USENIX Association

Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium (JVM '02)

San Francisco, California, USA
August 1-2, 2002

A Modular and Extensible JVM Infrastructure*

Patrick Doyle and Tarek S. Abdelrahman
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{doylep,tsa}@eecg.toronto.edu

Abstract

This paper describes the design, implementation, and experimental evaluation of a modular and extensible Java® Virtual Machine (JVM) infrastructure, called Jupiter. The infrastructure is intended to serve as a vehicle for our research on scalable JVM architectures for a 128-processor cluster of PC workstations, with support for shared memory in software. Jupiter is constructed, using a building block architecture, out of many modules with small, simple interfaces. This flexible structure, similar to UNIX® shells that build complex command pipelines out of discrete programs, allows the rapid prototyping of our research ideas by confining changes in JVM design to a small number of modules. In spite of this flexibility, Jupiter delivers good performance. Experimental evaluation of the current implementation of Jupiter using the SPECjvm98 benchmarks shows that it is on average 2.65 times faster than Kaffe and 2.20 slower than the Sun Microsystems JDK (interpreter versions only). By providing a flexible JVM infrastructure that delivers competitive performance, we believe we have developed a framework that supports further research into JVM scalability.

1 Introduction

The use of the Java® programming language has been steadily increasing over the past few years. In spite of its popularity, the use of Java remains limited in high-performance computing, mainly because of its execution model. Java programs are compiled into portable stack-based *bytecode* instructions, which are then interpreted by a run-time system referred to as the Java Virtual Machine (JVM). The limited ability of a Java compiler to optimize stack-based code and the overhead resulting from interpretation lead to poor performance of Java programs compared to their C or C++ counterparts.

Consequently, there has been considerable research aimed at improving the performance of Java programs. Examples include: just-in-time (JIT) compilation [1, 2], improved array and complex number support [3, 4], efficient garbage collection [5, 6], and efficient support for threads and synchronization [1].

The majority of this research has focused on improving performance on either uniprocessors or small-scale SMPs. Our long-term research addresses scalability issues of the JVM for large numbers of processors. In particular, our goal is to design and implement a JVM that scales well on our 128-processor cluster of PC workstations, interconnected by a Myrinet network, and with shared memory support in software. However, in order to carry out this research, we require a JVM infrastructure that allows us to rapidly explore design and implementation options. While there exist a number of JVM frameworks that we could use [1, 7, 8, 9], these frameworks provide limited extensibility and are hard to modify. Hence, we embarked on the design and implementation of a modular and extensible JVM, called Jupiter. It uses a building block architecture which enhances the ability of developers to modify or replace discrete parts of the system in order to experiment with new ideas. Further, to the extent feasible, Jupiter maintains a separation between orthogonal modifications, so that the contributions of independent researchers can be combined with a minimum of effort. In spite of this flexibility, Jupiter supports simple and efficient interfaces among modules, hence preserving performance. In this paper, we

*This work was supported by research grants from NSERC and CITO. Patrick Doyle was supported in part by a scholarship from Sun Microsystems.

focus on this Jupiter infrastructure. In particular, we describe the overall architecture, various implementation aspects, and performance evaluation of Jupiter.

The current implementation of Jupiter is a working JVM that provides the basic facilities required to execute Java programs. It has an interpreter with multithreading capabilities. It gives Java programs access to the Java standard class libraries via a customized version of the GNU Classpath library [10], and is capable of invoking native code through the Java Native Interface [11]. It provides memory allocation and collection using the Boehm garbage collector [12]. On the other hand, it currently has no bytecode verifier, no JIT compiler, and no support for class loaders written in Java, though the design allows for all these things to be added in a straightforward manner. The performance of Jupiter's interpreter makes it comparable to commercial and research interpreters, while still maintaining a high degree of flexibility.

The remainder of this paper is organized as follows. In Section 2 we give an overview of Jupiter's architecture. In Section 3 we present details of Jupiter's design and implementation. In Section 4 we present the results of our experimental evaluation of Jupiter. In Section 5 we give an overview of related work. Finally, in Section 6 we provide some concluding remarks.

2 System Architecture

The overall structure of Jupiter is depicted in Figure 1. In the center is the `ExecutionEngine`, the control center of the JVM, which decodes the Java program's instructions and determines how to manipulate resources to implement those instructions. The resources themselves are shown as ovals, and include Java classes, fields, methods, attributes, objects, monitors, threads, stacks and stack frames (not all of which are shown in the diagram).

The responsibility for managing each resource is delegated by the `ExecutionEngine` to a particular *Source* class, each shown as a rectangle within the pie slice that surrounds the resource it manages. The Sources insulate the `ExecutionEngine` from the details of how resources are managed. Sources share a simple, uniform interface: every Source class has one or more `get` methods which return an instance of the appropriate resource. Each `get` method has arguments specifying any information needed by the Source to choose or allocate that resource, and the Source is responsible for deciding how the resource should be created, reused, or recycled.

An incarnation of Jupiter, then, is constructed by assembling a number of Source objects in such a way as to achieve the desired JVM characteristics, a scheme referred to as a *building-block architecture* [13]. As each Source is instantiated, its constructor takes references to other Sources it needs in order to function. For instance, as shown in Figure 1, the `ObjectSource` makes use of a `MemorySource`, so the `ObjectSource` constructor would be passed a reference to the particular `MemorySource` object to which it should be connected. The particular Source objects chosen, and the manner in which they are interconnected, determines the behaviour of the system.

The assembly of a JVM out of Jupiter's Source objects is much like the manner in which UNIX command pipelines allow complex commands to be constructed from discrete programs: each program is "instantiated" into a process, and each process is connected to other processes, via pipes, as described by the command syntax; once the process-and-pipe structure has been assembled, data begins to flow through the structure, and the resulting behaviour is determined by the particular choice of programs and their interconnections. Likewise, an incarnation of Jupiter is first constructed by instantiating and assembling Sources. Once the JVM is assembled, the Java program begins to flow through it, like data through the command pipeline. The behaviour of the JVM is determined by the choice of Source objects and their interconnections.

