Implementing Fast JVM Interpreters Using Java Itself [*]

Michael Bebenita   Andreas Gal   Michael Franz

Department of Computer Science
University of California, Irvine
Irvine, CA 92697

ABSTRACT

Most Java Virtual Machines (JVMs) are themselves written in unsafe languages, making it unduly difficult to build trustworthy and safe JVM platforms. While some progress has been made on removing *compilers* from the trusted computing base (using certifying compilation), JVM *interpreters* continue to be built almost exclusively in C/C++. We have implemented an alternative approach, in which the JVM interpreter itself is built in Java, and runs atop a host JVM execution environment. Despite benefiting from the additional safety guarantees of the JVM runtime system, the execution overhead of our nested Java interpreter is quite acceptable in practice. Our results suggest that implementors should concentrate their efforts on optimizing just-in-time compilers rather than on interpreters. If a mixed-mode VM environment is desired, a generic JVM interpreter can subsequently be created using Java itself.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors — *Interpreters, run-time environments*

General Terms

Design, Experimentation, Performance

Keywords

Java Virtual Machine, interpreter design, minimal trusted computing base, metacircular / self-interpreters.

*This research effort was partially funded by the National Science Foundation (NSF) under grant CNS-0615443. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science foundation (NSF), or any other agency of the U.S. Government.

1. INTRODUCTION

The Java language and its Java Virtual Machine (JVM) runtime system [2, 17] have profoundly changed the way that applications are written and deployed. Code producers compile source code into JVM intermediate code ("bytecode") that is portable across computer architectures and operating systems. Programs are then shipped in this intermediate format to code consumers who use a virtual machine to execute the bytecode.

In addition to the benefit of portability, the JVM bytecode is also verifiably type-safe, allowing code consumers to check before execution that a program is well behaved with respect to certain memory and type safety properties. As a result, Java programs are generally safer and less prone to code injection vulnerabilities than programs written in legacy languages such as C.

The use of Java (and similar languages such as C# [19]) has spread to almost all computing domains, from desktop applications to high-performance computing [14], and includes even Java-based operating systems [12] and operating system components [28]. However, when it comes to building the underlying JVM interpreters, the type-unsafe C [15] and C++ [22] languages are still the far most common implementation languages.

In this paper we present J^{2}VM, a pure Java implementation of a JVM bytecode interpreter. Implementing the JVM interpreter in Java itself has a number of advantages over the traditional C or C++ based implementation approach.

C is a weakly typed language that can be understood as a thin layer above the machine language. While C abstracts the details of the specific machine instructions, it maintains the low-level semantics of the machine with regards to the memory architecture. Programmers can construct their own memory management algorithms and use arbitrary pointer arithmetic. While this flexibility allows for the fine-tuning of memory access patterns and cache behavior, it also makes C code vulnerable to buffer overflows and memory errors. If such errors exist within the implementation of a JVM bytecode interpreter, they can be exploited by malicious Java applets to escape out of the "sandbox" of the virtual machine. Considering the size of commercial JVM implementations (e.g., Sun's Hotspot VM consists of roughly half a million lines of C++ and assembly language code), it is almost inevitable that such a huge code base contains errors.

By implementing the interpreter itself in type-safe Java, we can give much stronger guarantees regarding its type safety and memory correctness, increasing the overall robustness of our JVM platform. It also makes it much easier to evolve the JVM. In the case of an interpreter that is written in an unsafe language, every change to the code base necessitates a re-certification of the complete code. In the context of a high-assurance deployment environment, this can be a very costly endeavor.

A Java-based interpreter implementation has a number of other advantages. Traditional C-based interpreters are pre-compiled to executable platform-specific machine code. This significantly limits the amount of runtime optimization and customization that can be done on the interpreter code. For example, the JVM Tools Interface (JVM TI) permits developers to debug and single step the execution of JVM bytecode. To enable this functionality, every time the JVM interpreter executes a bytecode instruction it has to check whether a debugger is attached and whether a breakpoint has been reached. At least one check and one conditional branch is incurred for *every* instruction that is executed, even if no debugger is attached—which is almost always the case in a production environment.

Similarly, debugging code in the interpreter itself can get in the way of production performance. The Hotspot VM, for example, allows execution in a more verbose mode that makes it easier to diagnose VM errors more easily. However, since the VM is pre-compiled and such diagnostic code cannot be disabled without a runtime penalty, the VM can be built in several different configurations with various debugging and optimization features enabled or disabled. To reduce the runtime overhead in configurations that support such optional features, Hotspot then self-modifies its own machine code dynamically in memory to remove or insert conditional counters or debugger invocations in the already compiled interpreter code.

A Java-based interpreter is much more flexible in this regard, because its shipped in bytecode form. Once running, it is subject to just-in-time (JIT) compilation just like any other Java code in the system. Using dead code elimination, optimizing JIT compilers can remove any overhead associated with debugging code if the VM is started with its debugging capabilities disabled. No machine-specific native code manipulation is necessary, and the entire interpreter and all optional parts can be implemented in pure high-level Java.

Similarly to removing the runtime overhead of unused debug code, Java-based interpreters are also load-time and runtime extensible in an efficient manner. We use J^{2}VM as part of an experimental trace-based just-in-time compiler framework. [9] To record traces, we extend the interpreter to monitor frequently executed bytecode traces and record them as needed. The trace monitor and recorder are implemented as regular Java programs, and when they are unneeded can be disabled via a command line switch. When disabled, the trace monitor and recorder component has a zero runtime cost since its call sites are eliminated by the underlying JIT compiler during code optimization. Such extensible virtual machines are an ideal platform for education and academic research.

