Efficient Interpreter Optimizations for the JVM

Gülfem Savrun-Yeniçeri Wei Zhang Huahan Zhang Chen Li
Stefan Brunthaler Per Larsen Michael Franz

Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{gsavruny, wei.zhang, huahanz, li.chen, s.brunthaler, perl, franz}@uci.edu

Abstract

The Java virtual machine is a popular target for many language implementers. Due to the unusually poor performance of hosted interpreters, many programming language implementers resort to implementing a custom compiler that emits Java bytecode instead. We studied performance of these hosted interpreters targeting the JVM and identified common bottlenecks preventing their efficient execution. First, similar to interpreters written in C/C++, instruction dispatch is expensive on the JVM. Second, Java's array semantics dictate expensive runtime exception checks, which negatively affect array performance essential to interpreters.

We present two optimizations targeting these bottlenecks and show that the performance of the optimized interpreters increases dramatically: we report speedups by a factor of up to 2.45 over the Jython interpreter, and 3.57 over the Rhino interpreter respectively. Furthermore, the performance attained through our optimizations is comparable with custom compiler performance. We provide an easily accessible annotation-based interface to enable our optimizations. Thus, interpreter implementers can expect substantial performance boosts in a matter of hours of implementation effort.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Interpreters, Optimization

General Terms Design, Languages, Performance

Keywords Interpreters, just-in-time compilers, threaded code, dynamic languages, Jython, Rhino, Java virtual machine

1. Motivation

The easiest way of getting a new dynamic language off the ground is by writing an interpreter. If that interpreter runs on top of a widely available virtual machine such as the Java Virtual Machine (JVM), the new language then becomes instantly available on all the platforms that already have the underlying VM. Unfortunately, this "hosted" approach in which one VM runs on top of another has serious performance implications. Compiler-based implementations of the dynamic language typically run much faster than hosted ones.

Hence, implementers often invest effort into building a compiler once a dynamic language becomes more popular. Among possible compiler-based implementation choices are "native" compilers that translate directly into executable machine code and "host-VM targeted" compilers that translate into a host VM's intermediate language and thereby preserve portability of the implementation. Many recent implementations of dynamic languages follow this "host-VM targeted" model, for example both Jython and Rhino implement a custom compiler in addition to their bytecode interpreters. Similar to their corresponding traditional C/C++ implementations, these custom compilers frequently emit a simple bytecode representation of the program, but usually do not perform expensive optimizations. Nevertheless, implementation costs for these custom compilers are significantly higher than the effort needed to implement an interpreter.

In this paper we investigate the performance potential of optimizing hosted JVM interpreters, with the intent to reconcile the ease-of-implementation of interpreters with the performance of custom compilers. To this end, our paper makes the following contributions:

- We illustrate the primary factors explaining why interpreters targeting the Java virtual machine are inefficient (Section 2).
- We describe the implementation of two optimizations targeting the identified inefficiencies (Section 3). We added these optimizations to the virtual machine, such that they are available to all language implementers targeting the JVM by using annotations.
- We report the results of a careful and detailed evaluation of two hosted JVM interpreters, Jython/Python and Rhino/JavaScript (Section 4). Following these data, we conclude:
  - **Performance**: We report speedups of up to a factor of 2.45 and 3.57 for Jython and Rhino, respectively (Section 4.2.1).
  - **Ease of implementation**: Manually transforming an existing bytecode interpreter to use our optimizations requires orders of magnitude less effort than implementing a custom Java bytecode compiler (Section 4.5).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPPJ'13, September 11–13, 2013, Stuttgart, Germany.
Copyright © 2013 ACM 978-1-4503-2111-2/13/09. . . $15.00.
http://dx.doi.org/10.1145/2500828.2500839

2. Efficient Interpretation

Interpreters are attractive precisely because they inhabit a sweet spot on the price/performance curve of programming language implementation. With comparatively little effort, implementers get off the ground quickly and obtain portability at the same time, too. Prior research addresses the performance problems of interpreters, identifying several techniques that have significant optimization potential. In 2003, Ertl and Gregg pinpointed the most important bottleneck for interpreters: *instruction dispatch* [12].

Each interpreter implements its own instruction set, where the interpreter needs to dispatch from one instruction to its successor, i.e., to execute an instruction the interpreter has to decode and dispatch it first. As a result, instruction dispatch is on the critical path and its performance greatly affects overall interpreter performance. When implementing an interpreter, developers usually use the classic switch-dispatch technique, illustrated in the upper half of Figure 1. In this figure, instruction dispatch encompasses the decoding of the opcode and subsequent dispatching via switch(opcode). From a performance perspective, the biggest problem with switch-dispatch is its inherent impediment to modern CPU's branch predictors. Specifically, the compiler will generate an address-table for all the case-blocks and use an indirect branch instruction to find the address for the current opcode. Consequently, *all* interpreter instructions share just *one* indirect branch instruction. Since the target for each of those indirect branches depends entirely on the interpreted program and not on the machine context, the branch prediction will invariably fail most of the time.

Several optimization techniques address this problem. For example, researchers proposed to combine multiple instructions into so called *superinstructions* [14, 25]. The rationale behind this is that for frequently executed interpreter sequences, a superinstruction, comprising the single instructions of the sequence, eliminates the necessary inter-instruction dispatches. Analogously, using a register-based architecture instead of the more common stack-based architecture helps to mitigate performance penalties incurred by instruction dispatch overhead [26]. But, both of these techniques only shift the problem, without solving it.

The technique that solves the branch prediction problem of switch-dispatch is called *threaded code* [1], and has been previously explored by several researchers [11, 19]. The basic idea of threaded code is to remove the dispatch loop, and "spread out" the instruction dispatch to each operation implementation, or i-op for short. For example, by using computed jumps (e.g., something like `goto **ip++;` for C/C++) at the end of each i-op, there are multiple indirect branches, giving branch prediction more context. If, for example, instruction B is likely to follow instruction A, then the likelihood of correct branch prediction increases. To improve this even further, Ertl and Gregg introduced a technique known as replication [13], where frequently occurring instructions, such as load instructions, are duplicated multiple times and distributed over each original instruction sequence.