Figure 2 shows part of a typical running incarnation of Jupiter, consisting of interconnected Source objects through which the resources of the executing Java program flow. Also depicted is a typical collection of resource objects. In particular, the `Context` object represents the call stack for an executing Java program. To begin execution, a `Context` is constructed and passed to an `ExecutionEngine`, which sets the rest of the JVM in motion to interpret the program. From the `Context` object, the `MethodBody` object can be reached, which possesses the Java instructions themselves. By interpreting these instructions and manipulating the appropri-

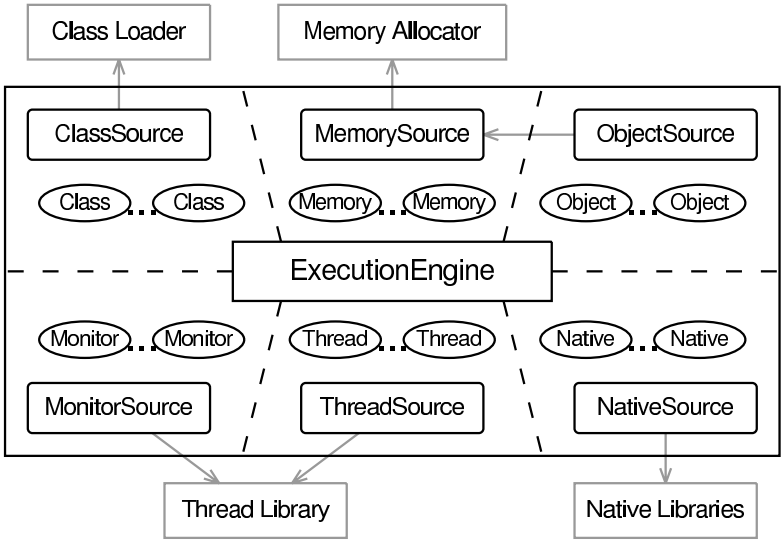


Figure 1: Jupiter's conceptual structure. Resource management responsibility is divided and delegated to a number of Source modules, leaving the `ExecutionEngine` with an "abstract," simplified view of the system's resources.

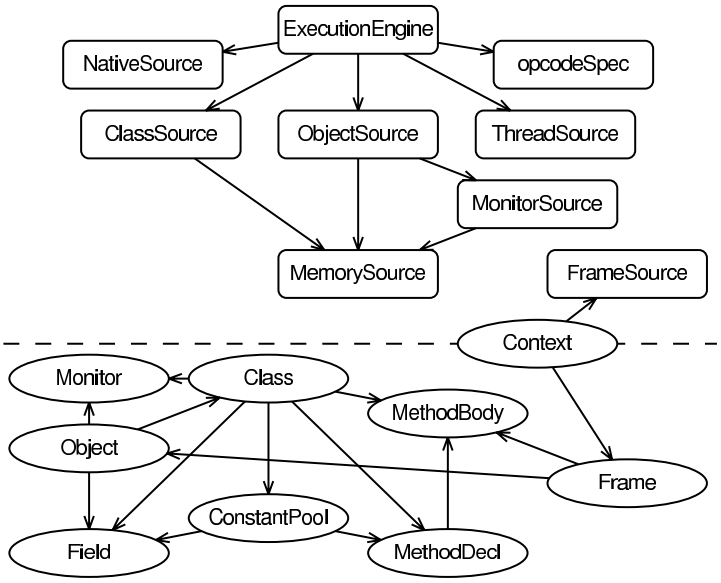


Figure 2: A typical building-block structure. Sources are shown above the dashed line, and resources below. The `Context` object acts as a link between the two when the Java program first begins to execute.

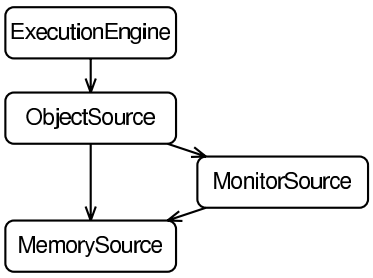


Figure 3: A simple object allocation building-block structure.

ate sources and resources in the appropriate way, Jupiter is able to perform the indicated operations, thereby executing the Java program.

2.1 System Flexibility

In this section, we demonstrate Jupiter's flexibility by examining several configurations of the system's building-block modules. We focus on a recurring example—the object creation subsystem. Through examples, we present several hypothetical ways in which Jupiter could be modified to exploit memory locality on a non-uniform memory access (NUMA) multiprocessor system. In such a system, accessing local memory is less time-consuming than accessing remote memory. Hence, it is desirable to take advantage of local memory whenever possible.

Object creation begins with `ObjectSource`, whose `getObject` method takes a `Class` to instantiate, and returns a new instance of that class. At the implementation level, Java objects are composed of two resources: memory to store field data, and a monitor to synchronize accesses to this data. In order to allocate the memory and monitor for a new `Object`, the `ObjectSource` uses a `MemorySource` and a `MonitorSource`, respectively. The `MemorySource` may be as simple as a call to a garbage collected allocator such as the Boehm conservative collector [12]. Typically, the `MonitorSource` uses that same `MemorySource` to allocate a small amount of memory for the monitor.

The objects employed by such a simple scheme are shown in Figure 3, where arrows indicate the *uses* relation between the modules. The `ExecutionEngine` at the top is responsible for executing the bytecode instructions, and calls upon various facility classes, of which only `ObjectSource` is shown. The remainder of this section will explore the system modifications that can be implemented by reconfiguring the building blocks of this archetypal object allocation scheme.

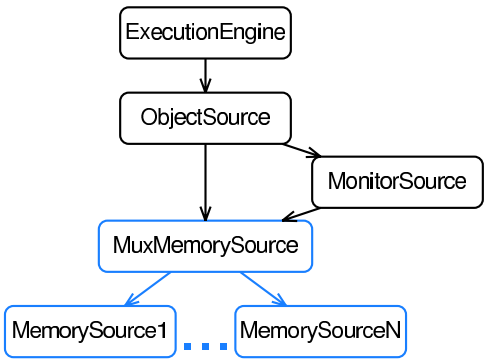


Figure 4: Locality decisions made at the `MemorySource` level.

