A

Efficient Hosted Interpreters on the JVM

GÜLFEM SAVRUN-YENİÇERİ, WEI ZHANG,
HUAHAN ZHANG, ERIC SECKLER, CHEN LI,
STEFAN BRUNTHALER, PER LARSEN, and MICHAEL FRANZ,
University of California, Irvine

Many guest languages are implemented using the Java Virtual Machine as a host environment. There are two major implementation choices: custom compilers and so-called hosted interpreters. Custom compilers are complex to build but offer good performance. Hosted interpreters are comparatively simpler to implement but until now have suffered from poor performance.

We studied the performance of hosted interpreters 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 semantics require expensive runtime exception checks that negatively affect array performance essential to interpreters.

We present two optimizations targeting these bottlenecks and show that the performance of optimized interpreters increases dramatically: we report speedups by a factor of up to 2.45 over the Jython interpreter, 3.57 over the Rhino interpreter, and 2.52 over the JRuby interpreter, respectively. The resulting performance is comparable with that of custom compilers. Our optimizations are enabled by a few simple annotations that require only modest implementation effort; in return, performance increases substantially.

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

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Interpreters, threaded code, just-in-time compilers, dynamic languages, Jython, Rhino, JRuby, Java virtual machine

ACM Reference Format:
Savrun-Yeniçeri, G., Zhang, W., Zhang, H., Seckler, E., Li, C., Brunthaler, S., Larsen, P., and Franz, M. 2013. Efficient Hosted Interpreters on the JVM. *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 24 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. MOTIVATION

The easiest way to get 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 JVM, the new language then becomes instantly available on all the platforms that already have the underlying VM. Unfortunately, this "hosted" approach in which

---

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 gifts from Google and Oracle.
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.
Author's addresses: G. Savrun-Yeniçeri (corresponding author), W. Zhang, H. Zhang, E. Seckler, C. Li, S. Brunthaler, P. Larsen, and M. Franz, Computer Science Department, University of California, Irvine, Irvine, CA; email: gsavruny@uci.edu.
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.
© YYYY ACM. 1544-3566/YYYY/01-ARTA $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

one VM runs on top of another has serious performance implications. Compiler-based implementations of dynamic languages 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 instruction set 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 three hosted JVM interpreters, Jython/Python, Rhino/JavaScript, and JRuby/Ruby (Section 4). Following these data, we conclude:
  - **Performance**: We report speedups of up to a factor of 2.45, 3.57, and 2.52 for Jython, Rhino, and JRuby, respectively (Section 4.2).
  - **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).

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. 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* [2003b].

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 CPUs' branch predictors. Specifically, the compiler will generate an address-table for all the case-blocks and use a native indirect branch instruction to find the address for the current `opcode`. Consequently, *all* interpreter instructions share just *one* native indirect branch instruction. Since the target for each

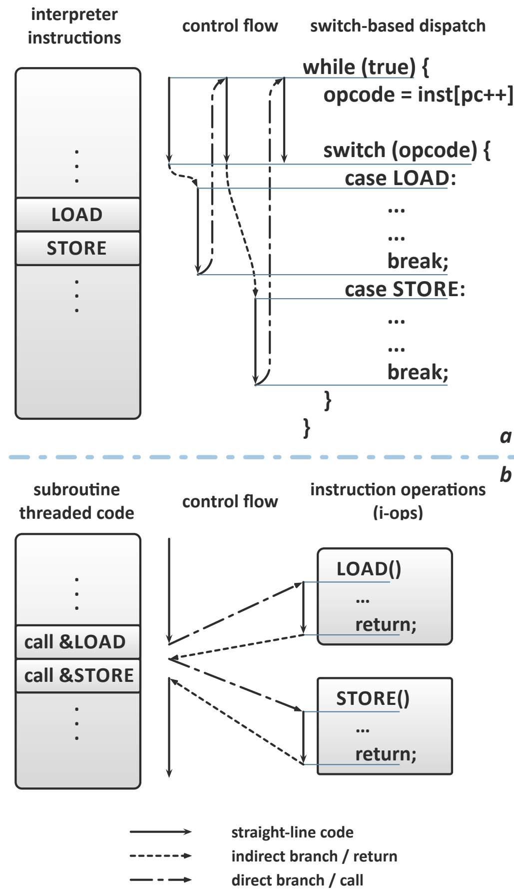


Fig. 1: Switch-based dispatch vs. subroutine threading.

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* [Proebsting 1995; Ertl and Gregg 2004]. 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 [Shi et al. 2008]. 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* [Bell 1973], and has been previously explored by several researchers [Kogge 1982; Debaere and van Campenhout 1990]. The basic idea of threaded code is to "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 [2003a], 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 native return instructions transferring control back to the list of native call instructions. This native indirect branch/return instruction is highly likely to be correctly predicted, as 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 [Berndl et al. 2005], requires machinery similar to a small just-in-time compiler. This is due to the requirement of dynamically generating the sequence of native 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.