The lower half of Figure 1 shows a particularly effective variant of threaded code, called *subroutine threaded code*. The guiding principle is to move the interpreter instruction operation implementation (recall the i-op abbreviation) to functions and then create a sequence of function calls for each bytecode instruction. Therefore, branch prediction is only necessary for the return instructions transferring control back to the list of call instructions. This indirect branch/return instruction is highly likely to be correctly predicted, as

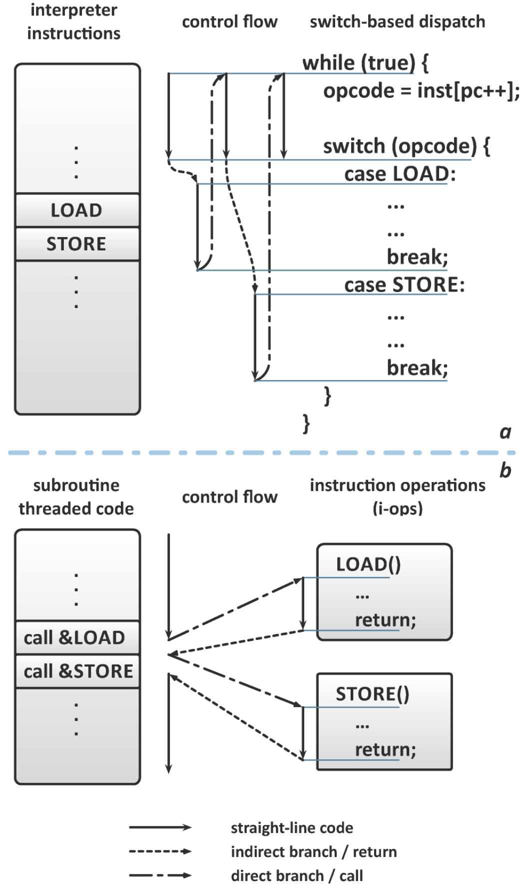

Figure 1: Switch-based dispatch vs. subroutine threading.

modern x86 CPUs include a "return address stack" for exactly this purpose. Generating subroutine threaded code, or its even more effective context-threaded code derivative [2], requires machinery similar to a small just-in-time compiler. This is due to the requirement of dynamically generating the sequence of call-instructions and subsequently executing it, whereas for other threaded code techniques, the program "lives" in data memory and the execution happens in the dispatch loop.

Unfortunately, all of the machinery required to create and execute threaded code is not available to Java programmers. It is impossible to program indirect branches in Java, such as those generated by computed goto instructions. And while it is certainly possible to create Java bytecode for a subroutine threaded code variant, its performance is not going to be on par with the manually generated sequence of call instructions.

A separate problem preventing efficient interpretation is more subtle and specific to hosted interpreters. Virtual machine interpreters use auxiliary data structures for passing operands. In a stack-based architecture, this data structure is usually referred to as the *operand stack*. Interpreter operations push operands onto the operand stack via load instructions, and pop them off in operation instructions, such as add or multiply instructions. Usually, developers use an array to implement this operand stack. Thus, pushing

operands corresponds to writing to the array, and popping operands to reading from the array.

In a stack-based virtual machine interpreter, almost every instruction is going to push its result on the array. In addition, Shi et al. [26] measured that load instructions for Java bytecode account for almost half of all executed interpreter instructions. Similar measurements for Python confirm that this observation holds for Python bytecode interpreters, too [6]. In consequence, we establish that due to this unusually high frequency, efficient interpreters need high array store performance.

In systems programming languages, such as C and C++, implementers need not give this a second thought, because arrays are implicitly low-level and therefore yield the desired performance. But in Java, array semantics are different. In addition to the `NullPointerException` and `ArrayIndexOutOfBoundsException` exceptions in the case of arrays, Java's array semantics guarantee type-safety, too. To this end, Java checks whether the insertion-candidate object matches the type of other array elements for *each* write to an array. Whenever the Java virtual machine detects a type-safety violation, it will raise an `ArrayStoreException`.

While it is known that exception checks are expensive, and that exception elimination has been actively researched and successfully addressed in previous work, the case for virtual machine interpreters is particularly pathological. The reason for this is twofold: (i) the unusually high frequency of array stores on the operand stack, and (ii) expensive nature of involved type checking operations.

3. Reusable, Annotation-based Optimizations

The previous section highlights the importance of optimizing both instruction dispatch and high performance array stores to make hosted JVM interpreters efficient. In this section, we are going to discuss our implementation of reusable, annotation-based interpreter optimizations addressing both issues. By adding these optimizations to the Java virtual machine, they become immediately available to all hosted language implementations. As a result, language implementers can leverage this foundation to unleash performance potential previously reserved for custom compilers.

3.1 Efficient Instruction Dispatch

As we described in a previous section (see Section 2), traditional threaded code implementation techniques cannot be translated to Java. This is primarily due to the restricted access to pointers, preventing the use of computed goto's or function pointers. We address this fundamental issue by adding the ability to generate efficient subroutine threaded code to the Java virtual machine. The following sections are going to address the two different perspectives to providing subroutine threaded code to implementers: the point-of-view of programming language implementers on the one hand, and the perspective of JVM implementers on the other hand.

The Language Implementer's Perspective

For a language implementer, our current full-fledged prototype implementation requires only negligible sets of changes. First, we assume that the language implementer already implemented a switch-dispatch based interpreter. Frequently, this resembles a straightforward port of an interpreter present in a system that does not target the Java virtual machine. For example, Jython's bytecode interpreter resembles the interpreter of CPython.

The following listing presents an abstracted view of this interpreter, with the `BINARY_ADD` i-op highlighted:

```
1  while (True) {
2      int opcode= instructions[pc++];
3      /* optional operand decoding */
4      switch (opcode) {
5          case Opcode.LOAD_FAST:
6              /* i-op implementation
7                 omitted
8               */
9          case Opcode.BINARY_ADD: {
10             PyObject b = stack.pop();
11             PyObject a = stack.pop();
12             stack.push(a._add(b));
13             break;
14         }
15     }
16 }
```

Listing 1: Original switch-based interpreter.

For the implementer to enable threaded code generation and execution, we require only two steps:

1. extract the i-op implementations to their own methods, and
2. add annotations to these methods.

Listing 2 illustrates these changes for the `BINARY_ADD` instruction:

```
1 @I_OP(Opcode.BINARY_ADD)
2 public void binary_add() {
3     PyObject b = stack.pop();
4     PyObject a = stack.pop();
5     stack.push(a._add(b));
6 }
```

Listing 2: After performing transformations.

Listing 2 shows that the annotation on line one binds the actual *opcode* value to the `binary_add` method. Our implementation of automated threaded code generation requires this mapping of opcodes to method addresses; the following section explains this in sufficient detail.

It is worth noting that this transformation is purely *mechanical*. This is precisely, why we believe that this transformation could be automated in future work. For example, by annotating only the dispatch loop, one should be able to automate the subsequent processing steps. But, even without automation, this task is straightforward and introduces two new lines of code per i-op (one for the annotation, and one for the method declaration), and replaces the `break` statement with a closing parenthesis.