Suppose the memory allocator on a NUMA system takes a node number as an argument and allocates memory in the physical memory module associated with that node:

```
void *nodeAlloc(int nodeNumber, int size);
```

We can make use of this interface, even though our `getMemory` function of the `MemorySource` facility does not directly utilize a `nodeNumber` argument. We do so by having one `MemorySource` object for each node in the system. We then choose the node on which to allocate an object by calling upon that node's `MemorySource`.

There are a number of ways the `ExecutionEngine` can make use of these multiple `MemorySources`. One way would be to use a "facade" `MuxMemorySource` module that chooses which subordinate node-specific `MemorySource` to use, in effect multiplexing several MemorySources into one interface. This is shown in Figure 4. `MuxMemorySource` uses appropriate heuristics (such as first-hit or round-robin) to delegate the request to the appropriate subordinate `MemorySource`. The advantage of such a configuration is that it hides the locality decisions inside `MuxMemorySource`, allowing the rest of the system to be used without any modification.

A second possibility is to manage locality at the `ObjectSource` level on a per-object basis, as shown in Figure 5. `MuxObjectSource` is similar to `MuxMemorySource`, in that it uses some heuristic to

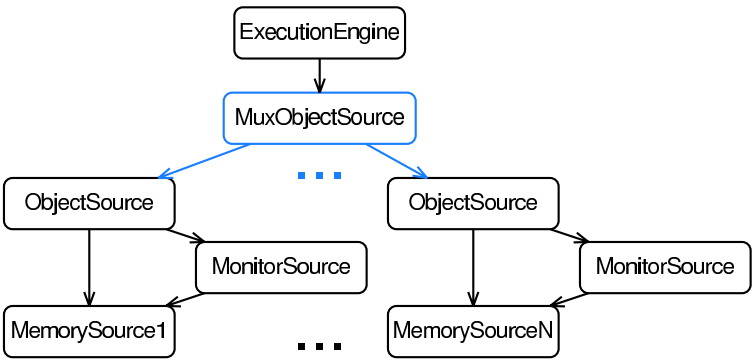


Figure 5: Locality decisions made at the `ObjectSource` level.

determine the node on which to allocate an object. We can use the same node-specific `MemorySource` code as in the previous configuration from Figure 4. We can also use the same `ObjectSource` and `MonitorSource` classes as in the original configuration (Figure 3); we simply use multiple instances of each one. Very little code needs to change in order to implement this configuration.

Yet a third possibility is to allow the `ExecutionEngine` itself to determine the location of the object to be created. Since the `ExecutionEngine` has a great deal of information about the Java program being executed, it is likely to be in a position to make good locality decisions, on a per-thread basis. In this configuration, shown in Figure 6, the `ObjectSource` and `MemorySource` remain the same as in the original configuration. The execution engine chooses where to allocate its objects by calling the appropriate `ObjectSource`. Again, we have not changed `ObjectSource` or `MonitorSource` classes, and the node-specific `MemorySource` class is the same one from the previous configurations.

These examples demonstrate the flexibility of Jupiter's building-block architecture. Each scheme has advantages and disadvantages, and it is not clear which is best. However, the ease with which they can be incorporated allows researchers to implement and compare them with minimal effort.

2.2 Performance Considerations

At first glance, it would appear that our flexible building block structure impairs the performance of the JVM. A researcher who was not concerned with flexibility could simply hard-code the `ObjectSource` to call the `nodeAlloc` function directly. In contrast, our system appears to have two efficiency problems:

- *Call overhead.* Each object allocation request must pass through a number of modules, with each call contributing overhead.

- *Object proliferation.* One node-specific `MemorySource` is required for each node; hence, with hundreds of nodes, hundreds of `MemorySources` will be needed, which would be unnecessary if `ObjectSource` were to call `nodeAlloc` directly.

For a researcher interested in performance, it would be tempting to bypass the module structure entirely, thereby degrading the extensibility of the system.

However, careful exploitation of compiler optimizations allows Jupiter to achieve the performance of the less flexible scheme, without sacrificing flexibility. The reason that this is possible is that each node-specific `MemorySource` is associated with a particular node for the duration of its lifetime, making the node number for each `MemorySource` *immutable*. Immutable data can be freely duplicated without concern for consistency among the multiple copies, since the data never changes. As a result, immutable data that is normally passed by reference can instead be passed by value, with no change to the system's behaviour. This removes the need to dereference pointers, and also eliminates the alias analysis difficulties that make pointer-based code hard to optimize. The system can continue to use the usual abstract, high-level interfaces, and the compiler can produce highly efficient code.

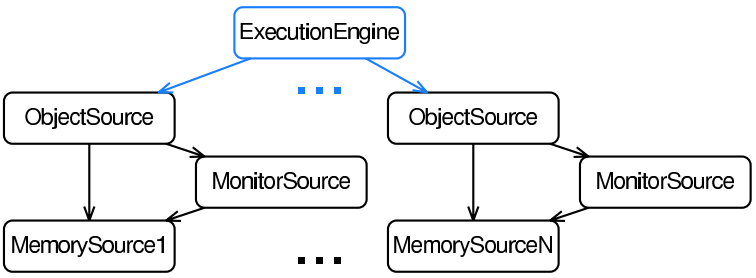


Figure 6: Locality decisions made by the `ExecutionEngine` itself.