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 onto stack. In addition, Shi et al. [2008] 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 [Brunthaler 2011]. 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 the 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) the 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 discuss our implementation of reusable, annotation-based interpreter optimizations addressing both issues. By adding these optimizations to the Java virtual

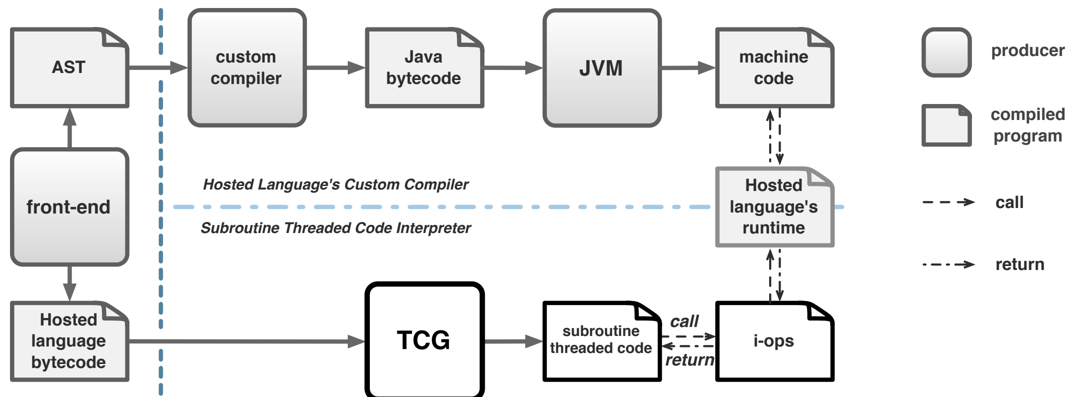


Fig. 2: Subroutine threaded code system overview.

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.

Figure 2 illustrates the overall architecture of our system compared to the hosted language's custom compiler. The front end parses hosted language's source code and generates an abstract syntax tree (AST). Then, the hosted language's custom compiler (Figure 2, upper half) compiles the AST to Java bytecode. The underlying JVM compiles the generated Java bytecode down to native machine code. On the other hand, our subroutine threaded code interpreter (Figure 2, lower half) takes the hosted language's bytecode generated by the front end, and generates threaded code by using threaded code generator (abbreviated as TCG). Threaded code is native machine code that dispatches to the hosted language's i-ops at runtime.

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 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.

Listing 1: Original switch-based interpreter.

```
while (True) {
    int opcode= instructions[pc++];
    /* optional operand decoding */
    switch (opcode) {
        case Opcode.LOAD_FAST:
            /* i-op implementation
               omitted
            */
        case Opcode.BINARY_ADD: {
            PyObject b = stack.pop();
            PyObject a = stack.pop();
            stack.push(a._add(b));
            break;
        }
    }
}
```

Listing 1 presents an abstracted view of this interpreter, with the `BINARY_ADD` i-op highlighted. 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 Python's `BINARY_ADD` instruction:

Listing 2: After performing transformations.

```
@I_OP(Opcode.BINARY_ADD)
public void binary_add() {
    PyObject b = stack.pop();
    PyObject a = stack.pop();
    stack.push(a._add(b));
}
```

Listing 2 shows that the annotation on the first line 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 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 we will first summarize 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 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. We implemented both subroutine threaded code and direct threaded code in our prototype.

To generate subroutine threaded code, we need to map the hosted virtual machine instructions to the native machine instructions. This requires (i) decoding of the hosted 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 I-Ops Code Table Initialization

```
1: procedure INITIALIZEICT(i_ops)
2:     for i_op ∈ i_ops do
3:         opcode ← i_op.GETANNOTATIONVALUE()
4:         ICT[opcode] ← COMPILE(i_op)
5:     end for
6: end procedure
```

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 when booting up the Java virtual machine.

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
```

Next, we need to generate the actual subroutine threaded code. Algorithm 2 presents the algorithm at the heart of our threaded code generator (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 native call instruction or a native 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 native jump instructions into standard subroutine threaded code, too, but in contrast we do not perform its tiny inlining [Berndl et al. 2005].

Inlining Control-Flow Instructions. Inlining unconditional branches is straightforward. We just translate an unconditional branch into a native `jmp` instruction, thus eliminating the corresponding native `call` instruction. To find the native 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 native `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 hosted 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 actual branch takes place based on the returned result. Listing 3 shows the machine code generated for a conditional branch. The i-op of JUMP_IF_TRUE returns a value indicating whether the branch is taken or not. After the comparison, the actual jump executes.

Listing 3: Emitted native machine code for a conditional branch.

```
call  &jump_if_true
test  rax, rax
jnz   &branch_target
```

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

Listing 4: Python function `add`.

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

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 5: Python bytecode sequence for `add` function.

```
LOAD_FAST 0 (item0)
LOAD_FAST 1 (item1)
BINARY_ADD
RETURN_VALUE
```

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.

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

```
mov  rsi, 0x0       ;LOAD_FAST 0
call &load_fast     ;LOAD_FAST 0
mov  rsi, 0x1       ;LOAD_FAST 1
call &load_fast     ;LOAD_FAST 1
call &binary_add    ;BINARY_ADD
call &return_value  ;RETURN_VALUE
ret
```

Putting It All Together

Algorithm 3 details the actual implementation of how all of the parts fit together. We lazily generate subroutine threaded code at runtime, 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.

Algorithm 3 Putting It All Together.

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

Direct Threaded Code Dispatch

In addition to the subroutine threaded code, our system supports the translation to direct threaded code. As described in Section 2, direct threaded code [Bell 1973] duplicates the instruction dispatch to each i-op.

We implement direct threaded code dispatch using an intrinsic `next` method. Intrinsic is a special mechanism of the underlying Java VM, and defines a set of functions that the JIT compiler compiles in a special way. The underlying JIT compiler compiles the call to `next` directly to an indirect branch. To duplicate the instruction dispatch, the language implementer needs to insert a call to `next` in each transformed i-op method (Listing 7). The initialization of the ICT in turn uses the addresses of the modified i-op methods. After that, the direct threaded code generator produces direct threaded code by replacing each opcode with the method address of its i-op. Listing 8 illustrates the generated direct threaded code. At runtime, the intrinsified `next` method fetches the address of the next i-op and performs an indirect branch to the next i-op. Note that the dispatch from one i-op to another does not allocate a new stack frame.

Listing 7: Modified i-op for direct threaded code.

```
@I_OP(Opcode.BINARY_ADD)
public void binary_add(Address[] table, int pos) {
    PyObject b = stack.pop();
    PyObject a = stack.pop();
    stack.push(a._add(b));
    next(getAddress(table, pos++), this, table, pos);
}
```

Listing 8: Translated direct threaded code for `add` function.