The JVM Implementer's Perspective

The effort of adding a threaded code generator to the Java virtual machine pales in comparison to writing a full-blown just-in-time compiler. But, before delving into the implementation details let us first recapitulate what we know from threaded code. This is interesting insofar, as in a virtual machine environment we are free to support multiple threaded code techniques and the VM can choose which one to generate based on other sources of information, such as platform or profiling information. As mentioned in Section 2, subroutine threaded code is a particularly attractive derivative of the original threaded code technique. This is due to having the full context of each function's

Algorithm 1 I-Ops Code Table Initialization

```
1: procedure InitializeICT(class)
2:     for method ∈ class.getDeclaredMethods() do
3:         i_op ← method.getAnnotation()
4:         if i_op ≠ ∅ then
5:             opcode ← i_op.getAnnotationValue()
6:             ICT[opcode] ← compile(i_op)
7:         end if
8:     end for
9: end procedure
```

Algorithm 2 Threaded Code Generator

```
 1: procedure GenThreadedCode(py_code)
 2:     threaded_code ← allocate(py_code)
 3:     while py_code ≠ ∅ do
 4:         opcode ← py_code.nextOpcode()
 5:         compiled_i_op ← ICT[opcode]
 6:         address ← compiled_i_op.getEntryPoint()
 7:         code ← genCallOrJump(opcode, address)
 8:         threaded_code.append(code)
 9:     end while
10: end procedure
```

bytecode sequence mapped to native machine code. It is precisely this representation, which allows full exploitation of the speculative execution features of modern CPUs.

In consequence, a JVM implementation might choose among several different threaded code representations. For instance, it is reasonable to start out executing by generating direct threaded code interpreter at first, and switching to a subroutine threaded code interpreter only for frequently executed functions. We chose to only implement subroutine threaded code in our prototype, but there is no restriction preventing the use of multiple different kinds of threaded code.

To generate threaded code, we need to map the guest virtual machine instructions to the native machine instructions. This requires (i) decoding of guest virtual machine instructions, and (ii) finding operation implementations, the i-ops, corresponding to opcodes. The former is mostly an engineering problem, as most interpreter implementations use reasonably similar instruction encodings. An industrial-strength implementation could, e.g., support multiple different instruction encodings by having separate annotations or more parameters for each annotation. The second point, binding opcode values to i-ops, is precisely what we require from guest language implementers. To emit the actual native machine code, we rely on the Java virtual machine implementation's backend and assembler; this makes our implementation portable by design.

Algorithm 1 shows how we build an internal representation—known as the i-ops code table, or ICT for short—that maps each opcode to the native machine address of the compiled code. It is worth noting that we leverage the existing JIT compiler for generating the i-ops assembly code. Furthermore, we initialize the ICT in the static constructor of our subroutine threaded code interpreter.

Next, we need to generate the actual subroutine threaded code. Algorithm 2 presents the algorithm at the heart of our threaded code generator (abbreviated as TCG): This algorithm contains several interesting details. First of all, we see that we need to be able to decode the python bytecode representation, *py_code*. For both of the language implementations we evaluated on our prototype, the necessary changes are restricted to this part only. Then, we see how we use the ICT to map the *opcode* to the actual machine code *address*. Finally, we see that our threaded code generator emits either a call instruction or a jump instruction on line 7 via genCallOrJump. This is due to handling simple intra-procedural control flow, such as backwards jumps and returns. Context-threaded code includes inlining jump instructions into standard subroutine threaded code, too, but in contrast we do not perform its tiny inlining [2].

Inlining Control-Flow Instructions Inlining unconditional branches is straightforward. We just translate an unconditional branch into a `jmp` machine instruction, thus eliminating the corresponding `call` instruction. To find the native machine jump target, we need to extract the virtual target address from the input bytecode instruction, and convert it to the corresponding target address at machine level while assembling the `jmp` instruction.

Inlining conditional branches is not directly possible, however. The key issue preventing this is that we cannot in general infer Boolean expression evaluation logic implemented in the interpreted host language. To optimize subsequent processing using native machine instructions, we require the guest language implementers to return integer results consistent with commonly agreed upon semantics, i.e., zero for false and non-zero for true. The following listing illustrates the problem:

```
22 call   &jump_if_true
27 test   rax, rax
32 jnz    &branch_target
```

Listing 3: Emitted native machine code for conditional branch.

An Example

Listings 4, 5, and 6 explain the details of threaded code generation as described above.

```
def add(item0, item1):
  return item0 + item1
```

Listing 4: Python function `add`.

Listing 4 shows a simple Python function, `add`, that "adds" two local variables, `item0`, and `item1`. Due to dynamic typing and ad-hoc polymorphism, we will not know which operation to invoke until we witness the actual operands and choose the actual operation, e.g., numeric addition or string concatenation, at run-time.

```
0 LOAD_FAST 0 (item0)
3 LOAD_FAST 1 (item1)
6 BINARY_ADD
7 RETURN_VALUE
```

Listing 5: Python bytecode sequence for `add` function.

Listing 5 presents the Python bytecode representation of the `add` function of Listing 4, as emitted by the CPython compiler. Both of the `LOAD_FAST` instructions require an operand, zero and one respectively, to identify the corresponding local variables.

Listing 6 displays the native machine code we emit for the bytecode sequence of Listing 5. Note that we inline the immediate operands zero and one from the `LOAD_FAST` instructions directly into the generated native machine code,

thereby eliminating instruction decoding altogether. Switch-based interpreter decodes the optional operands from the instruction stream as shown in Listing 1. However, subroutine threaded code interpreter does not operate on the instruction stream, so we decode the the optional operands during threaded code generation, and inline into the threaded code.

```
 8 mov   rsi, 0x0          ;LOAD_FAST 0
13 call  &load_fast        ;LOAD_FAST 0
18 mov   rsi, 0x1          ;LOAD_FAST 1
23 call  &load_fast        ;LOAD_FAST 1
28 call  &binary_add       ;BINARY_ADD
33 call  &return_value     ;RETURN_VALUE
38 ret
```

Listing 6: Emitted subroutine threaded code for `add` function.

Putting It All Together

Algorithm 3 Putting It All Together.

1: **procedure** INTERPRET(*method*)
2:     *code* ← GETTHREADEDCODE(*method*)
3:     **if** *code* = ∅ **then**
4:         *code* ← GENTHREADEDCODE(*method*)
5:         *method*.CACHE(*code*)
6:     **end if**
7:     CALLTHREADEDCODE(*code*)
8: **end procedure**

Algorithm 3 details the actual implementation of how all parts fit together. We lazily generate subroutine threaded code at run-time, in a just-in-time fashion, i.e., before its first execution. If our code cache is of limited size, we can easily remove previously generated, cached subroutine threaded code. Subsequent invocation triggers re-generation, which requires only a linear pass and therefore is efficient. The execution of the threaded code interpreter starts by calling threaded code, i.e. transferring the control to the generated machine code.