To illustrate how this is achieved in our example of object allocation, we begin with the standard Jupiter `MemorySource` interface declarations:

```
typedef struct ms_struct *MemorySource;
void *ms_getMemory(MemorySource this,
                   int size);
```

Because the node number for each `MemorySource` is immutable, it can be passed by value. This can be implemented by replacing the standard declarations with the following:

```
typedef int MemorySource;

static inline void
*ms_getMemory(MemorySource this,
              int size){
  if(this == MS_MUX)
    return nodeAlloc(/* The appropriate
                        node */, size);
  else
    return nodeAlloc(this, size);
}
```

In this version, a `MemorySource` is no longer a pointer to a traditional "heavyweight" object; instead, it is simply an integer representing the node number itself. The `MuxMemorySource` is represented by the special non-existent node number `MS_MUX`. To allocate memory, this code first checks whether the `MuxMemorySource` is being used. If so, it uses the desired locality heuristic to choose a node; otherwise, if a particular node-specific `MemorySource` is used, then memory is allocated from the corresponding node.

With these definitions in place, the existing abstract high-level function calls can be transformed by the compiler into efficient code. Beginning with this:

```
void *ptr = ms_getMemory(
                obs_memorySource(), size);
```

The compiler can perform a succession of function inlining optimizations to produce this:

```
void *ptr = nodeAlloc(/* The appropriate
                         node */, size);
```

Hence, there is no longer any performance penalty for using Jupiter's `MemorySource` interface. This example demonstrates how careful design and implementation allows Jupiter to achieve good performance without any cost to flexibility.

3 System Components and Implementation

In this section, we give a brief tour of the modules which constitute the Jupiter system. The functionality of these modules is exposed through a number of interfaces, known as the *base* interfaces, which are fundamental to Jupiter's design. For each facility provided by Jupiter, we first present its base interfaces, and describe it in terms of the responsibilities it encapsulates. We then describe the current implementation of that facility. It is important to note that these are examples showing the current implementation of the Jupiter facilities. The design of Jupiter allows these implementations to be changed easily.

3.1 Memory Allocation

The `MemorySource` base interface encapsulates the memory allocation facility. It provides just one func-

tion, called "getMemory," which takes the size of the memory block required, and returns the resulting block. The current implementation has seven MemorySources, which can be used alone, or in combination, to produce a wide variety of effects:

- MallocMemorySource calls the standard C malloc function to allocate memory. This was useful early in development of the system.
- BoehmMemorySource calls the Boehm conservative garbage collector [12].
- BoehmAtomicMemorySource also calls the Boehm collector, but it marks memory chunks as being pointer-free (that is, *atomic*). This is useful to prevent the garbage collector from unnecessarily scanning for pointers within large arrays of non-pointer data.
- ArenaMemorySource doles out chunks of memory from a given contiguous block, called an *arena*.
- MemoryCounter keeps track of which parts of Jupiter have allocated the most memory. This is useful in debugging to help reduce memory usage.
- Tracer annotates each memory block with information that allows for memory profiling. This is useful to diagnose cases when memory is not being garbage-collected properly, to find out why memory is being retained.
- ErrorMemorySource reports an allocation error when called from specified points within the Jupiter source code. This is useful for testing Jupiter's error handlers, by injecting errors into Jupiter that are otherwise difficult to reproduce.

3.2 Metadata and Method Dispatch

Jupiter creates metadata resource objects to represent the Java program itself. These objects take the form of Classes, Fields, MethodDecls, and MethodBodies. Jupiter accesses classes by name through the ClassSource interface, using a function called getClass. Once a Class has been acquired, its Fields, MethodDecls and MethodBodies can be accessed in order to perform the operations required by the running Java program. A Field encapsulates the data required to locate a field within an object. Typically, it contains nothing more than the field's offset. A MethodDecl encapsulates the data stored in the constant pool to represent METHODREF and INTERFACEMETHODREF entries. A MethodBody encapsulates the data that represents a method implementation; for non-native methods, it holds the bytecode instructions.

Method dispatch is modeled as a mapping from a MethodDecl to a MethodBody. The data structures involved in this mapping are shown in Figure 7. First, the ConstantPool of the target object's Class is consulted to acquire a MethodDecl object. If the MethodDecl has not yet been resolved, the ConstantPool calls upon the Class to locate the MethodDecl, for which it uses a hash table keyed by method name and type signature. As with most JVMs, once the MethodDecl has been acquired, it is cached by the ConstantPool to accelerate subsequent accesses.

Having acquired the MethodDecl, it must now be dispatched to a particular MethodBody. This is done using jump tables indexed by an offset stored in the MethodDecl. Interface methods use a two-level table scheme that allows them to be dispatched in constant time [14]. Once the MethodBody has been acquired, it can be executed, either by interpreting its bytecode, or, for native methods, by invoking the appropriate function through the Java Native Interface (JNI) [11].

3.3 Object Manipulation

Java objects are manipulated through three interfaces: Object, which encapsulates field layout and access; ObjectSource, which encapsulates object allocation and garbage collection; and Field, described above, which encapsulates the data required to locate a field within an object.

These interfaces are implemented as shown in Figure 8, which has similarities to the method dispatch structures shown earlier in Figure 7. The Field abstraction serves an analogous purpose to MethodDecl, representing a field reference in the constant pool. The Field is then passed to the Object, which returns the value of that field.

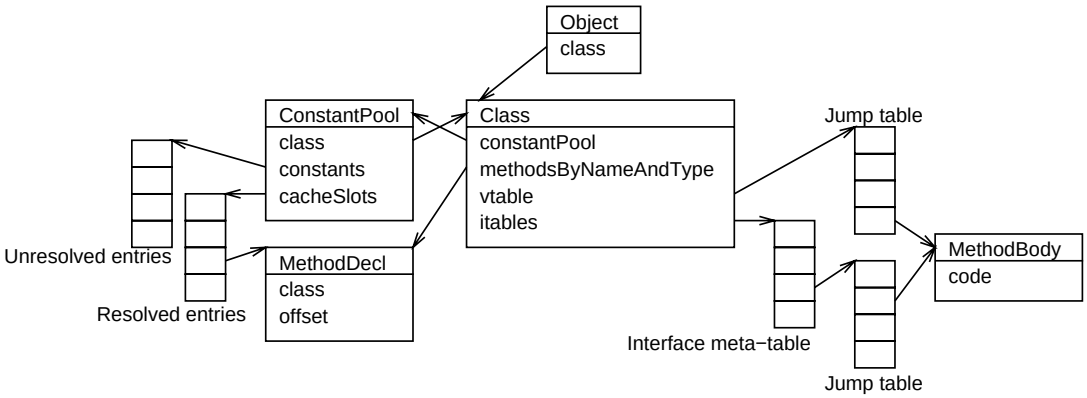


Figure 7: The method lookup objects. Lookup proceeds first as shown on the left to acquire a `MethodDecl`, then on the right to acquire a `MethodBody`.