```
&load_fast
0 (item0)
&load_fast
1 (item1)
&binary_add
&return_value
```

As described above, the `next` instruction dispatch performs a native indirect branch instead of a call. Therefore, i-ops need to reuse the same stack frame allocated for each Python function invocation. We implement this using two special i-ops, `PROLOGUE` and `EPILOGUE`. Both of them are manually assembled instead of compiled from Java source code. `PROLOGUE`, used at the beginning of a function, allocates a stack frame that is big enough to accommodate all i-ops. `EPILOGUE`, used to model `RETURN`, deallocates the stack frame and returns. The interpretation of a Python method always start with a `PROLOGUE` and end with an `EPILOGUE`.

Stack frame reusing reduces the number of native machine instructions executed for each hosted virtual machine instruction dispatch. However, in direct threaded code, hosted virtual machine instruction dispatches of the same i-op share the same indirect branch, which comparing to subroutine threaded code dispatch decreases the likelihood of correct branch predictions.

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 `PyStack` to manage the operand stack. 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 [Leroy 2003], 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 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.

Note that the same argument holds not only for Jython, but for other implementation, too, such as Rhino/JavaScript and JRuby, 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 Jython, Rhino and JRuby. We start by explaining the system setup and Maxine, which is the underlying Java VM used in our implementation. Then, we show our benchmark results for Jython, Rhino and JRuby, and analyze the effectiveness of subroutine threaded code and 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. Moreover, we present the results from our direct threaded code interpreter for Jython. Then, we compare the baseline of using the client compiler, C1X, against using the HotSpot server compiler, C2. Furthermore, we present our results of investigating the implementation effort required by implementing an interpreter and a custom compiler. Finally, we compare the frequency of array store checks between regular Java programs and our subroutine threaded code interpreter.

4.1. System Setup

Hardware and Java virtual machines. We use Oracle Labs' meta-circular Maxine [Wimmer et al. 2013] 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 [Kotzmann et al. 2008], 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 [Paleczny et al. 2001], 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.

Interpreters. We evaluated the performance potential of our optimizations using three popular language implementations targeting the Java virtual machine: Jython, Rhino and JRuby, which implement Python, JavaScript, and Ruby, respectively. Both Jython and Rhino have a bytecode interpreter, therefore, we use their existing bytecode interpreters in our evaluation. On the other hand, JRuby does not have a bytecode interpreter, so we port the YARV (yet another RubyVM) [Sasada 2005] interpreter, which is the bytecode interpreter of Ruby in C, into Java. These three language implementa-

tions 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 implemented our system for Jython version 2.7.0a2, Rhino version 1.7R4, and JRuby version 1.7.3.

Interpreter Benchmarks. We select several benchmarks from the computer language benchmarks game [Fulgham 2002], 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, Rhino, and JRuby: binarytrees, fannkuchredux, fasta, mandelbrot, meteor (not only available for Rhino), 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 [Intel 2012]. We run ten repetitions of each benchmark with each argument and report the geometric mean over all runs.

4.2. Raw Interpreter Performance

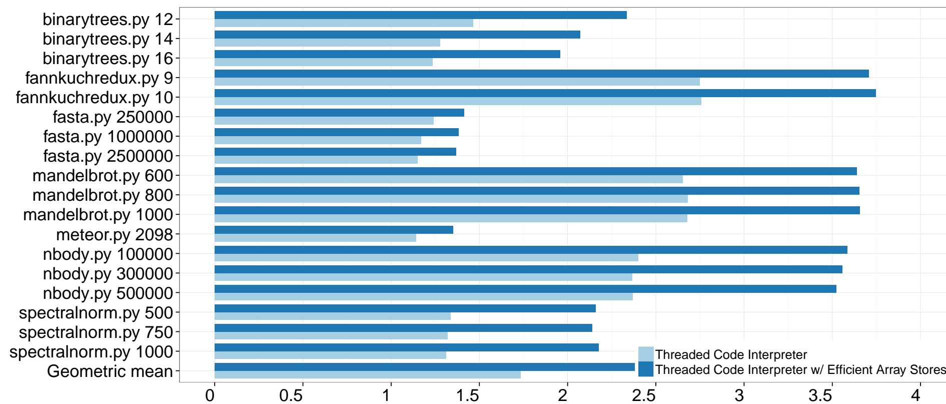
Figure 3 reports the speedups of our interpreter optimizations over Jython's, Rhino's and JRuby's switch-based interpreters, respectively. We normalize the performance by the switch-based interpreter.

