optimized for more than a decade for each respective language.

7.3 Impact of Optimizations

Figure 11 evaluates the impact of several core primitives and the importance of several compiler optimizations. We dis-

	JavaScript		Ruby		R	
	mean	max	mean	max	mean	max
no partial evaluation	104x	320x	156x	1307x	15x	514x
no assumptions	3.6x	18.5x	2.3x	4.7x	1.2x	2.2x
no escape analysis	7.9x	33.4x	36.9x	194.2x	4.4x	16.4x

Figure 11: Slowdown when disabling optimizations, relative to our system with all optimizations (lower is better).

able optimizations selectively and report the slowdown. For each language, we show the geometric mean of the slowdown for all benchmarks presented in the previous section, and the maximum slowdown (the slowdown of the benchmark that performs worst when disabling that optimization).

The configuration *no partial evaluation* disables PE, i.e., disables dynamic compilation of guest language functions. Execution runs in the interpreter only. Performing no PE at all leads to more than 100x slowdown for JavaScript and Ruby, but can vary widely for R workloads given that the interpreter overhead depends on the operand size and many R programs operate on large vectors. Our interpreters are written in a high-level style in the managed host language and do not use fine-tuned assembler code templates, so the performance difference between interpreter and optimized code is higher than in traditional VMs.

Most core primitives presented in this paper cannot be disabled. For example, when disabling the handling of `PEFinal` fields, PE does not traverse much of the interpreter because such fields are pervasive in all interpreter data structure, i.e., disabling `PEFinal` fields is equivalent to disabling PE completely. The only core primitive that can be disabled independently is the handling of assumptions. The configuration *no assumptions* shows the performance impact when the method `isValid` for assumptions is not intrinsified, i.e., when an explicit check for validity is emitted in compiled code. This leads to a slowdown of about 2-3x. One caveat here is that language implementers know that assumptions have zero overhead, so there might be a bias towards using more assumptions than necessary. It might be possible to coalesce fine grained assumptions into coarse grained assumptions and remove redundant assumption checks.

Escape analysis is an essential compiler optimization. The interpreter uses objects pervasively to pass data between interpreter methods, e.g., to store local variables. The configuration *no escape analysis* leads to an order of magnitude slowdown. The slowdown is higher for Ruby because Ruby implements even basic arithmetic operations as method calls, which leads to more temporary objects used in the interpreter at function call boundaries. R code that spends time in long running computation loops does not depend that heavily on escape analysis. In contrast, escape analysis for Java has been reported to yield a speedup of about 2% for the Java DaCapo benchmarks, and about 10% for the Scala DaCapo benchmarks [52]. This underscores

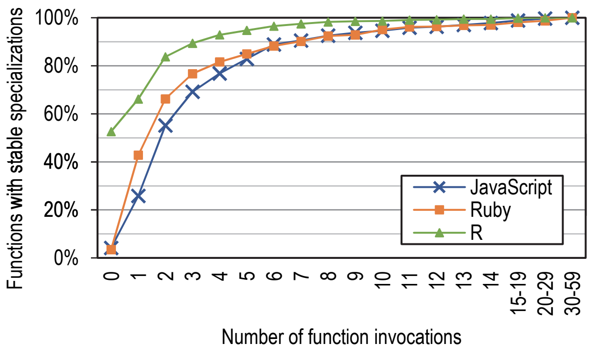


Figure 12: Percentage of functions with stable specializations after a certain number of function invocations.

that escape analysis is optional for compilation of normal Java bytecode, but mandatory for our approach.

7.4 Stabilization of Specializations

We measure how fast interpreter specializations stabilize. A stable state is necessary before PE, otherwise the compiled code would deoptimize immediately. Figure 12 plots the percentage of functions with a stable state against the number of function invocations (the x-axis of the chart). We exclude functions that were run only once. For JavaScript and Ruby, only a few functions are stable before the first invocation, i.e., have no interpreter state that is specialized. R behaves differently because the implementation uses many small dispatch functions that do not need any specialization.

In all languages, specializations stabilize quickly and show the same specialization behavior. 95% of all functions do not change specialization state after 10 invocations. The maximum number of specialization changes is 56, i.e., the last data point in Figure 12 is 100% for all languages. Therefore, 56 is the worst case upper bound for deoptimization and recompilation of any function in our benchmarks. Any reasonable compilation policy initiates compilation of a function only after several executions, so the compiled code of most functions is never deoptimized. This shows that our approach of specialization in the interpreter is practical and does not lead to an excessive amount of deoptimization and recompilation.