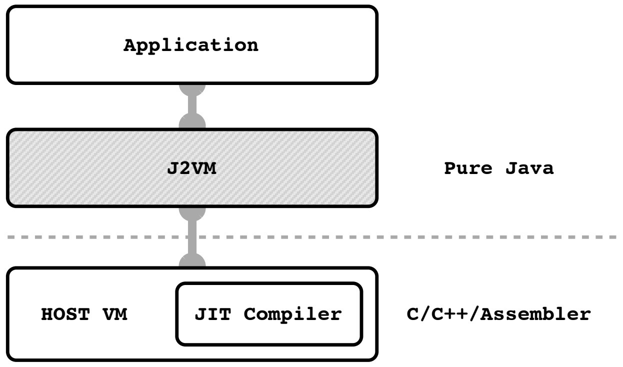

Figure 1: J^{2}VM is implemented in pure Java and executes JVM bytecode opcode by opcode. J^{2}VM runs on top of a *host VM*. The host VM can be interpreter-less (such as Jikes RVM).

J^{2}VM's enhanced safety, configurability, and extensibility does not come entirely for free. Some of the added type-safety and memory-safety checks that are implicitly provided by Java have to be performed at runtime. This causes our interpreter to be approximately 4 to 6 times slower on computationally intensive code than a comparable C-based implementation. However, as we will show in the benchmark section, this worst-case overhead is only incurred in synthetic benchmarks. In practice, the majority of performance-critical code is compiled directly into native code and hence not run by our slower interpreter. As a consequence, for typical workloads of today's JVMs, the overall performance degradation that results from using a JVM interpreter written in Java is not usually noticeable.

The remainder of this paper is organized as follows. In Section 2 we describe the overall architecture of the J^{2}VM system. Section 3 discusses how to optimize the interpreter implementation to achieve optimal performance. Our benchmarks are contained in Section 4. In Section 5 we give an overview of related work. The paper ends in Section 6 with conclusions and an outlook to future work.

2. THE J^{2}VM INTERPRETER

J^{2}VM is a JVM interpreter written in pure Java that runs on top of a traditional Java virtual machine such as Sun's Hotspot JVM or IBM's J9 VM, which we call the *host VM*. J^{2}VM executes application programs by interpreting the application code opcode by opcode. The overall architecture of a system incorporating J^{2}VM is shown in Figure 1.

For our investigation, we used Sun's Hotspot JVM for 64-bit x86 (AMD64) as the host VM. While the Hotspot VM contains a Java bytecode interpreter of its own, this is not necessary for our system. The intended application of our pure Java-based JVM interpreter is to be deployed in a system that only features a Java to native code compiler such as IBM's Jikes Research VM. Our Java-based JVM interpreter can then run on top of such a compilation-only system to provide low-latency execution for non performance-critical code.

By using a mixed-mode VM such as Hotspot for our experiments, we slightly underestimate the performance of our own interpreter since initially our interpreter is run by the Hotspot interpreter (double interpretation), which results in a severe performance degradation. However, the interpreter loop of J^{2}VM is quickly detected to be "hot" and is subsequently compiled into directly executable machine code by the Hotspot JIT compiler. For sufficiently long running programs, the performance that we measure when using Hotspot as our host VM should be reasonably close to directly compiling VMs (such as Jikes RVM).

J^{2}VM is implemented as a regular Java program that takes the name of the application to execute as command line argument. J^{2}VM then uses the class loader of the underlying host VM in combination with the BCEL [3] bytecode engineering library to load all class files referenced by the application program. Multithreaded programs are executed by creating one Java thread in the host VM running a copy of the interpreter for every thread the application starts.

J^{2}VM executes the application program by interpreting each bytecode instruction, starting with the `main` method of the application class. J^{2}VM maintains an internal data structure for the interpreter stack and local variables that is separate and distinct from the interpreter stack and local variables used by the host VM. All interpreted bytecode instructions operate on J^{2}VM's own data structures.

Simple bytecode instructions such as `iadd` are executed directly by J^{2}VM, i.e. by popping two integer values off the internal stack data structure, adding them, and pushing the result back onto the J^{2}VM-internal stack.

Both stack area and local variables are organized as one coherent array of *cells*. Each cell holds exactly one scalar data type (*integer*, *float*, *long*, *double*, *reference*), whereas *long* and *double* values occupy two consecutive cells. Method frames are allocated from this consecutive array by first reserving the number of local variable cells required by the method, and then using the remainder of the array as stack area. When a method is called, the JVM specification prescribes that arguments for the invoked method be pushed on the stack. We use this fact to propagate the arguments to the invoked method without actually copying data. For this we overlay the argument area of the stack of the caller with the local variable area of the callee. The cells pushed onto the stack become the argument part of the local variable area of the callee. The stack for the callee will then be allocated behind this new local variable area (Figure 2).

J^{2}VM supports throwing and catching exceptions. Exceptions are not thrown by our interpreter code but are instead the result of the host VM raising an exception while executing the implementation of the bytecode instruction being executed at that moment. When reading from an array, for example, we pop the array object reference and the index value from the internal stack and then pass them unchecked to the reflection interface of the host VM to execute the array access operation. If the index is out of bounds, the host VM will raise an exception, and thus making it redundant for us to check for such error conditions.

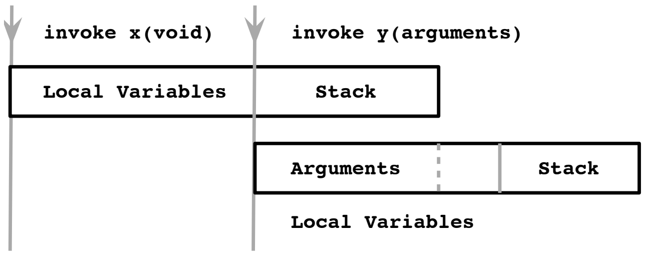

Figure 2: JVM stack and local variables are held in a contiguous array of *cells*. Local variables are allocated first in the method frame, allowing the arguments for subsequently invoked methods to become part of the activation record of the callee.