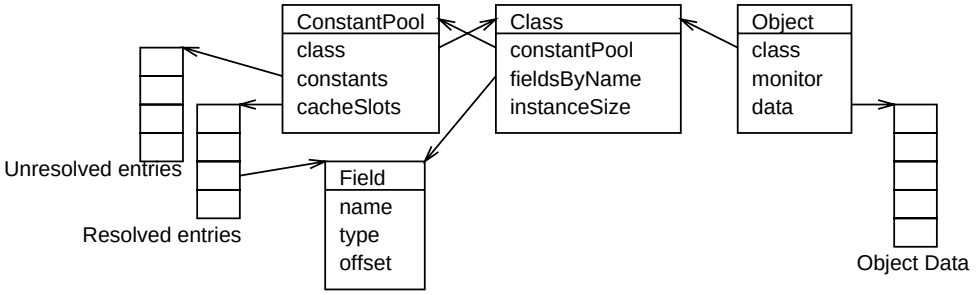


Figure 8: The Jupiter objects responsible for the layout of Java objects.

3.4 Java Call Stack

The call stack of the running Java program is modelled by three interfaces: `Frame`, which encapsulates the data stored in a single stack frame, such as the operand stack and local variables; `FrameSource`, which encapsulates the allocation and layout of `Frame`s, controlling such things as the argument-passing mechanism; and `Context`, which encapsulates the storage and management of the call stack as well as the locking logic required by synchronized methods.

Representing the Java context entirely as a data structure allows Java threads to be migrated, simply by executing a given `Context` on a different thread. This stands in contrast to the more straightforward scheme used in Kaffe, which implements method invocation by recursion within the execution engine [9], causing context information to be stored on the native stack, and precluding this kind of migration.

The objects representing the execution stack are shown in Figure 9. The system's view of the stack is provided by the `Context` object on the left, which encapsulates a large array of word-sized *slots* in which the stack contents are stored. Each slot is capable of holding up to 32 bits of data; 64-bit data types require two adjacent slots, as prescribed by the Java specification [15].

Individual stack frames are manipulated through the `Frame` interface, shown as a trapezoid in the figure. A number of design techniques provide the illusion that `Frame`s are just like any other objects [16], but in reality, the data for each `Frame` is stored in a contiguous group of slots within the slot array. This allows the frames to be overlapped, making method argument copying unnecessary, while preserving the object-oriented interface. The resulting layout of two adjacent stack frames is shown in the diagram, with overlapping frames labelled twice to indicate their role in each of the frames.

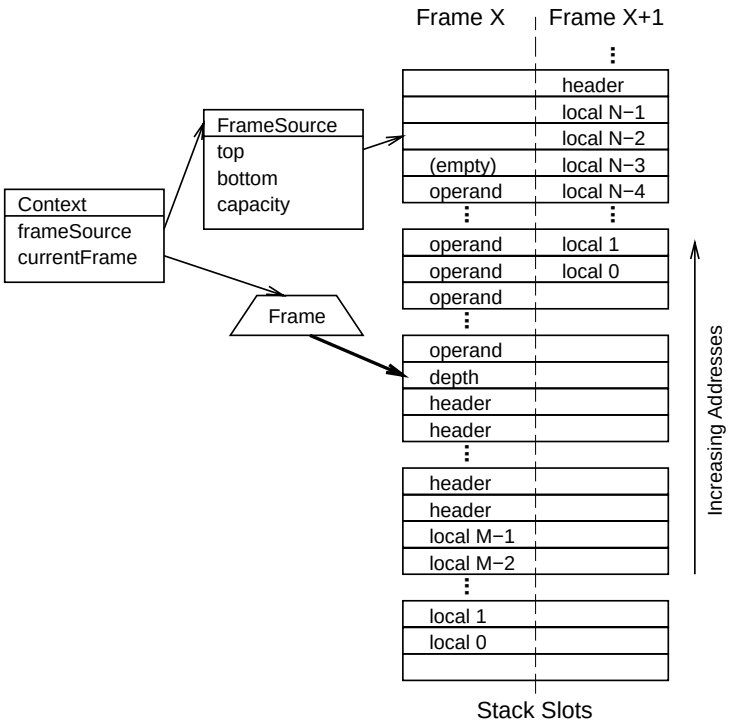


Figure 9: The stack layout. Each stack slot is labelled twice, for its role in the two overlapping frames. The slot marked "`(empty)`" is the portion of the operand stack space which does not currently contain data.

3.5 Bytecode Interpretation

The `ExecutionEngine` decodes the bytecode and performs the actions necessary to implement each instruction. Its interface is quite simple, consisting of a single function that takes a `Context` as an argument, and executes the method whose frame is on top of the `Context`'s call stack.

Since Jupiter's design delegates much of the execution responsibility to other parts of the system, not much remains to be done by the `ExecutionEngine` itself. The current interpreter implementation divides the functionality into three modules, which are shown along with the `ExecutionEngine` interface in Figure 10. These modules are each responsible for implementing a portion of the `ExecutionEngine` functionality:

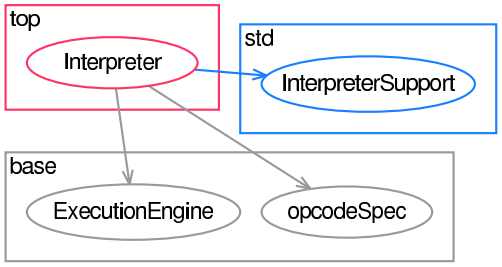


Figure 10: The bytecode execution modules.