We achieve a $1.73\times$ speedup over the switch-based interpreter only from subroutine threaded code in Jython. Similarly, we achieve a $2.13\times$ speedup from subroutine threaded code in Rhino. Furthermore, we gain a $1.64\times$ speedup over the switch-based interpreter by applying subroutine threaded code in JRuby.

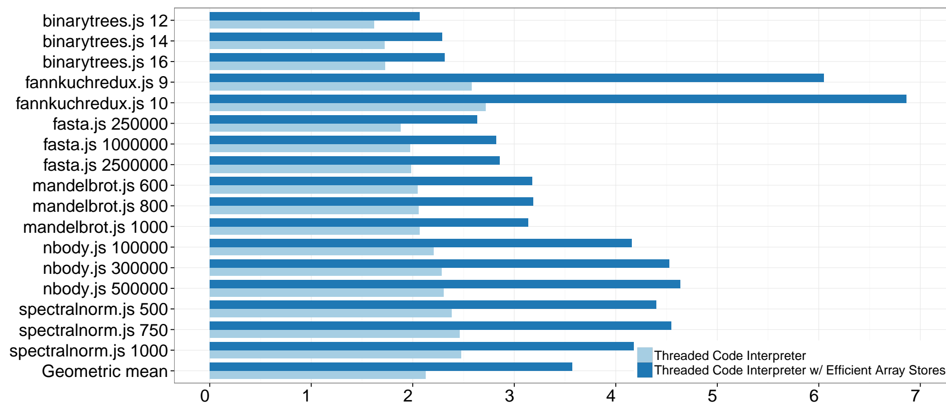
The average speedup of $2.13\times$ over Rhino's switch-based interpreter is higher than the average speedups from Jython's and JRuby's interpreter from subroutine threaded code. The reason is that Rhino executes more instructions for the same benchmarks, so the dispatch overhead is higher in Rhino.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

Efficient Array Store Performance. We gain an additional 42% speedup by applying efficient array stores to our subroutine threaded code interpreter for Jython. With the efficient array stores, we achieve a $2.45\times$ speedup over the switch-based interpreter in Jython. Similarly, we achieve an extra 68% speedup by applying the same optimization to our subroutine threaded code interpreter for Rhino. When combined with the efficient array stores, we achieve a $3.57\times$ speedup over the switch-based interpreter in Rhino. Moreover, we gain an additional 54% speedup from efficient array stores in JRuby. Together with the efficient array stores, we achieve a $2.52\times$ speedup over the switch-based interpreter in JRuby. We use the `perf` [Linux 2009] tool to measure the number of native machine instructions eliminated by this optimization. We find out that array store optimization removes 35% of the executed native machine instructions on average in our subroutine threaded code interpreter for Jython. Similarly, it reduces the executed native machine instructions by 44% on average in our subroutine threaded code interpreter for Rhino. Furthermore, it removes 31% of the executed native machine instructions on average in our subroutine threaded code interpreter for JRuby.

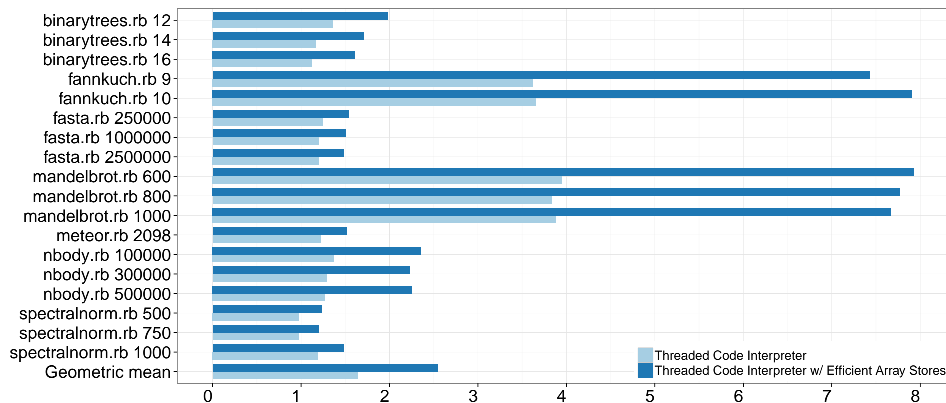
The spectrum of the speedups is relatively wide. Our subroutine threaded code interpreter and efficient array stores together performs better in the benchmarks that have higher dispatch overhead, such as fannkuchredux. For example, the Python version of this benchmark only has one Python function, therefore, the interpreter is only invoked once. Currently, Maxine does not support on-stack replacement which allows



(a) Jython



(b) Rhino



(c) JRuby

Fig. 3: Speedups of subroutine threaded code interpreter over switch-based interpreters.

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 subroutine threaded code interpreter speedups are lower for the call-intensive binarytrees benchmark.

Figure 4 shows the speedups of our direct threaded code interpreter over the switch-based interpreter for Jython. Direct threaded code itself achieves an average speedup of $1.66\times$ over the switch-based interpreter. Combined with the efficient array stores, it achieves an average speedup of $2.45\times$ over the switch-based interpreter.

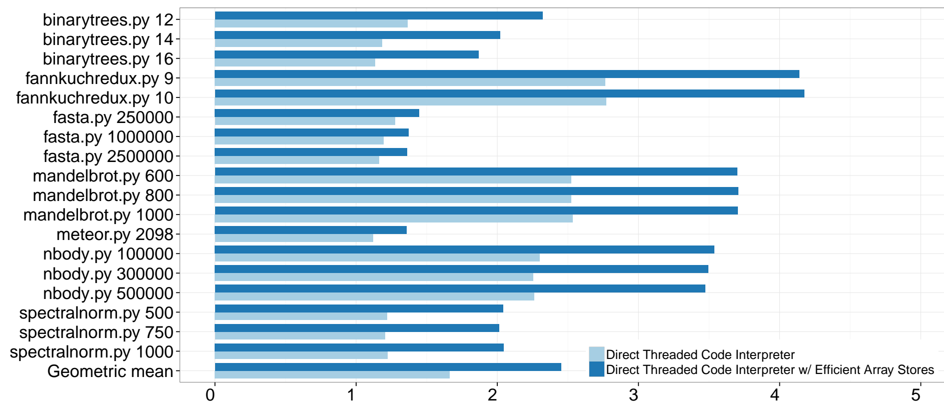


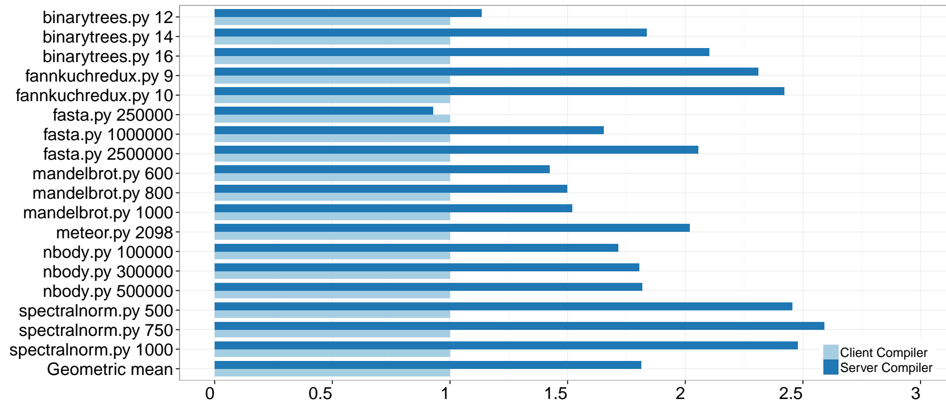
Fig. 4: Speedups of direct threaded code interpreter over Jython's switch-based interpreter.

4.3. HotSpot Server Compiler Performance

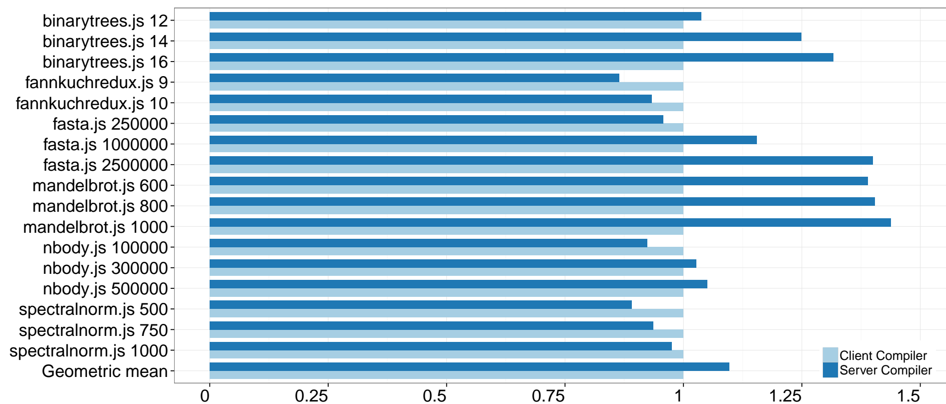
Previous studies of threaded code used a traditional, ahead-of-time compiler which performs many time-consuming optimizations. In contrast, Maxine's C1X compiler—and the compiler it is modeled after, HotSpot's client compiler C1 [Kotzmann et al. 2008]—focuses on keeping compilation time predictably low by omitting overly time-consuming optimizations. On the other hand, HotSpot's server compiler—known as C2 [Paleczny et al. 2001]—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 compiling Jython's switch-based interpreter with the server compiler. Figure 5 shows the speedups of compiling Jython's switch-based interpreter with the Java Hotspot's server compiler. We normalize the performance by the client compiler. We find that the server compiler delivers 81% better performance on average for Jython. Similarly, it achieves 10% better performance on average for Rhino. Moreover, server compiler delivers 9% better performance on average for JRuby. 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 [Ertl and Gregg 2003b].