3.2 Efficient Array Stores

The second major problem affecting hosted interpreter performance on the Java virtual machine is array-write performance. We identified the detrimental effect of preserving array type-safety for hosted interpreters. Since almost all interpreter instructions write the results of their execution to the operand stack, repeatedly verifying the type-safety of the array used to model the operand stack is expensive. This is particularly costly due to the nature of type compatibility checks incurred by hosted language implementations.

For example, Jython uses the following implementation to manage the operand stack:

```
1 static class PyStack {
2     final PyObject[] stack;
3     /* ... details omitted ... */
4 }
```

Listing 7: Jython's operand stack implementation.

Internally, `PyStack` uses an array of `PyObject` objects to implement a stack. However, `PyObject` is the root class of Jython's object hierarchy, used to model Jython's object model on the Java virtual machine. During actual interpretation, elements of the `stack` array will be instances of `PyDictionary`, `PyComplex`, `PyInteger`, etc. As a result, type checking the `stack` Java array requires repeatedly verifying that the objects actually derive from `PyObject`.

It turns out, however, that checking this exception is completely *redundant*. Since `PyObject` is the root class for all Jython-level classes, it follows that a sound interpreter implementation will exclusively operate on objects corresponding to this class hierarchy. Consequently, while checking the `ArrayStoreException` is necessary in the general case, it is strictly not necessary for a hosted interpreter operating on its own class hierarchy. Put differently, by construction the interpreter will never operate on an object not deriving from `PyObject`.

Similar to Java bytecode verification [21], we should be able to verify that the interpreter only operates on objects of the same type. We would need to implement the data-flow analysis described by Leroy [21] to apply to all i-ops and show that for all possible combinations, operands are bounded by the `PyObject`, or some other base class for another interpreter implementation. We did not, however, implement this step and leave this for future work. Our current prototype implementation provides an annotation that acts like an intrinsic and instructs the just-in-time compiler to omit the `ArrayStoreException` check.

Please keep in mind that the same argument holds not only for Jython, but for other implementation, too, such as Rhino/JavaScript, which we use in our evaluation. Similarly, for languages like Python, JavaScript, and Ruby, we can compute the maximum stack size for the operand stack and could subsequently eliminate the `ArrayIndexOutOfBoundsException` by verifying that an actual function sequence's stack height does not exceed its precomputed limit.

4. Evaluation

In this section, we evaluate the performance of our system for both Jython and Rhino. We start by explaining the system setup, and Maxine, which the underlying Java VM used in our implementation. Then, we show our benchmark results for Jython and Rhino, and analyze the effectiveness of subroutine threaded code and array store optimization. Furthermore, we analyze the performance effect of using the HotSpot server compiler. Finally, we present our results of investigating the implementation effort required by implementing an interpreter and a custom compiler.

4.1 System Setup

Hardware and Java virtual machines. We use Oracle Labs' meta-circular Maxine [29] research virtual machine to implement the previously described optimizations. Maxine's just-in-time compilation strategy eschews a mixed-mode interpretation strategy that is found in the HotSpot virtual machine, and relies on a fast, non-optimizing template just-in-time compiler, known as T1X, instead. Once profiling information embedded by T1X discovers a "hot" method, Maxine relies on its derivation of the optimizing HotSpot client compiler [20], named C1X, to deliver high performance execution. Since Maxine does not have regular release numbers, we used the Maxine build from revision number 8541 (committed on October 16th, 2012) for implementing our optimizations.

For our comparison against the HotSpot server compiler [24], we use Oracle's HotSpot Java virtual machine version 1.6.

Regarding hardware, we use an Intel Xeon E5-2660 based system, running at a frequency of 2.20 GHz, using the Linux 3.2.0-29 kernel and `gcc` version 4.6.3.

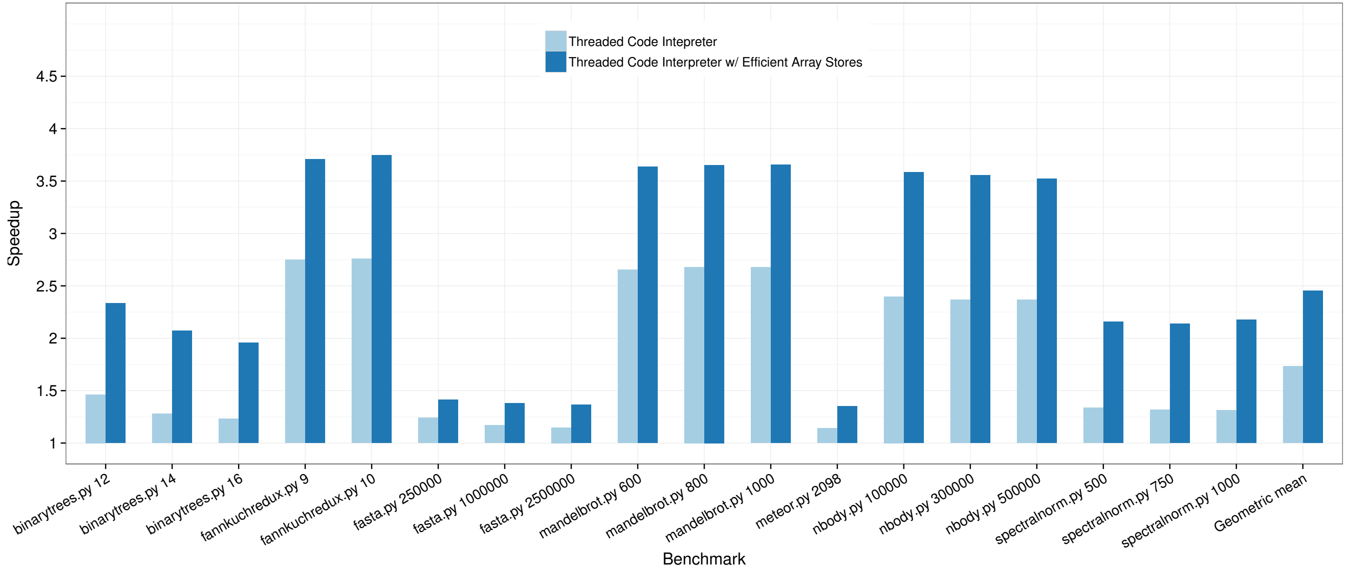

Figure 2: Speedups over Jython's switch-based interpreter.