The interpreter loop of J^{2}VM is surrounded by a catch block that intercepts all exceptions and then uses the internal program counter to determine which instruction triggered the exception, and the exception handler location to branch to. The interpreter then resumes at the address of the appropriate exception handler inside the method bytecode. If no local handler is available, J^{2}VM pops the method frame from the internal stack and re-throws the exception in the surrounding scope.

Native methods cannot be executed directly by our interpreter. Instead, J^{2}VM will copy the arguments from its internal stack and invoke the requested native method via the reflection interface of the host VM.

Similarly to native methods, constructor invocations require special treatment. In the JVM bytecode language objects are allocated first using the `NEW` instruction, and subsequently one of the constructors of the corresponding class is invoked using the `INVOKESPECIAL` instruction. The bytecode verifier ensures that all objects that are allocated are initialized by invoking a constructor first before performing any other operations on the newly allocated object. The Java source code language does not preserve this semantic split between allocation and invocation. In Java source code it is not possible to allocate an object without initializing it, or to explicitly run a constructor on an object.

To deal with this semantic gap, J^{2}VM supports two execution modes: *relaxed* and *strict*. In *relaxed* mode J^{2}VM uses the reflective capabilities of the underlying host VM to allocate and initialize objects in one step. For this, the `NEW` instruction does not actually allocate an object. Instead, it merely pushes a placeholder value on the internal stack. Subsequently, when the `INVOKESPECIAL` instruction is executed to initialize the allocated object, J^{2}VM allocates and initializes the object in one step using the HostVM's reflection interface. As a result, in *relaxed* mode constructor code is not actually executed by our own interpreter but directly by the host VM.

In *strict* mode we use the `sun.misc.Unsafe` functionality of the host VM to allocate uninitialized objects and then interpret the constructor invocation as any other method call.[1] This approach is safe as long we only execute verified bytecode that is guaranteed to invoke the constructor before performing any other operation on the object. This is guaranteed since we use the host VM's class loader to load call classes referenced by the application, and the corresponding bytecode is automatically verified by the host VM.

3. OPTIMIZING THE INTERPRETER

While designing the J^{2}VM system we attempted various optimization techniques in order to improve interpretation efficiency. The unpredictably of the underlying VM makes the evaluation of optimization ideas difficult, and sometimes unreliable. Micro benchmarks results vary wildly when different VM command line parameters are used. We have tested our optimization ideas on the Hotspot Server VM with and without the `-Xcomp` switch, which forces the Host VM to compile methods before executing them. The following section describes some of the optimizations that were considered.

3.1 Virtual Machine Stack

Most C/C++ JVM interpreters simulate the Java stack using an array of 32-bit wide words. On 64-bit platforms, stack cells are 64-bits wide in order to accommodate 64-bit pointers, but still conform to Java's 32-bit stack cell semantics. Integral values can be stored directly on the stack, taking up one or two stack cells depending on the type. Due to the loose typing rules of C/C++ floating point values can be simply written into the stack data structure, even if it was defined as an array of integer words. This is not the case for Java. In Java, bit conversion must be done explicitly through a native call to `Float.floatToIntBits()` or `Double.doubleToLongBits()`. Fortunately most Java JIT compilers attempt to inline these native method calls. However this simple optimization is not guaranteed, and may not happen at all during the first phase of interpretation when the underlying VM still runs in interpretation mode. Thus, maintaining a Java array of integers (or longs) does not necessarily deliver good performance for a Java-based interpreter implementation.

Furthermore, type and memory safety in Java prevents us from converting object references into integral types, and vice versa. Strictly speaking, it is possible to retrieve an object's memory address using the `java.lang.Unsafe` class, and operate on its fields directly via unsafe `get/put` methods. This can be done by overlaying two objects, of different class types, in memory, where one object's `reference` field overlays another's `integer` field. This creates a loophole by which we can retrieve an object's reference value.

Unfortunately, this fast but unsafe way of poking objects does not work with all garbage collection algorithms. Some VMs such as the Hotspot VM use a copying garbage collector scheme where references are updated in memory as objects move within the current generation or are copied to the old generation. For this to work it is essential that the garbage collector knows precisely which values are references

[1]The `sun.misc.Unsafe` interface requires applications to run in privileged mode and is not available inside applet code. A J^{2}VM interpreter running inside an applet can only run in *relaxed* mode.

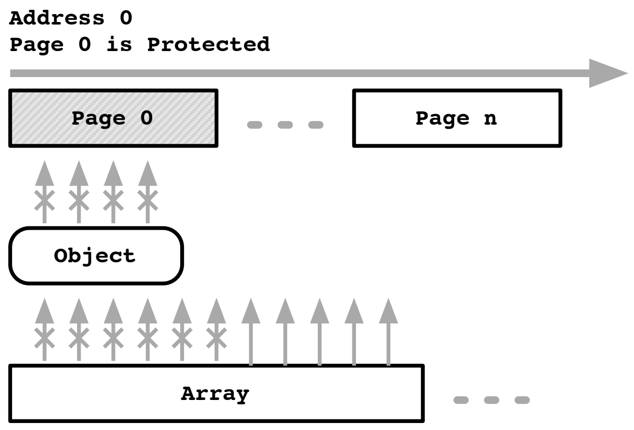

Figure 3: An explicit null check can be avoided for smaller Java objects by protecting the first few pages in memory. Dereferencing a field with a small offset relative to the null reference will then raise an exception. This optimization is not possible for arrays of unknown static size. Hence, accessing arrays in Java usually implies a dynamic null check at runtime.

and which are not. If we attempt to convert a reference into a scalar value, i.e. to store it on a stack implemented as an array of integers, the garbage collector is unable to update the reference when it moves the corresponding object and future attempts to convert that number back to a reference will result in an invalid address where the object is no longer located.

For the above mentioned reasons, the J^{2}VM maintains three separate stack locations for each of these three value type categories, one for *longs*, *doubles* and *objects*. In order to provide a fast interface with other components of the J^{2}VM system, we have chosen to implement the stack as an array of *Cell* objects, where each *Cell* object has three fields, one for each data type. We have not observed a significant performance difference between this approach and maintaining three separate arrays (instead of one array of cells).