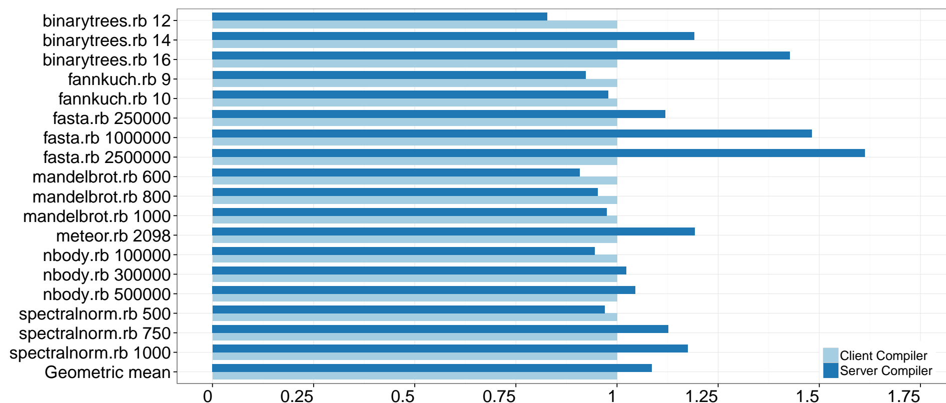
Furthermore, our technique allows the fast client compiler to outperform the server compiler without using any of the more expensive optimization techniques, which certainly has practical implications, for example in embedded systems or smartphones, where energy-efficiency is key.



(a) Jython



(b) Rhino



(c) JRuby

Fig. 5: Performance of compiling switch-based interpreters with Java Hotspot's server compiler relative to client compiler.

4.4. Custom Compiler Performance

Figure 6 shows the speedups of our optimizations over Jython's, Rhino's and JRuby's custom Java bytecode compilers, respectively. 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 efficient array stores, 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 efficient array stores, the subroutine threaded interpreter achieves 72% of the performance of Rhino's custom compiler. Moreover, subroutine threaded code brings 36% of the performance of the custom compiler in JRuby. When combined with the efficient array stores, it delivers 58% of the performance of the custom compiler.

Our average performance of $0.72\times$ and $0.58\times$ compared to Rhino's and JRuby's custom compilers are lower than our average performance of $0.99\times$ compared to Jython's custom compiler. The reason is that Rhino and JRuby have a more complex compiler implementing aggressive optimizations, such as data flow and type inference analyses. For example, 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 efficient array stores 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.

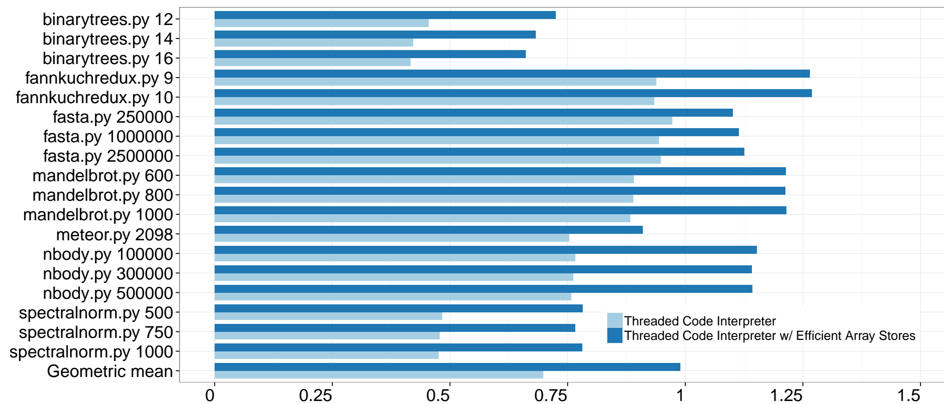
For call intensive benchmarks, such as binarytrees, our optimized interpreter performs worse than the custom compiler. The optimizing JIT compiler is able to recompile the Java methods generated by the custom compiler at a very early stage in this benchmark.

Figure 7 reports the performance of our direct threaded code interpreter for Jython compared to the compiler. Our direct threaded code achieves 66% of the performance of Jython's compiler, and with the efficient array stores it achieves 98% of the performance of the Jython's compiler.

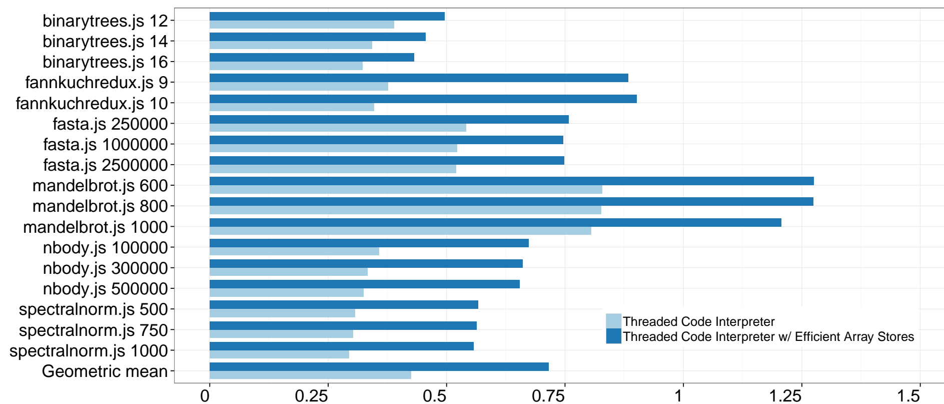
4.5. Implementation Effort

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.

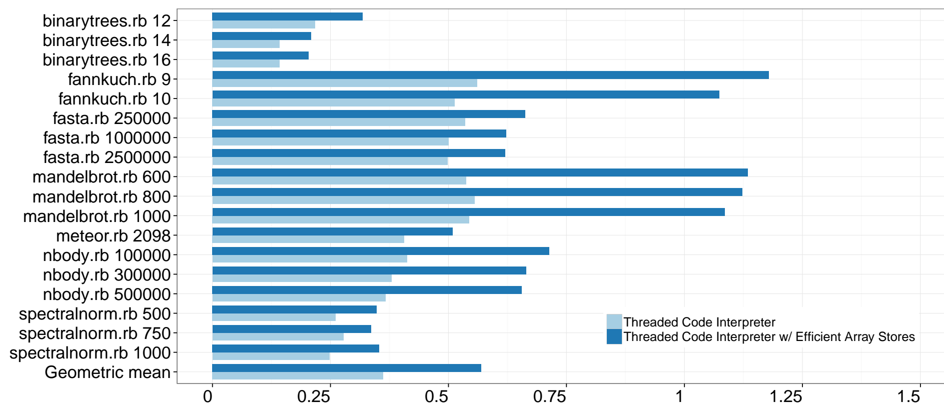
Comparing the number of lines of code in an interpreter and a custom compiler gives an indication of the complexity of a custom compiler. Therefore, we counted the number of lines of code in an interpreter and a custom compiler for various programming language implementations. We used the `sloccount` program [David A. Wheeler 2001] to



(a) Jython



(b) Rhino



(c) JRuby

Fig. 6: Performance of subroutine threaded code interpreter relative to custom Java bytecode compilers.

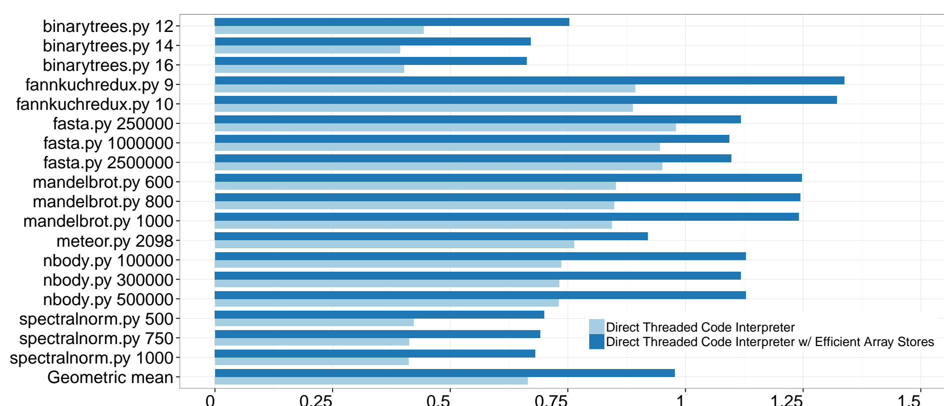


Fig. 7: Performance of direct threaded code interpreter relative to Jython's custom Java bytecode compiler.

measure the number of Java lines of code for Jython, JRuby, and Rhino. Table I 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.

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 I: Implementation Effort Comparison.

Jython's custom compiler has 5007 lines of Java code. This line count does not include the ASM Java bytecode framework [Bruneton et al. 2002] 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 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 ~ 140 new lines of Java code in Rhino, and ~ 160 new lines of Java code in JRuby.

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, and JRuby requires 317 lines of Java code.

4.6. Array Store Check Frequency