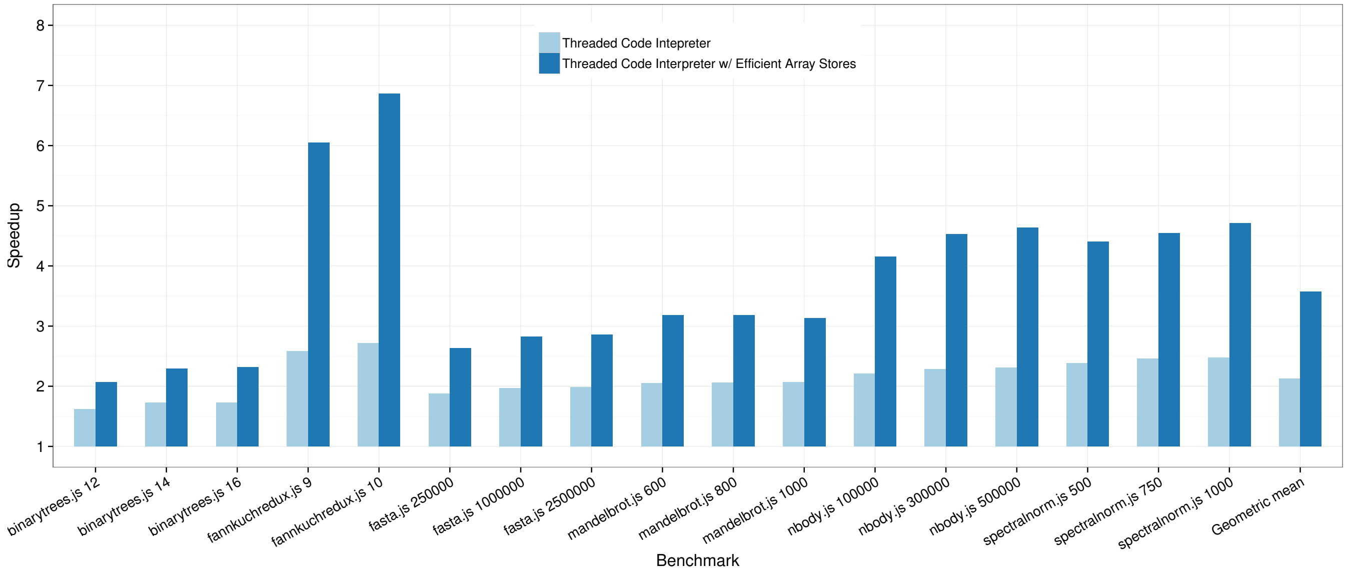

Figure 3: Speedups over Rhino's switch-based interpreter.

Interpreters. We evaluated the performance potential of our optimizations using two mature and popular interpreters targeting the Java virtual machine: Jython and Rhino, which implement Python and JavaScript, respectively. Both of these interpreters not only implement a virtual machine interpreter, but for performance reasons also implement their own custom compilers, which compile their corresponding input languages directly down to Java bytecode. Therefore, we can provide comprehensive performance evaluation that compares against both, interpreters and custom compilers. We optimized the interpreters for Jython version 2.7.0a2, and Rhino version 1.7R4.

Interpreter Benchmarks. We select several benchmarks from the computer language benchmarks game [15], a popular benchmark suite for evaluating the performance of different programming languages. We use the following benchmarks to measure the performance of our modified systems in Jython and Rhino: binarytrees, fannkuchredux, fasta, mandelbrot, meteor (only available for Jython), nbody, and spectralnorm.

Procedure We run each benchmark with multiple arguments to increase the range of measured running time. We turn dynamic frequency and voltage scaling off to minimize measurement noise [16]. We run ten repetitions of each bench-

mark with each argument and report the geometric mean over all runs.

4.2 Performance

We evaluate the performance of subroutine threaded code interpreter with and without array store optimization. First, we compare our optimizations against switch-dispatch interpreters. Next, we compare the performance of our optimized interpreters against custom Java bytecode compilers of Jython and Rhino, respectively. Then, we evaluate performance improvements resulting from improving array stores in isolation. Finally, we compare the baseline of using the client compiler, C1X, against using the HotSpot server compiler, C2.

4.2.1 Raw Interpreter Performance

Figures 2 and 3 report the speedups of our interpreter optimizations over Jython's and Rhino's switch-based interpreters, respectively. We normalize the performance by the switch-based interpreter.

We achieve a 1.73× speedup over the switch-based interpreter only from subroutine threaded code in Jython. With the array store optimization, we achieve a 2.45× speedup over the switch-based interpreter. Similarly, we achieve a 2.13× speedup from subroutine threaded code, and when combined with the array store optimization, we achieve a 3.57× speedup over the switch-based interpreter in Rhino.

The average speedup of 2.13× over Rhino's switch-based interpreter is higher than the average speedup of 1.73× over Jython's interpreter from subroutine threaded code. The reason is that Rhino executes more instructions than Jython for the same benchmarks, so the dispatch overhead is higher in Rhino. Therefore, reducing the dispatch overhead by subroutine threaded code gives higher speedups in Rhino.

The spectrum of the speedups is relatively wide. Our subroutine threaded code interpreter and array store optimization together performs better in the benchmarks that have higher dispatch overhead, such as fannkuchredux. For example, the Python version of this benchmark only has one Python function, therefore, the interpreter is only invoked once. Currently, Maxine does not support on-stack replacement (OSR) which allows the VM to replace the stack frame with that of an optimized version. Therefore, Maxine never gets a chance to recompile the switch-based interpreter for fannkuchredux.

When C1X recompiles the switch-based interpreter, it produces better code, so the switch-based interpreter performs better. For instance, the threaded code interpreter speedups are lower for the call-intensive binarytrees benchmark.

4.2.2 Custom Compiler Performance

Figures 4 and 5 show the speedups of our optimizations over Jython's and Rhino's custom Java bytecode compilers. We normalize the performance by the custom compiler performance, i.e., values lower than 1.0 indicate a slow-down relative to the custom compiler.

For Jython, subroutine threaded code achieves 70% of the performance of the custom compiler. Together with the array store optimization, it delivers 99% of the performance of Jython's custom compiler. Likewise, subroutine threaded code itself delivers 42% of the performance of the custom compiler in Rhino. In combination with the array store optimization, the subroutine threaded interpreter achieves 72% of the performance of Rhino's custom compiler.

Our average performance of 0.72× compared to Rhino's custom compiler is lower than our average performance of 0.99× compared to Jython's custom compiler. The reason is that Rhino has a more complex compiler implementing aggressive optimizations, such as data flow and type inference analyses. Rhino's custom compiler uses nine different optimization levels, and we use the highest optimization level in our evaluation.

Our subroutine threaded code interpreter with array store optimization outperforms the custom compiler in fannkuchredux, fasta, mandelbrot, and nbody benchmarks in Jython. We report the two highest speedups in fannkuchredux, and mandelbrot.

