

# Jupiter: A Modular and Extensible JVM

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

## Abstract

This paper describes our work-in-progress on the design and implementation of Jupiter: a modular and extensible Java Virtual Machine (JVM) infrastructure. Jupiter serves as a vehicle for our research on JVM architectures that deliver scalable high performance for scientific applications on large numbers of processors. Our goal is to run Jupiter on our 128-processor cluster of PC workstations that supports shared memory in software. Jupiter is constructed out of many discrete modules with small, simple interfaces, much like Unix shells build complex command pipelines out of discrete programs. This structure allows rapid prototyping of our research ideas by confining changes in JVM design to a small number of modules. The structure is also efficient despite its flexibility, resulting in no loss in performance. We describe the basic architecture of Jupiter and give an example of how its components may be used to compose object creation subsystems with different locality policies.

## 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 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. Large-scale parallel computing is a viable means of delivering high-performance, but there appears to be little or no research that examines the performance of the JVM for large numbers of processors. Existing research and production JVMs are designed for small-scale SMPs (8 or 16 processors). It remains unclear how these JVMs will perform well in a large-scale parallel environment.

Our research addresses scalability issues of the JVM. 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. This paper reports our initial work on the design and implementation of an infrastructure JVM in support of this research.

We believe that JVM scalability can be achieved by examining four aspects of its design:

1. *Memory locality.* At present, objects are allocated on the heap with little or no consideration for locality. While this approach may be appropriate for uniprocessors or small-scale SMPs, it is unlikely to work well on a cluster of workstations where remote memory access is one or two orders of magnitude slower than local memory access. Hence, one of our goals is to develop allocation heuristics for enhancing locality.
2. *Parallel garbage collection.* Garbage collection can consume a considerable amount of application time. Typically, JVMs employ “stop-the-world” garbage collectors, where program threads are halted during garbage collection [5]. This approach will not work for large numbers of processors, for two reasons. First, the cost of “stopping the world” is considerably higher when the number of processors is large. Second, using a single thread to collect garbage results in an unacceptably large sequential fraction for any application. Consequently, we are developing a multi-threaded “on-the-fly” garbage collector that scales well to large numbers of processors.
3. *Memory consistency model.* To achieve scaling performance on a large number of processors, it is important to exploit the “relaxed” Java Memory Model [7]. Presently no JVM implements the JMM faithfully, and indeed many implement it incorrectly, leading to lack of coherence and loss of optimization opportunities [8]. The specification of the JMM is presently under revision. We will investigate the use of this revised model within a JVM and determine impact on performance.
4. *Efficient Threads and Synchronization.* With a large number of processors, it is critical to provide efficient threading support and synchronization mechanisms that scale well. We are examining means of providing such support.

In order to carry out our research, we require a JVM infrastructure that allows us to explore design and implementation options. There exist a number of JVM frameworks that we could use [1, 9, 10, 11]. 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. Jupiter implements design patterns which enhance 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 focus on the Jupiter framework and how it may be used to explore some of the research issues described above.

The remainder of this paper is organized as follows. In Section 2 we give an overview of Jupiter and its design. In Section 3 we demonstrate Jupiter’s flexibility and efficiency by presenting several configurations of the object creation subsystem. In Section 4 we describe how a JIT may be incorporated into Jupiter. In Section 5 we give an overview of related work. Finally, in Section 6 we provide some concluding remarks.

## 2 Basic Design and Implementation

In this section, we give an overview of Jupiter’s architecture and describe some of its implementation details.

## 2.1 Overview

The philosophy behind Jupiter is to construct a JVM out of many discrete units with small, simple interfaces, much like Unix shells build complex command pipelines out of discrete programs. By keeping the interfaces between the units small, we reap the benefits of information hiding. Furthermore, by careful design of the interfaces, we avoid the need to sacrifice performance in order to achieve modularity.