As mentioned in Section 3.2, we find that efficient array stores are crucial for the performance of the interpreters. Interpreters written in Java are atypical since they perform significantly more of array store checks. To show that array store checks in interpreters are more frequent than regular Java programs, we compare the DaCapo benchmarks [Blackburn et al. 2006] to our subroutine threaded code interpreter for Jython. We select several benchmarks from the DaCapo benchmark suite. Table II and Table III show the measurements from the DaCapo benchmarks and Jython's subroutine threaded code interpreter. The first column shows the total number of executed native machine instructions and the second column reports the total number of array store checks performed in millions for each specific benchmark. The last column shows the frequency of the array store checks per million native machine instructions.

The average frequency of the array store checks is 97 per million native machine instructions in the DaCapo benchmarks whereas it is 2,311 per million native machine instructions in our subroutine threaded code interpreter. Therefore, the frequency of the array store checks is $\sim 24\times$ that of typical Java programs. The high frequency of array store checks in interpreters means that array store optimizations are particularly effective at improving the performance of hosted interpreters.

Benchmark	# Native Machine Instructions	# Array Store Checks	Frequency
avrora	42,598	2	57
eclipse	433,621	16	39
fop	37,252	3	101
h2	230,879	47	204
luindex	24,730	2	86
lusearch	82,630	7	88
pmd	124,284	17	137
sunflow	119,884	2	17
tomcat	90,257	12	135
xalan	92,174	10	109

Table II: Array store check frequencies in millions for the DaCapo benchmarks.

Benchmark	# Native Machine Instructions	# Array Store Checks	Frequency
binarytrees 16	366,053	1,039	2,840
fannkuchredux 10	840,451	1,772	2,110
fasta 2,500,000	184,670	280	1,522
mandelbrot 1,000	168,844	373	2,217
meteor 2098	205,587	337	1,639
nbody 500,000	328,593	924	2,809
spectralnorm 1000	488,248	1,485	3,040

Table III: Array store check frequency in millions for Jython's subroutine threaded code interpreter.

4.7. Discussion

Unsurprisingly, comparing against the optimized vs. switch-dispatch virtual machine interpreters of both Jython, Rhino and JRuby shows 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 since fannkuchredux executes the most hosted virtual machine instructions and almost all instructions include an array store check before writing their result to the operand stack.

Relatively to custom Java bytecode compilers, we find that a Java virtual machine implementing our optimizations makes the price/performance ratio of interpretation even more attractive: with little manual modification, 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 (modulo 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 impact 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 investigate whether the evaluated custom Java bytecode compilers use an array to pass operands. Rhino's and JRuby's custom compilers map the local variables in the guest language to Java locals. Therefore, they operate on Java local variables and don't use arrays as operand stacks. On the other hand, Jython's custom compiler does not map Python variables to Java local variables. It generates virtual calls to Jython's runtime methods to perform operations. If the method returns a result, it pushes its result onto the Java operand stack. So, Jython's custom compiler uses Java operand stack to pass operands instead of implementing a separate operand stack. As a result, custom Java bytecode compilers do not necessarily benefit from array store optimization while interpreters gain substantial speedups from it.

Direct threaded code dispatch performs similar to subroutine threaded code dispatch when used for the hosted interpreter. It eliminates the overhead of allocating and deallocating a stack frame in each hosted virtual machine instruction dispatch. However, the direct threaded code translation only duplicates the indirect branch of the dispatch at the end of each i-op. Efficient indirect branch prediction relies on the existence of frequently repeating hosted virtual machine instruction patterns in Python programs. The translation to direct threaded code is more straightforward than that of the subroutine threaded code. It only involves replacing the opcode of each bytecode instruction with the address of the i-op and does not generate executable machine code. This simplifies the implementation of the threaded code generator. As a consequence, it is easier to generalize direct threaded code translation for different hosted languages. Although they perform similarly on modern x86 CPUs, preliminary experiments suggest that the performance of direct threaded code exceeds that of subroutine threaded code on ARM-based architectures. In future work, we will extend our system to be architecture aware, so it can automatically switch to a threaded code representation that performs better for the underlying hardware.