One of the drawbacks of Java's memory safety is the incurred expense of array bounds checking. For every array access, the JVM must ensure that the array's *base* address is not *null*, and that the *index* is within the bounds of the array. Most Java VM's avoid checking for null object references by protecting the first page–4K bytes on most machines– of virtual memory, and then relying on the memory management hardware to trigger an invalid memory access exception if the first page is accessed. This is only possible because for most objects the VM can guarantee that no field offset is larger than 4K. For offsets smaller than 4K, the field address $base + offset < 4K$ whenever $base = null$, and will thus cause an exception. This optimization is not possible for arrays since computed offsets can be much larger than 4K and there is no guarantee that $base + offset < 4K$ (Figure 3).

To eliminate array bounds checks when operating on the Java stack, we use an array backed doubly linked list of stack cells. At runtime we maintain a reference that points to the top of the stack. When popping elements off the stack, we follow the linked-list pointers of the top cell instead of accessing cells via the array, and thus avoid any array bounds checks.

Since every stack cell is a Java object, our stack data structure has a considerably larger memory footprint *per stack cell* than a single word that is used to represent a stack cell in a type-unsafe interpreter. Depending on the host virtual machine, each stack cell occupies 4 to 8 words in our interpreter.

From a global perspective, however, this relative overhead *per stack cell* has only a minimal impact on the large-scale cache behavior since most Java programs use a fairly small operand stack with a few dozen stack cells. However, even for recursive programs that require a large amount of stack space, i.e. *AlphaBetaSearch* in our benchmarks set, we have not observed a significant slowdown in comparison to non-recursive programs with a small stack footprint.

3.2 Bytecode Dispatch

In general, interpreters suffer from high instruction dispatch latency. Bytecode interpreters must first fetch bytecodes by indexing a bytecode array before branching to an appropriate bytecode handler using a C `switch` statement. After completing execution, the handler jumps back up to the bytecode fetch and dispatch code. For each bytecode dispatch two memory reads occur: one for indexing the bytecode array using the current program counter (`PC`) variable, and another for finding the destination address of the jump. More importantly, two branches occur, one for dispatch and one for return. The former branch is unpredictable and causes the processor's pipeline to stall. The second is predictable but still incurs some overhead.

Optimized C/C++ interpreters use a technique known as direct threading where the addresses of bytecode handlers are known at compile time and are patched into the bytecode array at runtime before interpretation begins. The optimized interpretation loop performs one memory read to fetch the bytecode handler address followed by a computed jump. Each handler, instead of returning to the top of the interpretation loop, can fetch the next bytecode handler address and jump to it directly, thus eliminating the extra return branch (Figure 5).

Java based interpreters have no control over the instruction dispatch mechanism, and are therefore at a disadvantage over their C/C++ counterparts. We have experimented with various dispatch techniques. We have tried using the JVM's virtual method dispatch mechanism. In this approach, each bytecode handler extends an abstract `Handler` base class, and overrides a virtual `execute` method. The bytecode array is then rewritten as an array of `Handler` objects. Using virtual method invokation the interpreter loop invokes the appropriate handler methods. Although this approach comes closer to the C/C++ direct threading mechanism, it has proved less efficient than the simple `switch` mechanism.

```
goto *code[PC];              switch (bytecode[PC++])
handler_1:                   case 1:
  ...                          ...
  goto *code[PC++];            continue;
handler_2:                   case 2:
  ...                          ...
  goto *code[PC++];            continue;
handler_3:                   case 3:
  ...                          ...
  goto *code[PC++];            continue;
```

Figure 5: Structure of a direct-threaded interpreter in C using computed `goto`s compared to our interpreter which instead uses a simple `switch` statement. The C interpreter avoids the branch back to the top of the loop at the expense of an unsafe computed `goto`.

3.3 Bytecode Rewriting

Since reading bytecodes from an array would be slow, J^{2}VM rewrites bytecode on demand into a more efficient internal representation. In order to avoid array bounds checking, our VM rewrites bytecode into a directed graph of *Op* objects. Each *Op* has a pointer to the next *Op*, or to the target *Op* in case of branch instructions. During the rewriting process, *Op* objects cache values from the rewritten class' constant pool. The bytecode rewriter also converts many of the specific bytecode instructions into parameterized instructions, i.e. `ILOAD_0, ILOAD_1, ILOAD_2` etc., become `ILOAD_X` where the value of `X` is stored as a parameter in the *Op* object. The parameterization of instructions has the benefit of keeping the interpreter's main loop code smaller, which improves the chances that the underlying VM's JIT will compile the interpreter loop. Initial implementations of the J^{2}VM suffered in performance due to the Hotspot VM's inability to compile very large methods, which resulted in double interpretation.

Due to the high instruction dispatch latency in the interpreter, the bytecode rewriter attempts to collapse frequent sequences of instructions into macro instructions, thus eliminating some of the cost associated with instruction dispatch. Previous research [20] has show that a relatively low number of bytecode sequences occur very frequently. Among the most frequent are `ALOAD_0 GETFIELD` which is a common pattern when an instance method tries to access an instance field, and `ILOAD ILOAD` which appears frequently in computationaly intense applications. The parameterization of instructions reduces the total number of instructions and allows us to cover a wider range of Java bytecode instructions with macro instructions. At the same time it enables a faster and leaner implementation of the main interpreter loop.

4. BENCHMARKS

In this section we will evaluate the performance of J^{2}VM relative to the highly optimized C/C++ and assembly language-based interpreter implementation of Sun's Hotspot 1.5 JVM. As a benchmark set we chose the Java Grande 2 [18] benchmark suite, because it is freely available, and it offers a wide range of problem sizes ranging from small scale micro benchmarks (section 1) and benchmark kernels (section 2) to larger size problems (section 3).