Jython's custom compiler compiles each Python program into one class file, and generates a Java method for each Python function. Maxine initially compiles each Java method using its template-based just-in-time compiler, T1X. When a method becomes hot, Maxine recompiles it using its optimizing just-in-time compiler, C1X. Hence, subsequent invocations of this method execute the optimized code. However, the Java method generated by Jython's custom compiler must be invoked at least more than a certain threshold number of times to trigger the recompilation by C1X. Fannkuchredux and mandelbrot have only one Python function that executes a hot loop. Without on-stack replacement, Maxine is not able to produce optimized code for these two benchmarks. As a result, the custom compiler does not perform well for these benchmarks.

For call intensive benchmarks, such as binarytrees, our optimized interpreter performs worse than the custom compiler. The optimizing JIT compiler is able to recompile the Java methods generated by the custom compiler at a very early stage in this benchmark.

4.3 Array Store Performance

We gain an additional 36% speedup by applying array store optimization to our subroutine threaded code interpreter for Jython. Similarly, we achieve an extra 68% speedup by applying the same optimization to our subroutine threaded code interpreter for Rhino. We counted the total number of array stores performed in our subroutine threaded code interpreters. On average, Rhino performs 21% more array stores than Jython. Moreover, we use the `perf` [22] tool to measure the number of machine instructions eliminated by this optimization. We find out that array store optimization removes 24% of the executed machine instructions on average in our subroutine threaded code interpreter for Jython. Furthermore, it reduces the executed machine instructions by 34% on average in our subroutine threaded code interpreter for Rhino. Since Rhino performs more array stores and array store optimization eliminates more instructions in Rhino, the speedup we get from this optimization is higher.

4.4 HotSpot Server Compiler Performance

Previous studies of threaded code used a traditional, ahead-of-time compiler which performs many time-consuming optimizations. In contrast, Maxine's C1X compiler—and the compiler it is modeled after, HotSpot's client compiler C1 [20]—focuses on keeping compilation time predictably low by omitting overly time-consuming optimizations. On the other hand, HotSpot's server compiler—known as C2 [24]—generates higher-quality code at the expense of higher latency imposed by longer compilation times.

To qualify the potential gain from using a more aggressive compiler, we compare the impact of our optimizations to

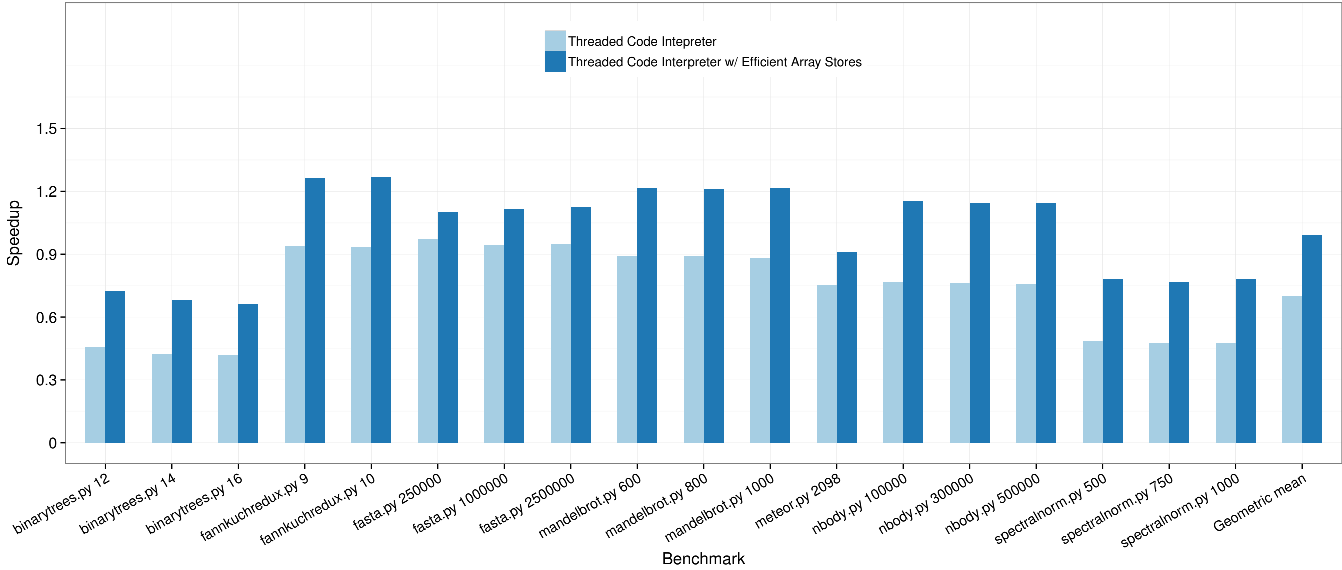

Figure 4: Performance relative to Jython's custom Java bytecode compiler.

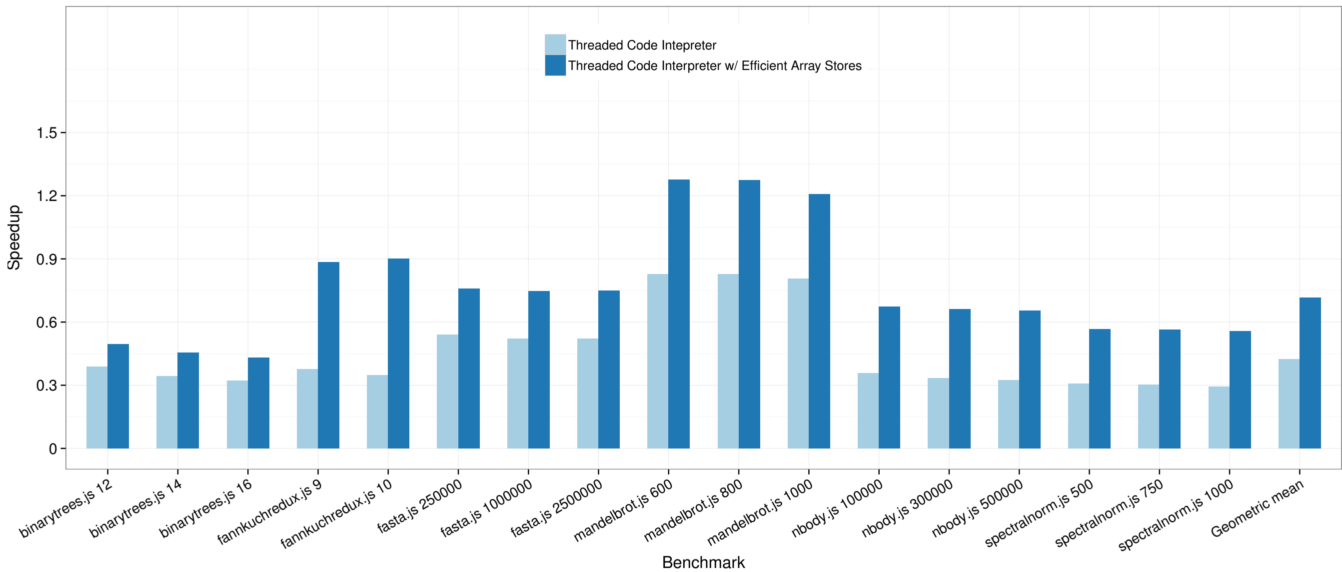

Figure 5: Performance relative to Rhino's custom Java bytecode compiler.

compiling Jython's switch-based interpreter with the server compiler. Since the server compiler will only optimize long running code, we increased the arguments when running the benchmark suite in this set of experiments. We found that the server compiler delivers 50% better performance on average. Therefore, while using an aggressively optimizing compiler does give a better baseline to the interpreter, it does not offset our performance gains, but puts them into perspective with the reported speedup potential in the relevant literature [12].