- The `opcodeSpec` module defines each of the Java opcodes in terms of Jupiter's `base` interfaces. It takes the form of a header file that is included (with `#include`) into the interpreter module. It is designed to be used by any `ExecutionEngine`, be it an interpreter or a JIT compiler.
- The `InterpreterSupport` module provides functionality that is independent of the particular interpreter implementation, such as the stack-unwinding algorithm for exception handling.
- The `Interpreter` module implements the `ExecutionEngine` interface, making use of the `opcodeSpec` and `InterpreterSupport` modules as necessary.

The current `ExecutionEngine` implementation is a *threaded interpreter*, meaning that, after execut-

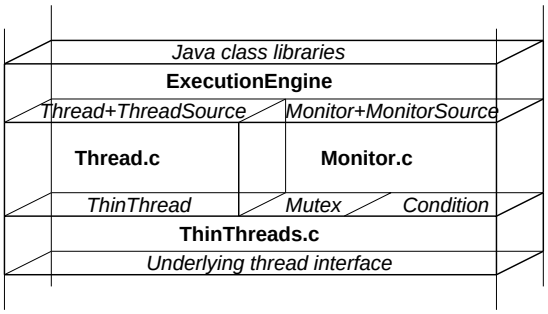


Figure 11: Multithreading modules and interfaces. Modules are shown as blocks divided by horizontal planes representing interfaces.

ing one opcode, it branches directly to the code for executing the next opcode [7]. This stands in contrast to the typical scheme, which uses a `switch` statement inside a loop to jump to the appropriate code. The threaded scheme eliminates the branch to the top of the loop, whose overhead can be substantial in an optimized interpreter like Jupiter's, as will be shown in Section 4.2. The current `ExecutionEngine` also does *bytecode substitution* to improve the performance of `getfield`, `putfield`, `invokevirtual` by dynamically replacing them with faster versions, as will be described in Section 4.2.

3.6 Threading and Synchronization

To maximize flexibility, Jupiter uses two levels of interfaces for threading and synchronization, shown in Figure 11. The high-level interfaces, called `Thread` and `Monitor` (plus the corresponding `Source`s), provide the full Java concurrency semantics. The low-level interfaces, called `ThinThread`, `Mutex`, and `Condition`, provide the minimal semantics required by Java. These low-level interfaces are referred to collectively as the `ThinThreads` interface, which provides a small subset of the POSIX threads semantics [17]. The high- and low-level interfaces are complimentary in several ways:

- `ThinThreads` encapsulates the thread library beneath Jupiter. `Thread` and `Monitor` encapsulate the threading needs of the Java program running on top of Jupiter.
- `ThinThreads` provides the minimal requirements to make implementing Java threads *possible*. `Thread` and `Monitor` provide the maximum support to make implementing Java threads *simple*.
- `ThinThreads` is designed so that the implementation code which connects to the underlying thread library can be trivial. `Thread` and `Monitor` are designed so that the client code which uses them to implement Java threads can be trivial.

Separating the Java concurrency semantics from the semantics of the underlying thread library makes threading and synchronization modules easier to implement and modify.

4 Experimental Evaluation

Jupiter is written in C using an object-oriented style. Although languages such as Java or C++ would provide more support for an object-oriented programming style, and hence more support for our flexible building block architecture, we elected to use C because we did not have confidence in the ability of other languages to deliver good performance. The current implementation comprises approximately 23,000 lines of C code, in about 170 files.

In this section, we quantify the performance Jupiter delivers on standard benchmarks. We also attempt to show the degree of flexibility Jupiter possesses by arguing the ease with which a number of performance optimizations were implemented.

4.1 Overall Performance

To test Jupiter's functionality and performance, we used it to run the single-threaded applications[1] from SPECjvm98 benchmark suite [18]. In this section, we present the execution times consumed by these benchmarks running on Jupiter, and compare them with results from Kaffe 1.0.6, and from the Sun Microsystems JDK v1.2.2-L. We find that Jupiter is faster than Kaffe and slower than JDK.

[1] Although multithreading is already functional in the version of Jupiter we use to report performance, multithreaded performance is currently being optimized. Hence, we elect to report only the performance of single-threaded applications.

	Benchmark	JDK	Jupiter	Kaffe	Jupiter/JDK	Kaffe/Jupiter
1	209_db	178s	282s	836s	1.59:1	2.96:1
2	228_jack	112s	213s	567s	1.91:1	2.66:1
3	201_compress	333s	700s	2314s	2.10:1	3.31:1
4	222_mpegaudio	276s	649s	1561s	2.35:1	2.40:1
5	213_javac	114s	313s	733s	2.74:1	2.35:1
6	202_jess	93s	257s	608s	2.76:1	2.36:1
				Geometric Mean	2.20:1	2.65:1

Table 1: Execution time, in seconds, of the benchmarks using each JVM. The ratios on the right compare the JVMs pairwise, showing the slower JVM's execution time relative to the faster one's.

Table 1 compares the execution times of each benchmark run on the three JVMs. All times were measured on a 533 MHz Pentium III with 512 MB of RAM running Linux, kernel version 2.2.19. Jupiter was compiled with `gcc` version 2.95.2 at optimization level `-O3`, with all source code combined into a single compilation unit to facilitate function inlining [16]. The times were reported by the UNIX "`time`" program, and therefore include all JVM initialization. All benchmarks were run with verification and JIT compilation disabled, since Jupiter does not yet possess either of these features. Averaged across all benchmarks (using the geometric mean), Jupiter was 2.20 times slower than JDK, and 2.65 times faster than Kaffe.