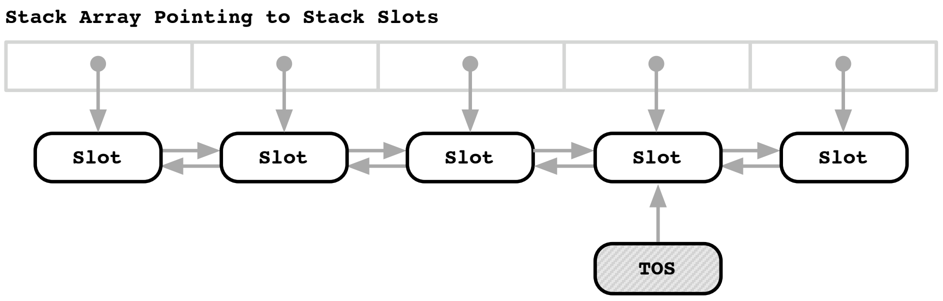

Figure 4: Stack slots can be accessed either by indexing the stack array, or by traversing the linked list starting at the top of stack (TOS) pointer. For example, `IADD` can be implemented as `TOS.prev.int = TOS.int + TOS.prev.int; TOS = TOS.prev;`

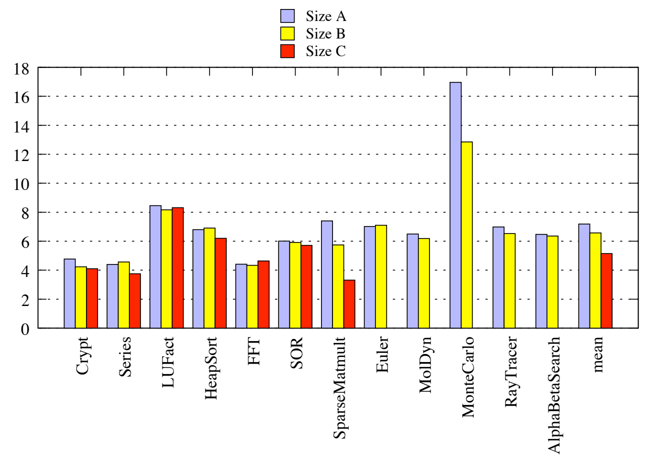

Figure 7: Relative performance of J^{2}VM in comparison to that of Sun's Hotspot JVM 1.5 interpreter for sections 2 and 3 of the Java Grande 2 benchmarks. For most benchmark programs, our interpreter is within factor 4-9x of the speed of the C/C++ and assembly language implementation.

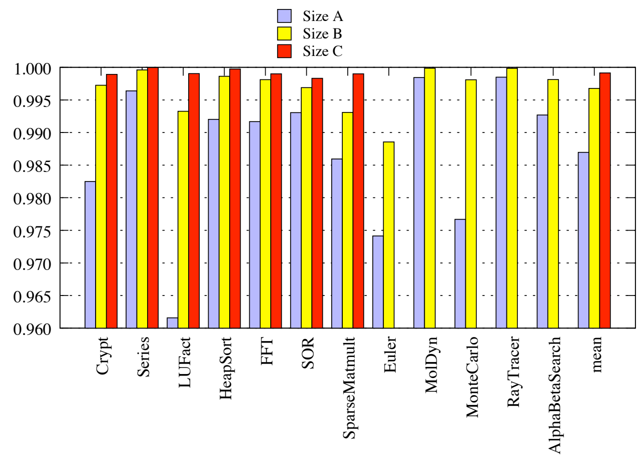

Figure 8: Proportion of overall execution time spent in compiled code for sections 2 and 3 of the Java Grande 2 Benchmarks. Even for the smallest problem size (Size A) which runs for a few seconds only, at least 96% of the overall execution time is spent executing compiled machine code rather than interpreting bytecodes. For larger problem sizes, the impact of interpreter speed on overall performance is almost negligible.

Java Grande 2 does not define problem size C for Section 3, only sizes A and B. Thus no results are shown in the graphs for Section 3 programs running problem size C.

The results for the Java Grande 2 micro benchmarks (section 1) are shown in Figure 6. For simple bytecode instructions such as arithmetic instructions, assignments to local variables, and casts, J^{2}VM is between 2 and 10 times slower than the Hotspot interpreter, with an average slowdown factor of approximately 6 (Figure 6(a)). This overhead is a result of the less optimal branching profile of the Java implementation, with two branches per instruction instead of the direct dispatch of the Hotspot interpreter.

In contrast, for array allocations, our interpreter achieves a slight *speedup* over the Hotspot interpreter for most benchmark tests (Figure 6(b)). This is a result of the JIT compiler being able to generate more optimal code for the Java implementation than the C/C++ compiler for the equivalent C/C++ interpreter implementation of the Hotspot VM. Whereas the Hotspot VM interpreter has to use a function call to the memory management subsystem to allocate the arrays, the code generated by the JIT compiler uses a more aggressive approach and inlines the fast path of the object allocation code directly into the generated machine code. By avoiding the method invocation overhead, our interpreter is able to outperform the C/C++ implementation.

For object allocations, however, we are not able to achieve the same speedup (Figure 6(c)), because for every allocation, we have to construct a call via the host VM's reflection interface to allocate the object. For this, amongst others, we have to allocate another temporary array to hold the parameter values, which explains the slowdown of approximately factor 7 for this test case.