8. Research Enabled by Our Approach

This paper focuses on the foundation of our system, i.e., the core principles and primitives of PE. Language developers usually wrap them in higher-level building blocks. Many of them can be shared between languages, for example the implementation of a language-independent object model [12, 60]. The implementation of specializations can be simplified using a domain-specific language [26].

Implementing many languages within the same framework offers other benefits. A language-independent instrumentation API separates language implementations from tools [57]. Specializations ensure that tools have zero overhead on peak performance when they are disabled; deoptimization and recompilation allow tools to be enabled and disabled any time during execution. This includes but is not limited to debuggers [46] and profilers [42].

While this paper focuses on JavaScript, Ruby [45], and R [53], other dynamic languages that this paper did not evaluate such as Python [62] reach a similar peak performance [31]. The approach also provides good peak performance for statically typed low-level programming languages such as C [19, 20] or LLVM bitcode [38]. Languages can be combined and optimized together, enabling multi-lingual applications without a performance overhead [21]. The integration of low-level languages is important because many high-level dynamic languages provide a foreign function interface to C [22].

9. Related Work

We introduced our ideas in a previous paper [61] that did not present an actual implementation and did not contain an evaluation with real-world languages. This paper shows which concepts described earlier are essential core primitives, and which ones are higher-level concepts that can be implemented using the core primitives.

PyPy [6] uses meta-tracing [7] to derive compiled code from an interpreter. The trace-based compiler observes the interpreter executions [8, 39]. Like our core primitives, the interpreter implementer has to provide hints to the compiler [10]. Trace-based compilers only trace some paths, and the language implementer has no control which paths are traced. This unpredictability of tracing leads to an unpredictable peak performance. In contrast, our method-based partial evaluation with explicit boundaries gives language implementers guarantees about what is compiled, and therefore predictable performance.

Partial evaluators can be classified as offline and online [27] and have been used in many projects to specialize programs. Schultz et al. [44] translate Java code to C code that serves as the input to an offline partial evaluator. Masuhara and Yonezawa [32] 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. Shali and Cook [47] implemented an offline-style online partial evaluator in a modified Java compiler. Bolz et al. [9] derive a compiler from a Prolog interpreter using online partial evaluation. Rompf et al. [40] derive a compiler from an interpreter using Lightweight Modular Staging. These approaches perform partial evaluation only once and do not provide a convenient mechanism to transfer back to an interpreter, i.e., they do not support the core primitives we proposed in this paper. We therefore believe that these approaches are not capable of optimizing dynamic languages.

Partial evaluators targeting Java suffer from the fact that Java bytecode cannot fully express all optimizations [43]. The intrinsic methods introduced by our core primitives expose few but sufficient hooks to influence compiled code.

A number of projects have attempted to use LLVM [29] as a compiler for high-level managed languages, such as Rubinius and MacRuby for Ruby [30, 41], Unladen Swallow for Python [55], Shark and VMKit for Java [5, 16], and McVM for MATLAB [11]. 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 interpreter; our system can be thought of as a *High-Level Virtual Machine* (HLVM).

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 Jikes Research VM [3] and the Maxine VM [58] are examples of metacircular Java VMs.

10. Conclusions

We presented a novel approach to implementing dynamic languages. It combines the use of partial evaluation for automatically deriving compiled code from interpreters and the use of deoptimization for speculative optimizations. Our runtime decouples the language semantics from the optimization system. This separation of concerns reuses a language-agnostic optimizing compiler and only requires an implementation of language semantics as an interpreter in a managed host language. The interface between the compiler and interpreter is a small set of core primitives. Language implementers use these primitives to specialize guest language operations and control deoptimization. Partial evaluation, enhanced with these core primitives, enables dynamic language implementations to achieve competitive and sometimes even superior performance compared to runtimes that have language-specific optimizing compilers with heavy investments in language-specific fine tuning. We believe that the reduced complexity for implementing languages in our system will enable more languages to benefit from optimizing compilers. Furthermore, we believe it can also lead to accelerated innovation in programming language implementations.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and contributions. We especially thank Mario Wolczko for feedback on this paper.

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

References

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

[3] 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.