In designing Jupiter, we have strived to achieve three goals with respect to modularity. The first goal is *atomicity*: any anticipated modification to the system should not require modules to be split. Dividing one module into two, when the original module was not designed to be split, is not something that Jupiter developers should have to face as a normal part of working on the system. The second goal is *coherence*: modifications should require changes to very few modules; preferably, just one. Otherwise, it becomes difficult to determine the set of modules that need to be changed, and modifications end up being spread across many modules, increasing the coupling among them. The third goal is *independence*: unrelated modifications should affect separate modules. This allows the work of multiple researchers to be combined more easily.

The overall structure of Jupiter is shown in Figure 1. Components which are part of Jupiter are shown inside in the dashed box, and external resources are outside it. Jupiter is written in an object-oriented style, and a running incarnation of Jupiter is constructed from instances of the Jupiter classes<sup>1</sup>. The classes belong to two groups: those representing transient resources used by a running Java program, and the “facility” classes that manage the resources. The resources include Java classes, fields, methods, attributes, objects, locks, threads, stacks and stack frames (not all of which are shown in the diagram).

Most of the facility classes take the form of *Sources*, which share a simple, uniform interface: every **Source** class has a single **get** method which returns an instance of the appropriate resource. The arguments of any **get** method reflect the 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. For example, **MemorySource** has a method **getMemory** which takes a **size** argument indicating the quantity of memory to allocate, and it may reuse chunks of memory if a garbage collector can determine that it is safe to do so. On the other hand, **ClassSource** has a method **getClass** which returns a **Class**; it takes the class name as an argument, and always returns the same **Class** object for each unique name. The **ExecutionEngine** directs the execution of the system, making use of the other facility classes to manage resources.

## 2.2 Implementation

Jupiter has a collection of *Base classes*, which provide interfaces to the basic services that any JVM requires (such as object creation, method dispatch, and memory allocation). Of course developers are permitted (and encouraged) to define their own classes, but the **Base** classes represent the fundamental design of Jupiter, and are to be implemented in any incarnation of the system. Any module which is written so that it depends only on the **Base** classes becomes an independent entity which can be used orthogonally with other such modules.

Jupiter is implemented in C, though the same module structure could be used in other languages. We use a coding style which encourages independence of modules, and preserves the object-oriented design. Classes are defined entirely by their methods, and the data structures used to implement the classes are not declared in header files; instead, an opaque reference type is declared in the header, and each method is declared to take one such reference (the “this” pointer)

---

<sup>1</sup>By “classes” we don’t mean Java classes, nor instances of `java.lang.Class`, but rather the classes which constitute Jupiter’s own source code.