For complex mathematical operations, the result differs significantly from the factor 6 slowdown observed on average for simple arithmetical operations (Figure 6(d)). While for

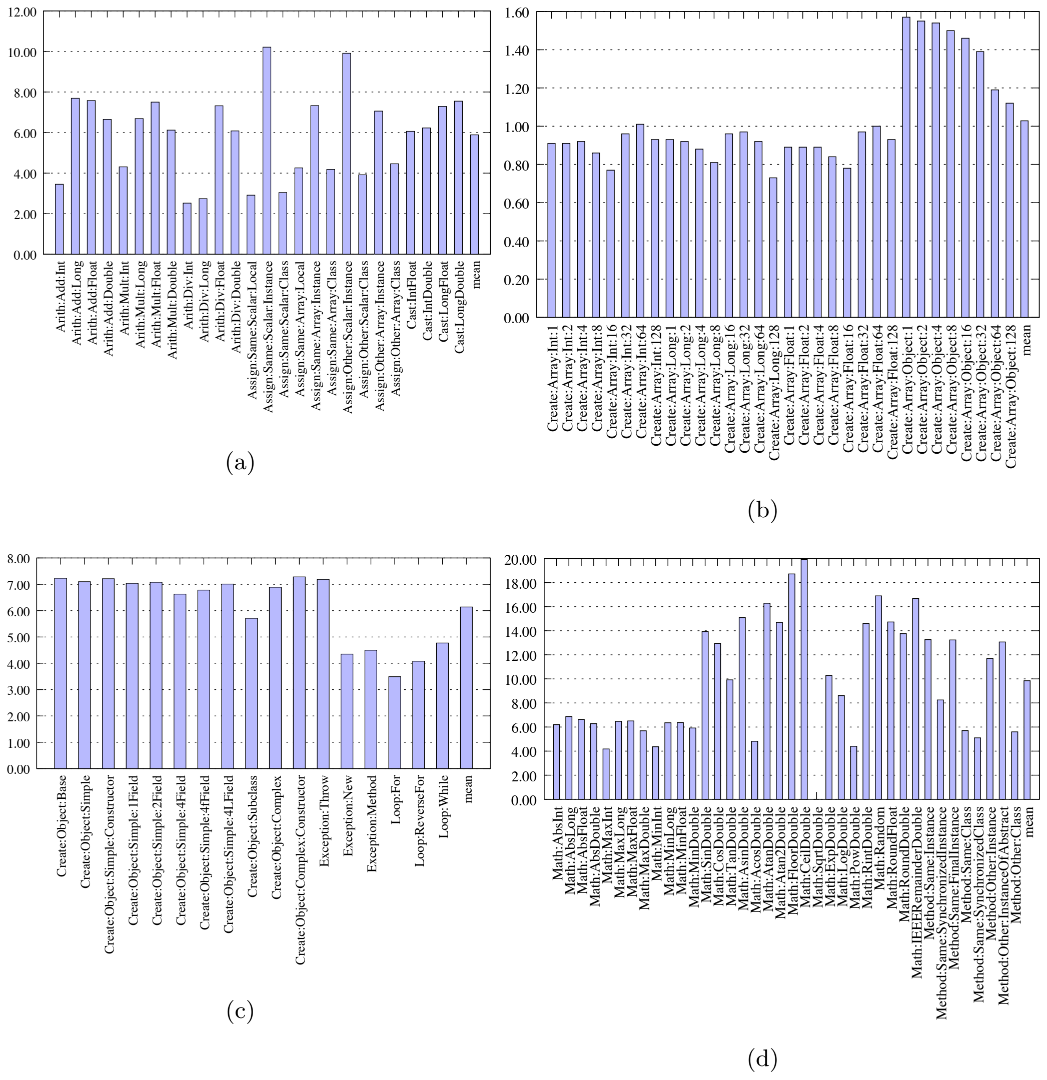

Figure 6: Relative performance of J^{2}VM in comparison to the highly optimized C/C++ and assembly language implementation used by Sun's Hotspot JVM 1.5. Each bar shows the slowdown relative to Hotspot VM in interpretation-only mode (JIT compiler disabled). The result for `Math:SqrtDouble` is a slowdown of 60 and is not shown in the graph for better readability.

some operations such as `abs`, `max`, and `min` we achieve a similar performance, other operations including `sin` and `floor` are 14 to 60 times slower than the Hotspot interpreter. This is a result of the Hotspot interpreter inlining certain complex mathematical operations instead of invoking the corresponding native code via the Java Native Interface (JNI). Our interpreter does not perform this optimization and thus has to construct an expensive reflective native method code invocation. We will show below, however, that this kind of micro optimization of the interpreter is in fact questionable since it would only affect computational intensive code which in turn is unlikely to undergo frequent interpretation and is translated to machine code by the JIT compiler instead.

The results for section 2 and 3 of the Java Grande 2 benchmark are shown in Figure 7. For most benchmark programs our interpreter is within factor 4-9x of the speed of Hotspot's C/C++ and assembly language implementation of the interpreter. The average relative performance improves from approximately factor 9 to factor 7 as the problem size increases. We attribute this mostly to cache effects. For the largest problem size, cache misses become more prevailing and thus overshadow the slower performance of the interpreter.

As shown in Figure 7, the slowdown for the MonteCarlo benchmark is significantly higher than for the other benchmarks. This results from the high frequency of native method calls performed by the MonteCarlo benchmark to obtain random numbers from the random number generator (which is implemented as native code). Since our native method invocation interface requires us to allocate an array of arguments at every native method call, our overall performance

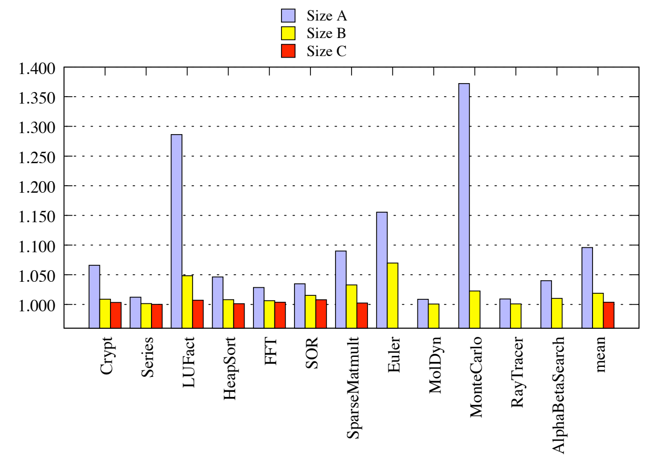

Figure 9: Simulated performance for a hypothetical Java system that uses Sun's Hotspot JVM 1.5 just-in-time compiler but our J^{2}VM interpreter. Performance is estimated relative to the original Hotspot JVM using its original interpreter. The interpretation slowdown of factor 4-9x of J^{2}VM affects only the results of the smallest problem size benchmark (Size A). For the larger problem sizes, the overall performance impact is negligible.

suffers significantly for such benchmarks that heavily rely on native method code.