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

[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*, 98(3):408–421, 2015. 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, M. Leuschel, and A. Rigo. Towards just-in-time partial evaluation of Prolog. In *Proceedings of the International Conference on Logic-Based Program Synthesis and Transformation*, pages 158–172. Springer-Verlag, 2010. doi: 10.1007/978-3-642-12592-8_12.

[10] 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.

[11] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the International Conference on Compiler Construction*, pages 46–65. Springer-Verlag, 2010. doi: 10.1007/978-3-642-11970-5_4.

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

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

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

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

[16] 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.

[17] GitHub. Graal multi-language VM, 2016. URL https://github.com/graalvm/.

[18] I. Gouy. The computer language benchmarks game, 2016. URL http://benchmarksgame.alioth.debian.org.

[19] M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic execution of C on a Java virtual machine. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 17–26. ACM Press, 2014. doi: 10.1145/2647508.2647528.

[20] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe execution of C on a Java VM. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*, pages 16–27. ACM Press, 2015. doi: 10.1145/2786558.2786565.

[21] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the Dynamic Languages Symposium*, pages 78–90. ACM Press, 2015. doi: 10.1145/2816707.2816714.

[22] M. Grimmer, C. Seaton, T. Würthinger, and H. Mössenböck. Dynamically composing languages in a modular way: Supporting C extensions for dynamic languages. In *Proceedings of the International Conference on Modularity*, pages 1–13. ACM Press, 2015. doi: 10.1145/2724525.2728790.

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

[24] 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. doi: 10.1007/BFb0057013.

[25] 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.

[26] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 123–132. ACM Press, 2014. doi: 10.1145/2658761.2658776.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

[34] C. Nutter. So you want to optimize Ruby, 2012. URL http://blog.headius.com/2012/10/so-you-want-to-optimize-ruby.html.

[35] Optcarrot. Optcarrot: A NES emulator for Ruby benchmark, 2016. URL https://github.com/mame/optcarrot.

[36] Oracle. Oracle Labs GraalVM downloads, 2016. URL http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/.

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

[38] M. Rigger, M. Grimmer, C. Wimmer, T. Würthinger, and H. Mössenböck. Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, pages 6–15. ACM Press, 2016. doi: 10.1145/2998415.2998416.

[39] 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.

[40] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision JIT compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 41–52. ACM Press, 2014. doi: 10.1145/2594291.2594316.

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

[42] G. Savrun-Yeniçeri, M. L. Van de Vanter, P. Larsen, S. Brunthaler, and M. Franz. An efficient and generic event-based profiler framework for dynamic languages. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 102–112. ACM Press, 2015. doi: 10.1145/2807426.2807435.

[43] 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.

[44] 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.

[45] C. Seaton. AST specialisation and partial evaluation for easy high-performance metaprogramming. In *Workshop on Meta-Programming Techniques and Reflection*, 2016.

[46] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 2:1–2:13. ACM Press, 2014. doi: 10.1145/2617548.2617550.

[47] 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.

[48] 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.

[49] D. Simon, C. Wimmer, B. Urban, G. Duboscq, L. Stadler, and T. Würthinger. Snippets: Taking the high road to a low level. *ACM Transactions on Architecture and Code Optimization*, 12 (2):20:1–20:25, 2015. doi: 10.1145/2764907.

[50] 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.

[51] 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.

[52] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 165–174. ACM Press, 2014. doi: 10.1145/2544137.2544157.

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

[54] TC39. Official ecmascript conformance test suite, 2016. URL https://github.com/tc39/test262.

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

[56] S. Urbanek. R benchmarks 2.5, 2008. URL http://r.research.att.com/benchmarks/.

[57] M. L. Van De Vanter. Building debuggers and other tools: We can "have it all". In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 2:1–2:3. ACM Press, 2015. doi: 10.1145/2843915.2843917.

[58] 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.

[59] C. Wimmer, V. Jovanovic, E. Eckstein, and T. Würthinger. One compiler: Deoptimization to optimized code. In *Proceedings of the International Conference on Compiler Construction*, pages 55–64. ACM Press, 2017. doi: 10.1145/3033019.3033025.

[60] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 133–144. ACM Press, 2014. doi: 10.1145/2647508.2647517.

[61] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of Onward!* ACM Press, 2013. doi: 10.1145/2509578.2509581.

[62] W. Zhang, P. Larsen, S. Brunthaler, and M. Franz. Accelerating iterators in optimizing AST interpreters. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 727–743. ACM Press, 2014. doi: 10.1145/2660193.2660223.