Furthermore, our technique allows the fast client compiler to outperform the server compiler without using any of the more expensive optimization techniques, which certainly has practical implications, for example in embedded systems or smartphones, where energy-efficiency is key.

4.5 Implementation Effort

4.5.1 Custom Compiler Effort

We found that some implementations targeting the Java virtual machine started out by porting their C implementation counterparts to Java. Due to the bottlenecks identified in Section 2, performance-conscious language implementers will invariably write a custom Java bytecode compiler. This investment, however, is costly in terms of initial implementation and continuous maintenance efforts.

Language	Custom Compiler Package	# Lines	Interpreter Package/File	# Lines	Reduction
Jython	org.jython.compiler	5007	org.jython.core.PyBytecode	1097	$\sim 5\times$
JRuby	org.jruby.compiler	5095			
	org.jruby.compiler.impl	6744			
	org.jruby.compiler.util	339			
	total	12178	org.jruby.ir.Interpreter.java	631	$\sim 19\times$
Rhino	org.mozilla.classfile	4183			
	org.mozilla.javascript.optimizer	5790			
	total	9973	org.mozilla.javascript.Interpreter.java	2584	$\sim 4\times$

Table 1: Implementation Effort Comparison.

Comparing the number of lines of code in an interpreter and those in a custom compiler gives an indication of the complexity of a custom compiler. Therefore, we counted the number of lines of code in an interpreter and a custom compiler for various programming language implementations. We used the `sloccount` program [10] to measure the number of Java lines of code for Jython, JRuby, and Rhino. Table 1 reports the lines of code numbers for each of these programming language implementations. The second and fourth columns list the package names counted. The third and fifth columns show the number of lines counted from these packages. The last column shows the factor of the code size reduction between the custom compiler and the interpreter of a particular implementation.

Jython's custom compiler has 5007 lines of Java code. This line count does not include the ASM Java bytecode framework [4] used by the compiler for bytecode assembling. Jython's custom compiler relies extensively on generating JVM-level calls to its runtime system to simplify the compilation process. This technique results in a relatively small and manageable compiler. Rhino has a more complicated, optimizing compiler consisting of 9973 lines of Java code.

4.5.2 Manual Transformations

As explained in section Section 3.1, the language implementer only needs to add two new lines of code for each i-op to the existing switch-based interpreter to enable threaded code generation. For example, this manual transformation results in ~ 250 new lines of Java code in Jython, and ~ 160 new lines of Java code in Rhino.

4.5.3 Virtual Machine Implementation Efforts

On the other hand, the threaded code generator for Jython requires only 337 lines of Java code. Similarly, the threaded code generator for Rhino needs 393 lines of Java code.

4.6 Discussion

Unsurprisingly, comparing against the optimized vs. switch-dispatch virtual machine interpreters of both Jython and Rhino paints a clear picture with significant speedups. It is worth noting, however, that half of the speedup is due to eliminating the expensive `ArrayStoreException` check. Interestingly, for fannkuchredux on Rhino, the dramatic speedup is due to having efficient array stores. This is due to fannkuchredux executing the most instructions, and almost all instructions cause an array store check for writing their result to the operand stack.

Regarding our comparison to custom Java bytecode compilers, we find that a Java virtual machine implementing our optimizations makes the interpreter price/performance sweet-spot even sweeter: with little manual modifications, an efficient hosted JVM interpreter is competitive with a simple custom compiler. Unlike Jython's custom compiler, Rhino's JavaScript custom compiler performs multiple optimizations. This generally decreases the potential of our optimizations—discarding the cases where Maxine cannot demonstrate its full potential due to the lack of on-stack replacement—by about a third, resulting in a performance within 60% of Rhino's custom compiler. Furthermore, we notice that the performance profile of eliminating the `ArrayStoreException` check changes noticeably. Compared with custom compiler performance, it does not contribute half, but roughly a third of the reported speedups. We have not yet investigated this in detail, but believe that the evaluated custom Java bytecode compilers do not use an array to pass operands, but rather use local variables instead.

Concerning the implementation effort, our evaluation data has two primary implications. First, using annotations for enabling our optimizations requires minimal effort by the guest language implementer; these data supports our claim that we can measure the effort in a matter of *hours*. Second, our investigation of implementation effort for the Java virtual machine implementer shows that the threaded code generators mostly diverge in supporting separate bytecode decoding mechanisms. Most of the bytecode decoding logic can be packed into separate annotations, and supporting a large set of different decoding mechanisms is mostly an engineering problem.

Regarding portability, our transformed interpreter satisfies functional portability, but it does not guarantee performance portability. Putting differently, we can still run the transformed interpreter in another JVM that does not support our optimizations. However; it may not provide good performance. Whenever the JVM implementers add our suggested optimizations into their implementation, it would improve the performance of all hosted interpreters.

5. Related Work

We group the related work into the most appropriate subsections, i.e., on interpreter optimizations and on approaches leveraging existing virtual machines. To the best of our knowledge, there is no previous work on adding interpreter optimizations to existing JIT compilers.

5.1 Threaded Code

The canonical reference for threaded code is Bell's paper from 1973, introducing direct threaded code [1]. In 1982, Kogge describes the advantages of using threaded code and systematically analyzes its performance potential [19]. In 1990, a book by Debaere and van Campenhout [11] reports the state-of-the-art for threaded code, including an in-depth

discussion of all threaded code techniques known at the time, plus the effects of using a dedicated co-processor to "offload" the interpretative overhead from the critical path of the CPU. In 1993, Curley [8] [9] explains the subroutine-threaded interpreter for Forth.

In 2003, Ertl and Gregg [12] find that using threaded code for instruction dispatch makes some interpreters more efficient than others. They describe how threaded code with modern branch prediction can result in performance speedups by a factor of up to 2.02. In 2004, Vitale and Abdelrahman [28] report dramatically lower effects on using threaded code when optimizing the Tcl interpreter. In 2005, Berndl et al. [2] introduce context threading technique for virtual machine interpreters which is based on subroutine threading. They show that context threading technique significantly improves branch prediction and reduces execution time for OCaml and Java interpreters. In 2009, Brunthaler [5] explains that the optimization potential for threaded code varies with the complexity of the i-ops.