Concerning the implementation effort, our evaluation led to the following insights. First, using annotations for enabling our optimizations requires minimal effort by the guest language implementer; this supports our claim that we can measure the effort in a matter of *hours*. Second, our investigation of the 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 factored into separate annotations which is why we think that supporting a large set of different decoding mechanisms is largely an engineering problem.

We present both the hosted language implementer and the JVM language implementer perspective in our implementation. When our dispatch optimizations can be applied automatically, the features of threaded code generator need not be exposed to the hosted language implementer. Therefore, regular Java programmers should not be able to access to the parts of the threaded code generator that might cause security holes in the JVM. On the other hand, our array store elimination might break the JVM code if used in inappropriate places. We can add an additional verification to make it more secure. However, JVMs already provide unsafe features such as `sun.misc.Unsafe` and JNI that can also destabilize the VM.

5. RELATED WORK

We group the related work into two subsections: 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 [Bell 1973]. In 1982, Kogge describes the advantages of using threaded code and systematically analyzes its performance potential [1982]. In 1990, a book by Debaere and van Campenhout [1990] 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 [1993a; 1993b] explains the subroutine threaded code for Forth.

In 2003, Ertl and Gregg [2003b] 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 [2004] report dramatically lower effects on using threaded code when optimizing the Tcl interpreter. In 2005, Berndl et al. [2005] 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 [2009] 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 [Oracle Corporation 2008] extends the Java HotSpot VM to run non-Java languages efficiently on a JVM. The project introduces a new Java bytecode instruction: `invokedynamic` [Sun Microsystems Inc. 2011]. 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. [2009] and Yermolovich et al. [2009] 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 [Chang et al. 2009] 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 the implementation effort/performance spectrum.

In 2012, Ishizaki et al. [2012] 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 three real-world hosted JVM interpreters: Jython, Rhino and JRuby. 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.

REFERENCES

James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (June 1973), 370–372. DOI:http://dx.doi.org/10.1145/362248.362270

Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. 2005. 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)*. IEEE Computer Society, Washington, DC, 15–26. DOI:http://dx.doi.org/10.1109/CGO.2005.14

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM Press, New York, NY, 169–190. DOI:http://dx.doi.org/10.1145/1167473.1167488

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. 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)*. Springer, 18–25. DOI:http://dx.doi.org/10.1145/1565824.1565827

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

Stefan Brunthaler. 2009. Virtual-Machine Abstraction and Optimization Techniques. In *Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09) (Electronic Notes in Theoretical Computer Science)*, Vol. 253(5). Elsevier, Amsterdam, The Netherlands, 3–14. DOI:http://dx.doi.org/10.1016/j.entcs.2009.11.011