To evaluate whether the slowdown we have measured above significantly impacts the overall performance of a mixed mode virtual machine where frequently executed bytecode is compiled to native machine code instead of interpreting it over and over, we measured the fraction of runtime the Hotspot VM spends interpreting bytecode vs directly executing Java code in compiled machine code form. For this, we modified the Hotspot VM interpreter to count the number of bytecode instructions it *interprets*.[2] We then ran the Java Grande 2 section 2 and section 3 benchmarks for all problem sizes (A, B, and C) first in mixed mode, and then in interpretation only mode. For each mode we counted the number of instructions interpreted.

In mixed mode, frequently executed code is compiled to machine code by the JIT compiler, and thus the VM *interprets* fewer bytecode instructions. The difference between the number of instructions interpreted in mixed mode (with compilation) and interpretation mode can be used to estimate the fraction of runtime spent interpreting code vs running compiled code.

Figure 8 shows the result of our analysis. Even for the smallest problem size (Size A), which runs for a few seconds only, at least 96% of the runtime is spent running compiled code. For larger problems (Size B) more than 99% of the runtime is spent in compiled code. For the largest problem size (Size C) the time spent interpreting instructions is essentially negligible. This extreme imbalance certainly only applies to frequently executed (computationally intensive) code which lends itself to JIT compilation. However, for code that is not executed frequently, and thus would not run as compiled code but instead in our interpreter, the resulting overall slowdown for the application would be negligible since that code is infrequently executed.

For "hot" code that dominates the program runtime, our slower interpreter only affects the 4% to 0.01% of the execution runtime that is not spent in compiled code. The overall slowdown of our system would be significantly smaller than 4 to 6 if J^{2}VM would directly interface with the JIT compiler and offload execution of "hot" code to the JIT compiler subsystem. While this is currently not the case in our prototype system running on top of Hotspot VM, this is merely an implementation limitation. Future versions of J^{2}VM will directly interface with a JIT compiler and support on-stack replacement of interpreted code with the compiled machine code equivalent.

To estimate the performance of such a mixed mode system that uses a pure Java-based JVM interpreter, we have applied the results of our analysis in Figure 8 to simulate the execution time of the Java Grande 2 benchmark running on the Hotspot VM using Hotspot's compiled code performance, but with our 4-9x times slower interpreter. The results are shown in Figure 9. The slowdown of our interpreter does visibly affect the results of the smallest problem size benchmark (Size A). For the larger problem sizes, on the other hand, the overall performance impact is negligible.

Interestingly, for the larger problem sizes the Euler benchmark has the highest overall slowdown (1.07), and not MonteCarlo as one might have expected. This is the case because MonteCarlo actually spends more time in compiled code than Euler (see Figure 8).

5. RELATED WORK

A number of related works have used Java to implement virtual machines or virtual machine components.

The JavaInJava [25] system by Taivalsaari et al. was one of the first JVMs implemented in Java. JavaInJava was not optimized for performance and was thus orders of magnitude slower than a C/C++ based interpreter. Taivalsaari argued that with further optimizations, a slowdown of 10-20 could be achieved, but that it would be "rather difficult to reach better performance than that without cheating" [25] due to the safety and sanity checks performed by the Java runtime system. J^{2}VM in fact reduces this overhead to a slowdown of 4-6 for large problem sizes. We attribute this in part to advances to dynamic compilation since 1998 when Taivalsaari's work was originally published.

A popular research VM for Java is IBM's Jikes RVM (formerly Jalapeño). [5]. It implements a complete Java VM in Java, and uses its own compiler to bootstrap itself. While Jikes RVM implements most common JVM components such as a just-in-time compiler and a garbage collector in Java, it does not actually have a bytecode interpreter. Instead, it uses a quick but not highly optimizing "baseline" compiler

[2]The Hotspot VM already supports counting bytecodes but uses a 32-bit counter which overflows within seconds on a modern machine. We implemented a 64-bit counter that can precisely track the number of instructions executed even for large benchmark programs.

to execute infrequent code.

In theory, Jikes RVM would be an ideal target for our Java-based interpreter. The main reason that we decided to not use Jikes RVM in our experiments and instead rely on Sun's Hotspot VM as host virtual machine is that Jikes RVM is a research prototype that usually trails the most recent Java specification and standard libraries. We also found that Jikes RVM as a research prototype contains quite a few highly version and release specific regressions that make it difficult to use for comparative benchmarks.

Besides JavaInJava, the Joeq VM [26] is the only other JVM interpreter written in Java that we are aware off. However, Joeq's interpreter makes extensive use of unsafe Java extensions and direct untyped memory access. Joeq is also significantly slower than J^{2}VM because its represents the stack as a single array of objects, and every time a value is pushed onto it, the value has to be boxed into an object.

The Squawk Java VM [21] uses a subset of the Java language to implement the interpreter and other VM components. This subset can be understood (with minor modifications) by both, a Java compiler and a C compiler. The latter is used to actually build the Squawk executable. Thus, Squawk's interpreter code is actually compiled as C code, similar to other C/C++ implementions of JVM such as Sun's Hotspot VM [24], IBM's Java VM [23], and open source systems such as Kaffe [27], Cacao [16], and SableVM [8].