All of the previous work in the area of threaded code focuses on interpreter implementations using a low level systems programming language with an ahead-of-time compiler. In addition to improving array store performance—which, after all, constitutes almost half of the speedup—we present threaded code as a simple, domain-specific optimization for virtual machines having a just-in-time compiler. Furthermore, wrapping threaded code generation in annotations gives implementers a straightforward way to enable efficient interpretation, which can also be reused for several different hosted interpreters.

5.2 Targeting Virtual Machines

The idea of optimizing dynamic languages in an existing JIT environment is not new. For example, the Da Vinci Machine Project [23] extends the Java HotSpot VM to run non-Java languages efficiently on top of a JVM. The project introduces a new Java bytecode instruction: `invokedynamic` [27]. This instruction's intent is to improve the costly invocation semantics of dynamically typed programming languages targeting the Java virtual machine. Similar to `invokedynamic`, our system can also be adopted by JVM vendors to improve the performance of guest language interpreters running on top of them.

In 2009, Bolz et al. [3] and Yermolovich et al. [30] describe an approach to optimize hosted interpreters on top of a VM that provides a JIT compilation infrastructure. This is particularly interesting, as this work provides a solution to circumvent the considerable implementation effort for creating a custom JIT compiler from scratch. There are two key parts in their solution. First, they rely on trace-based compilation [7] to record the behavior of the hosted program. Second, they inform the trace recorder about the instruction dispatch occurring in the hosted interpreter. Therefore, the subsequent trace compilation can remove the dispatch overhead with constant folding and create optimized machine code.

There are several differences between the approach of hierarchical layering of VMs and automatically creating threaded code at runtime. Our approach does not trace the interpreter execution, i.e., subroutine threaded code compiles a complete method, more like a conventional JIT compiler. In addition, our technique does not depend on any tracing subsystem, such as the trace compiler, recorder and representation of traces. Furthermore, our approach does not need to worry about bail out scenarios when the interpreter leaves a compiled trace so it does not require any deoptimization. Finally, we think that using both approaches together could have mutually beneficial results, though they inhabit opposing ends in implementation effort/performance spectrum.

In 2012, Ishizaki et al. [17] describe an approach to optimize dynamic languages by "repurposing" an existing JIT compiler. They reuse IBM J9 Java virtual machine and extend the JIT compiler to optimize Python. Their approach and ours share the same starting point, as we both believe that developing a JIT compiler for each dynamic language from scratch is prohibitively expensive. However, their technique is still complicated to use for programming language implementers, as they need to extend the existing JIT with their own implementation language. In contrast, our approach requires only minimal mechanical transformations that are independent of the interpreted programming language (e.g., Python or JavaScript) to generate a highly efficient interpreter.

6. Conclusion

In this paper we look into the performance problems caused by hosted interpreters on the Java virtual machine. Guided by the identified bottlenecks, we describe a system that optimizes these interpreters for multiple guest languages by using simple annotations. Specifically, we present two optimizations that improve interpretation performance to such a big extent that they become comparable to performance previously reserved for custom Java bytecode compilers. First, we optimize instruction dispatch by generating subroutine threaded code with inlined control-flow instructions. Second, we improve array store efficiency by eliminating redundant type-checks, which are particularly expensive for hosted interpreters.

To evaluate the potential of our approach, we apply this technique to two real-world hosted JVM interpreters: Jython and Rhino. Our technique gives language implementers the opportunity to execute their language efficiently on top of a Java virtual machine without implementing a custom compiler. As a result of freeing up these optimization resources, guest language implementers can focus their time and attention on other parts of their implementation.

Acknowledgments

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[1] J. R. Bell. Threaded code. *Communications of the ACM*, 16 (6):370–372, 1973.

[2] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society.

[3] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, Lecture Notes in Computer Science, pages 18–25. Springer, 2009. ISBN 978-3-642-03012-3.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[5] S. Brunthaler. Virtual-machine abstraction and optimization techniques. In *Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation, York, United Kingdom, March 2009 (BYTECODE '09)*, volume 253(5) of *Electronic Notes in Theoretical Computer Science*, pages 3–14, Amsterdam, The Netherlands, December 2009. Elsevier.

[6] S. Brunthaler. *Purely Interpretative Optimizations*. PhD thesis, Vienna University of Technology, February 2011.

[7] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: Trace compilation for the next generation web applications. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Washington, DC, USA (VEE '09)*, pages 71–80, New York, NY, USA, 2009. ACM, ACM.

[8] C. Curley. Optimizing FastForth: Optimizing in a BSR/JSR Threaded Forth. Forth Dimensons, 1993.

[9] C. Curley. Life in the FastForth Lane. Forth Dimensons, 1993.

[10] David A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount/`.

[11] E. H. Debaere and J. M. van Campenhout. *Interpretation and Instruction Path Coprocessing*. Computer systems. MIT Press, 1990. ISBN 978-0-262-04107-2.

[12] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, November 2003.

[13] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)*, pages 278–288, New York, NY, USA, 2003. ACM.

[14] M. A. Ertl and D. Gregg. Combining stack caching with dynamic superinstructions. In IVME '04 [18], pages 7–14. ISBN 1-58113-909-8.

[15] B. Fulgham. The computer language benchmarks game. `http://shootout.alioth.debian.org/`.

[16] Intel. Intel turbo boost technology – on-demand processor performance, 2012. URL `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`.

[17] 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 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 169–180, New York, NY, USA, 2012. ACM. URL `http://doi.acm.org/10.1145/2151024.2151047`.

[18] IVME '04. *Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04)*, New York, NY, USA, 2004. ACM. ISBN 1-58113-909-8.

[19] P. M. Kogge. Am architectural trail to threaded-code systems. *IEEE Computer*, 15(3):22–32, 1982.

[20] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot^{TM} client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008.

[21] X. Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[22] Linux. Perf. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[23] Oracle Corporation. Da Vinci Machine Project. `http://openjdk.java.net/projects/mlvm/`.

[24] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology - Volume 1*, JVM'01, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[25] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. pages 322–332, 1995.

[26] 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):1–36, 2008.

[27] Sun Microsystems Inc. JSR 292: Supporting Dynamically Typed Languages on the Java Platform. Early Draft, July 2011.

[28] B. Vitale and T. S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In IVME '04 [18], pages 42–50. ISBN 1-58113-909-8.

[29] C. Wimmer, M. Haupt, M. L. V. de Vanter, M. J. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4), 2013.

[30] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th Symposium on Dynamic Languages, Orlando, Florida, US, October 26th, 2009 (DLS '09)*, pages 79–88, New York, NY, USA, 2009. ACM, ACM.