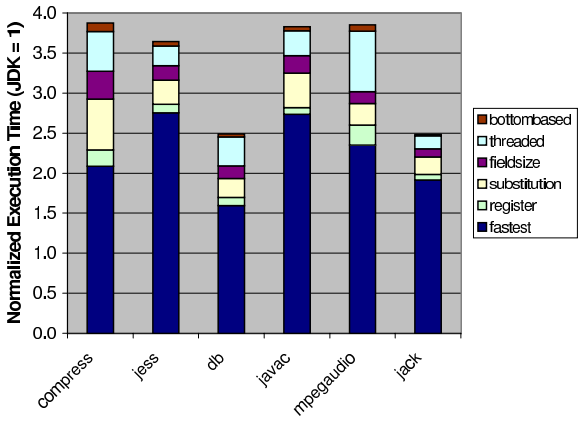


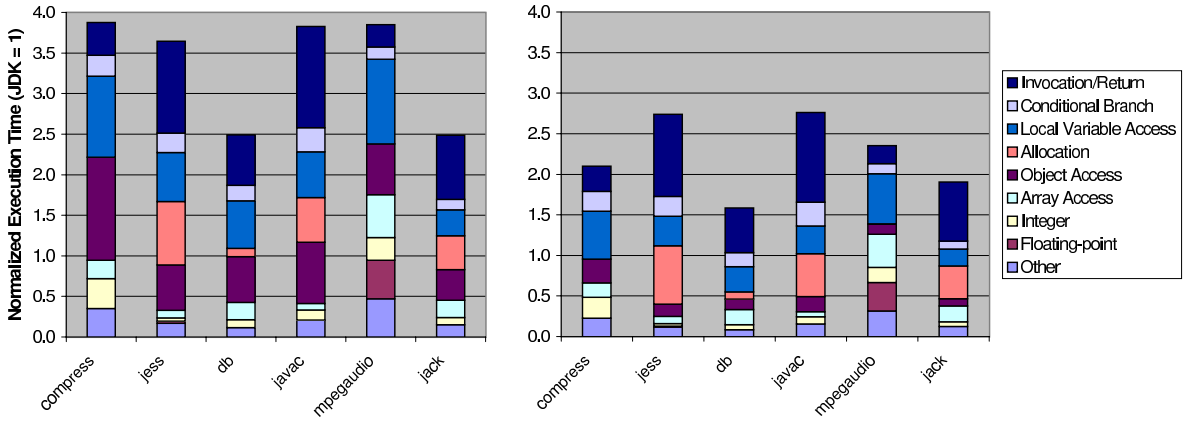
Figure 12: Effect of each optimization on benchmark execution times.

4.2 Performance Optimizations and Analysis

The current level of performance achieved by Jupiter requires that a number of optimizations be implemented. In this section, we briefly describe these optimizations and comment on how the flexible structure of Jupiter facilitated their implementation. The optimizations are:

- *bottombased*: the `Frame` interface was changed to use bottom-based operand stack indexing instead of top-based indexing to speed up stack accesses [16].
- *threaded*: the "loop-and-switch" interpreter was replaced by a threaded version to improve performance, as was described in Section 3.5.
- *fieldsize*: the field's size (either 4 or 8 bytes) was cached inside the `Field` pointer, relieving the interpreter from having to traverse data structures to find this information.
- *bytecode substitution*: the implementations of `getfield`, `putfield` and `invokevirtual` were changed so they replace themselves in the bytecode stream with respective quick versions the first time they execute. These quick versions assume that the field in the opcode has been resolved, and are specialized for the appropriate field size. Furthermore, the opcodes are re-written so that the field offset is stored directly in the bytecode stream, avoiding constant pool access. Subsequent executions of the same bytecode will find the quick instructions, which execute faster.
- *register*: a CPU register was assigned to store the location of the currently executing instruction, eliminating the need to load this value from memory in order to read and execute each instruction.

The impact of these optimizations is shown in Figure 12, which charts the execution-time reduction due to each optimization. The optimizations vary in the degree to which they benefit the performance of each application. For instance, the `fieldsize` and `substitution` optimizations, which targeted



(a) Before optimization (b) After optimization

Figure 13: Bytecode execution profile.

the `getfield` and `putfield` opcodes, were especially successful on `compress`, which spends a large portion of its time executing these two instructions. Also, `threaded` reduces the overhead of each opcode, producing a larger impact on applications that execute many "lightweight" opcodes, such as `mpegaudio`, compared to applications that execute relatively fewer, but more time-consuming, opcodes, such as `javac` and `jess`.

The above optimizations required minimal effort to implement in Jupiter, attesting to the flexibility of our system. For example, Jupiter's stronger encapsulation confines the modifications required to implement the `fieldsize` optimization to changing a half-dozen lines of code within the `Field` module. In contrast, Kaffe's equivalent of `opcodeSpec` explicitly tests the field type and calls one of a number of "`load_offset`" macros, passing only the field offset as a parameter. To take advantage of cached field size information, all implementations of the `load_offset` macros must be modified to pass the field size in addition to its offset—even those that are not affected by this optimization. Hence, in contrast to Jupiter, the lack of encapsulation within Kaffe causes the scope of this modification to encompass a large number of unrelated modules.

We believe that further improvements to the performance of Jupiter are still possible, which will bring its performance closer to that of the Sun Microsystems JDK. We profiled the amount of time consumed executing each kind of opcode, grouped into the following categories: *Invocation/Return*, *Conditional Branch*, *Local Variable Access Allocation*, *Object Access*, *Array Access*, *Integer*, *Floating-point*, and *Other*. The resulting execution profile is depicted in Figure 13, before and after the optimizations described above. The charts show that our optimization targeted mostly object access overhead, and that more performance-improving opportunities remain. For example, the two slowest benchmarks, `202_jess` and `213_javac`, have similar profiles, with large proportions of invocation/return and allocation opcodes. We will target these opcodes in future work.

Furthermore, Jupiter's coding style relies heavily on function inlining to achieve good performance, so a weakness in the compiler's inlining ability can have a substantial impact. For example, our examination of `gcc`-generated assembly code for `getfield` indicates that the code can be further optimized with nothing more than common subexpression elimination. The exact reason that `gcc` did not successfully perform this optimization is hard to determine. However, it appears that the implementation of `getfield` is too stressful on the inlining facility of `gcc`, hindering its ability to apply the optimization. Applying the optimization manually in the assembly code improves the performance of `getfield` by approximately 15%. The overall performance improvement due to the optimization depends on the application. For example, object access accounts for only 11% of `201_compress`, which would lead to less than 2% overall improvement, and even less for other applications[2]. Similar improvements are possible for other opcodes, and the accumulation of the individual improvements may be substantial.