While attempts to implement a JVM interpreter in Java are rare, a number of Java-based interpreters and emulators for other languages and systems exists. In previous work [10], we used the Java VM to execute PowerPC, MIPS and ARM Linux binary code by translating ("upcompiling") machine code instructions to equivalent Java bytecodes. The NestedVM [1] uses a similar approach to execute arbitrary C code on top of JVM by compiling it to MIPS native code, which is then in turn executed by a MIPS interpreter running on top of the JVM. Both systems achieve a reasonable performance, in particular when dealing with legacy code that was written for previous generations of machines.

Another fast emulator running on top of the JVM is the Playstation emulator by Howitt et al. [13], which emulates the MIPS CPU of a Playstation computer as well as associated peripheral devices. Howitt describes a series of optimizations that were used to improve the performance of the emulator, some of which are closely related to our work.

Our approach of running a VM on top of a core VM (as compiled code or an actual core interpreter) is closely related to Smalltalk's approach of meta interpretation [11]. Folliot et al. [7] proposed a similar system for Java called Virtual VM (VVM).

The optimizations of our Java implementation of the interpreter are closely related to the work on threaded code by Bell [4]. Our use of macro instructions to replace frequently executed bytecode sequences was inspired by Ertl's work on superinstructions [6] in the Forth interpreter, as well as O'Donoghue's et al.'s work on identifying a generic set of superinstructions for Java programs [20].

6. CONCLUSIONS AND OUTLOOK

We have presented J^{2}VM, a JVM interpreter written in pure Java. While our current prototype runs on top of a mixed-mode VM that already contains a C/C++ based interpreter, J^{2}VM could also be used on top of VMs that rely entirely on compilation. An example would be IBM's Jikes RVM, which provides no interpreter but several compilers spanning the range from "quick but dirty" to "slow but optimizing".

Our performance evaluation shows a 4-9x slowdown versus a highly optimized C/C++ interpreter on synthetic benchmarks. However, extrapolating from the behavior of Sun's mixed-mode HotSpot JVM, we show that the performance of the interpreter is not really all that critical and such a slowdown can be tolerated easily. Computationally intense code executes in compiled form most the time and is therefore not affected by a slower interpreter. For rarely executed code, on the other hand, the slowdown is essentially irrelevant.

In summary, we believe that it is time to abandon C and C++ as implementation languages for JVM interpreters. Instead, VM implementors should focus on providing a Java code execution environment through efficient dynamic code generation. Interpretation of performance-uncritical code can then be added atop such a system in a type-safe manner using only the features of the Java language itself, and at a minimal overall runtime expense.

As far as future work is concerned, our immediate plan is to directly link J^{2}VM to the dynamic compiler of the underlying JVM platform in order to permit compiled execution of frequently executed code fragments. This will enable us to directly measure the performance of J^{2}VM in a mixed-mode execution environment instead of having to estimate the performance by splitting out and scaling the interpretation fraction of the Hotspot VM's execution time.

7. REFERENCES

[1] B. Alliet and A. Megacz. Complete translation of unsafe native code to safe bytecode. In *IVME '04: Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators*, pages 32–41, New York, NY, USA, 2004. ACM Press.

[2] K. Arnold and J. Gosling. *The Java Programming Language (2nd ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[3] BCEL homepage. http://jakarta.apache.org/bcel/.

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

[5] M. Burke, J. Whaley, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, and H. Srinivasan. The Jalapeño dynamic optimizing compiler for Java. *Proceedings of the ACM 1999 Conference on Java Grande*, pages 129–141, 1999.

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

[7] B. Folliot, I. Piumarta, L. Seinturier, C. Baillarguet, C. Khoury, A. Léger, and F. Ogel. Beyond flexibility and reflection: the virtual virtual machine approach. In *NATO Advanced Research Workshop, Environments, Tools and Applications for Cluster Computing*, pages 17–26, Springer-Verlag, 2002.

[8] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01): April 23–24, 2001, Monterey, California, USA. Berkeley, CA*, 2001.

[9] A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine.* PhD thesis, University of California, Irvine, September 2006.

[10] A. Gal, M. Yang, C. Probst, and M. Franz. Executing Legacy Applications on a Java Operating System. *2004 ECOOP Workshop on Programming Languages and Operating Systems, Oslo, Norway, June*, 2004.

[11] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[12] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX Operating System. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association.

[13] M. Howitt and G. Sanderson. Writing a Sony PlayStation Emulator in Java Technology. Technical report, Lombardi Software, 2006.

[14] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, 2000.

[15] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[16] A. Krall and R. Grafl. CACAO: A 64 bit Java VM just in time compiler. In *PPoPP Workshop on Java for Science and Engineering Computation*, 1997.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[18] J. Mathew, P. Coddington, and K. Hawick. Analysis and Development of Java Grande Benchmarks. *Proceedings of the ACM Java Grande Conference, San Francisco, CA, June*, 1999.

[19] Microsoft Corporation. The Microsoft .NET platform, 2002. `http://www.microsoft.com/net/`.

[20] D. O'Donoghue and J. Power. Identifying and evaluating a generic set of superinstructions for embedded Java programs. *Proceedings of the International Conference on Embedded Systems and Applications*, pages 192–198, 2004.

[21] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[22] B. Stroustrup. *The Evolution of C++: 1985-1989.* MIT Press, Cambridge, MA, USA, 1993.

[23] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, et al. Evolution of a Java just-in-time compiler for IA-32 platforms. *IBM Journal of Research and Development*, 48(5/6):767–795, 2004.

[24] Sun Microsystems. The Java Hotspot Virtual Machine v1.4.1, Sept. 2002.

[25] A. Taivalsaari. Implementing a Java TM Virtual Machine in the Java Programming Language. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1998.

[26] J. Whaley. Joeq: A virtual machine and compiler infrastructure. *Science of Computer Programming*, 57(3):339–356, 2005.

[27] T. Wilkinson. Kaffe–a Java virtual machine. `http://www.kaffe.org/`, October 2006.

[28] H. Yamauchi and M. Wolczko. Writing Solaris device drivers in Java. In *PLOS '06: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*, page 3, New York, NY, USA, 2006. ACM Press.