Stefan Brunthaler. 2011. *Purely Interpretative Optimizations*. Ph.D. Dissertation. Vienna University of Technology.

Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. 2009. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM Press, New York, NY, 71–80. DOI:http://dx.doi.org/10.1145/1508293.1508304

Charles Curley. 1993a. Life in the FastForth Lane *(Forth Dimensons)*.

Charles Curley. 1993b. Optimizing FastForth: Optimizing in a BSR/JSR Threaded Forth *(Forth Dimensons)*.

David A. Wheeler. 2001. Sloccount. (2001). Retrieved January 25th, 2012 from http://www.dwheeler.com/sloccount/

Eddy H. Debaere and Jan M. van Campenhout. 1990. *Interpretation and Instruction Path Coprocessing*. MIT Press. I–IX, 1–192 pages.

M. Anton Ertl and David Gregg. 2003a. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM Press, New York, NY, 278–288. DOI:http://dx.doi.org/10.1145/781131.781162

M. Anton Ertl and David Gregg. 2003b. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism* 5 (November 2003), 1–25.

M. Anton Ertl and David Gregg. 2004. Combining Stack Caching with Dynamic Superinstructions, See IVME '04 [2004], 7–14. DOI:http://dx.doi.org/10.1145/1059579.1059583

Brent Fulgham. 2002. The Computer Language Benchmarks Game. (2002). Retrieved January 10th, 2012 from http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

Intel. 2012. Intel Turbo Boost Technology – On-Demand Processor Performance. (2012). Retrieved April 25th, 2013 from http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

Kazuaki Ishizaki, Takeshi Ogasawara, Jose Castanos, Priya Nagpurkar, David Edelsohn, and Toshio Nakatani. 2012. 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)*. ACM Press, New York, NY, 169–180. DOI:http://dx.doi.org/10.1145/2151024.2151047

IVME '04 2004. *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators (IVME '04)*. ACM Press, New York, NY.

Peter M. Kogge. 1982. Am Architectural Trail to Threaded-Code Systems. *IEEE Computer* 15, 3 (1982), 22–32.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1, Article 7 (May 2008), 32 pages. DOI:http://dx.doi.org/10.1145/1369396.1370017

Xavier Leroy. 2003. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning* 30, 3-4 (2003), 235–269. DOI:http://dx.doi.org/10.1023/A:1025055424017

Linux. 2009. Perf. (2009). Retrieved May 29th, 2013 from https://perf.wiki.kernel.org/index.php/Main_Page

Oracle Corporation. 2008. Da Vinci Machine Project. (2008). Retrieved May 29th, 2013 from http://openjdk.java.net/projects/mlvm/

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, 1–12. http://dl.acm.org/citation.cfm?id=1267847.1267848

Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. 322–332. DOI:http://dx.doi.org/10.1145/199448.199526

Koichi Sasada. 2005. YARV: Yet Another RubyVM: Innovating the Ruby Interpreter. In *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM Press, New York, NY, 158–159. DOI:http://dx.doi.org/10.1145/1094855.1094912

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. 2008. Virtual Machine Showdown: Stack Versus Registers. *ACM Transactions on Architecture and Code Optimization (TACO)* 4, 4, Article 2 (Jan. 2008), 36 pages. DOI:http://dx.doi.org/10.1145/1328195.1328197

Sun Microsystems Inc. 2011. JSR 292: Supporting Dynamically Typed Languages on the Java Platform. Early Draft. (July 2011). http://jcp.org/en/jsr/detail?id=292

Benjamin Vitale and Tarek S. Abdelrahman. 2004. Catenation and Specialization for Tcl Virtual Machine Performance, See IVME '04 [2004], 42–50. DOI:http://dx.doi.org/10.1145/1059579.1059591

Christian Wimmer, Michael Haupt, Michael L. Van de Vanter, Mick J. Jordan, Laurent Daynès, and Doug Simon. 2013. Maxine: An approachable Virtual Machine For, and In, Java. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4, Article 30 (Jan. 2013), 24 pages. DOI:http://dx.doi.org/10.1145/2400682.2400689

Alexander Yermolovich, Christian Wimmer, and Michael Franz. 2009. Optimization of Dynamic Languages Using Hierarchical Layering of Virtual Machines. In *Proceedings of the 5th Symposium on Dynamic Languages (DLS '09)*. ACM Press, New York, NY, 79–88. DOI:http://dx.doi.org/10.1145/1640134.1640147