[2]The small magnitudes of such improvements make them difficult to experimentally measure because they are within measurement error.

5 Related Work

There has been considerable work on improving performance of Java programs in uniprocessor environments [1, 2, 3, 4]. For example, Alpern et al. describe the design and implementation of the Jalapeño virtual machine [1], which incorporates a number of novel optimizations in its JIT compiler. Artigas et al. [3] investigate compiler and run-time support for arrays in Java, and show that improvements can be attained by eliminating run-time checks. Much of this work is orthogonal to ours, in that it improves uniprocessor performance. However, such improvements carry over to multiprocessors, and we expect them to be easily integrated into the Jupiter framework.

There are a number of JVMs and JVM frameworks designed for research into JVM design. They include the Sun Microsystems JDK [8], Kaffe [9], the IBM Jalapeño JVM [1], Joeq [19], OpenJIT [20], and SableVM [7]. However, these frameworks often address flexibility in particular dimensions of JVM design, while in contrast, Jupiter's flexibility is intended to be pervasive and fine-grained, allowing straightforward modification of any aspect of the system. For example, OpenJIT is an object-oriented framework for experimenting with JIT compiler designs and is implemented as a JIT-compiler plug-in to Sun's JDK, making it limited to JIT compiler research. Similarly, while Jalapeño (recently released as the Jikes RVM [21]) was designed as "flexible test bed where novel virtual machine ideas can be explored, measured, and evaluated"[1] in an industrial-grade server JVM, much of the work surrounding it explored JIT design. Though its object-oriented design undoubtedly possesses a large degree of inherent flexibility, it is unclear the extent to which the system is flexible in some aspects, such as object layout, stack layout, method dispatch, and so on.

6 Concluding Remarks

In this paper, we presented the design of a modular, flexible framework intended to facilitate research into JVM scalability. We described the building-block architecture employed, as well as the design and implementation of the key modules. Experimentation with our framework demonstrates that Jupiter's flexibility has facilitated a number of modifications, some of which are difficult to accomplish using Kaffe. Measurement of the execution time of the single-threaded SPECjvm98 benchmarks has shown that Jupiter's interpreter is, on average, 2.65 times faster than Kaffe, and 2.20 times slower than Sun's JDK. By providing a flexible JVM framework that delivers good performance, we hope to facilitate our, and others', research into JVM scalability.

Our future work on the Jupiter infrastructure will focus on three main aspects of its implementation. First, we will incorporate a trace-based JIT compiler, called RedSpot, whose implementation is currently underway. Second, we will extend the memory allocation interface in Jupiter to enable the use of a precise garbage collector, and to facilitate the implementation of parallel and concurrent garbage collection. Finally, we plan to modify the manner in which Jupiter stores metadata. At present, Jupiter spreads the responsibility for storing metadata throughout the system, leading to rather heavyweight objects. We believe that by storing the metadata separately, objects will be made lightweight, which would enable further performance optimizations.

References

[1] B. Alpern et al., "The Jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.

[2] K. Ishizaki et al., "Design, implementation and evaluation of optimizations in a just-in-time compiler," in *Java Grande*, pp. 119–128, 1999.

[3] P. Artigas, M. Gupta, S. Midkiff, and J. Moreira, "High performance numerical computing in Java: Language and compiler issues," in *Proc. of LCPC*, pp. 1–17, 1999.

[4] P. Wu, S. Midkiff, J. Moreira, and M. Gupta, "Efficient support for complex numbers in Java," in *Proc. of Java Grande*, pp. 109–118, 1999.

[5] T. Domani et al., "Implementing an on-the-fly garbage collector for Java," in *Proc. of the Symp. on Memory Management*, pp. 155–165, 2000.

[6] D. Bacon, C. Attanasio, H. Lee, V. Rajan, and S. Smith, "Java without the coffee breaks: A

nonintrusive multiprocessor garbage collector," in *Proc. of PLDI*, pp. 92–103, 2001.

[7] E. Gagnon and L. Hendren, "SableVM: A research framework for the efficient execution of Java bytecode," in *Proc. of USENIX JVM'01*, pp. 27–39, 2001.

[8] Sun Microsystems, http://www.java.sun.com, 2002.

[9] T. Wilkinson, "Kaffe — a virtual machine to run Java code," http://www.kaffe.org, 2002.

[10] "GNU Classpath," http://www.gnu.org/software/classpath/classpath.html, 2002.

[11] "Java Native Interface," http://java.sun.com/j2se/1.3/docs/guide/jni/index.html, 2002.

[12] H. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[13] O. Krieger and M. Stumm, "HFS: A performance-oriented flexible file system based on building-block compositions," *ACM Trans. on Computer Systems*, vol. 15, no. 3, pp. 286–321, 1997.

[14] F. Siebert and A. Walter, "Deterministic execution of Java's primitive bytecode operations," in *Proc. of USENIX JVM'01*, pp. 141–152, 2001.

[15] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.

[16] P. Doyle, "Jupiter: a modular and extensible Java Virtual Machine framework," Master's thesis, University of Toronto, 2002.

[17] IEEE/ANSI, "Posix threads extensions. IEEE/ANSI 1003.1c-1995," 1995.

[18] "SPECjvm98," http://www.specbench.org/osg/jvm98, 2002.

[19] J. Whaley, "joeq virtual machine," http://joeq.sourceforge.net/index.htm, 2002.

[20] F. Maruyama, "OpenJIT 2: The design and implementation of application framework for JIT compilers," in *Work-in-Progress Session of USENIX JVM'01*, 2001.

[21] "The Jikes research virtual machine (RVM)," http://www-124.ibm.com/developerworks/oss/jikesrvm/, 2002.

Trademarks