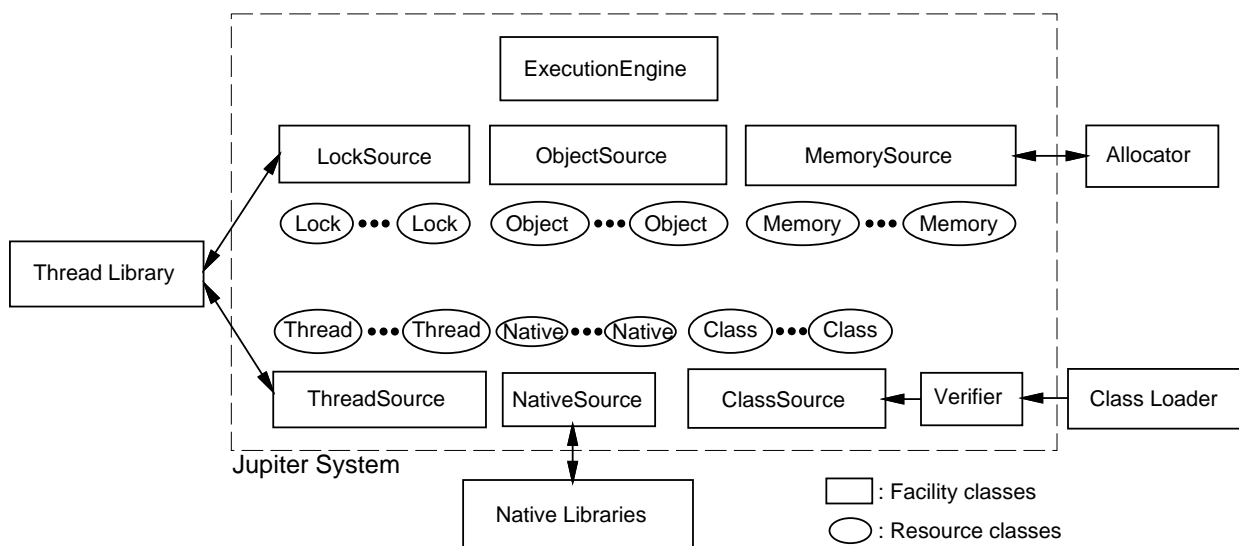


Figure 1: The high-level architecture of Jupiter.

as its first argument. Further, none of the **Base** headers declare any constructors: if construction of some class is important to model, then it has an associated **Source**; otherwise, construction is left as a matter of implementation. Thus, Jupiter’s class declarations are like Java’s interfaces: they provide method signatures, but no constructors, no method bodies, and no data representation.

This code structure may suggest a performance problem, since all JVM operations which cross module boundaries require function calls. However, a developer is always free to substitute his own version of a **Base** header—the include path used by Jupiter’s **make** system ensures that the developer’s own directory is searched before the **Base** directory—and so it can be made as efficient as necessary, using inline functions or even macros. So long as the new header is source-compatible with the existing one, it will remain inter-operable with the rest of the modules in the system.

A benefit of passing opaque reference types throughout the system is that these references are not required to be implemented as pointers. For instance, if a given object is small and immutable, it can be passed by value. Or, if the JVM were modified to span multiple memory spaces, a reference could be a handle to a remote object, and the handle itself could be passed by value. Requiring references to be pointers would make this sort of modification difficult, but passing opaque references by value makes it quite straightforward and efficient.

### 3 Object Creation — An Example of Flexibility

In this section, we demonstrate the system’s flexibility by examining several configurations of the object creation subsystem. Through examples, we present several ways in which Jupiter can be modified (which we refer to as “modes of extension”). In addition, we demonstrate how the module boundary overhead can be optimized away entirely when performance is important. Though we concentrate on object creation, the design philosophies presented here are ubiquitous in the Jupiter system, and the same level of flexibility can be found throughout.

#### 3.1 The Configurations

At the implementation level, Java objects are composed of two resources: memory to store field data, and a lock to synchronize accesses to this data. Objects are created by an **ObjectSource**,

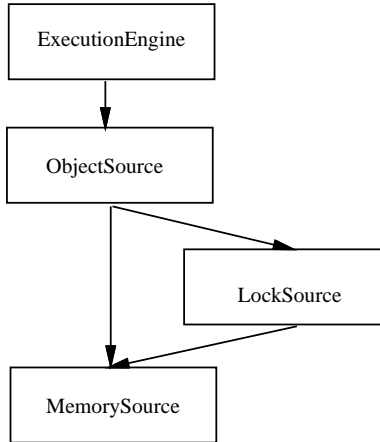


Figure 2: Relations between Jupiter objects in the simple scheme for object creation.

whose `getObject` method takes the `Class` to instantiate, and returns a new instance of that class. In order to allocate the memory and lock for this new instance, the `ObjectSource` uses a `MemorySource` and a `LockSource`, respectively.

The `MemorySource` may be as simple as a call to a garbage collector such as the Boehm conservative collector. The `LockSource` could use that same `MemorySource` to allocate a small amount of memory for the lock. The objects employed by such a simple scheme are shown in Figure 2, where arrows indicate the *uses* relation between objects. The `ExecutionEngine` at the top is responsible for executing the bytecode instructions, and calls upon various facility classes, of which only `ObjectSource` is shown.

Having established this basic structure, there are many modifications that can be explored. To begin with, suppose an alternative layout for the object fields is desired [9]. Such a modification simply substitutes a new `ObjectSource` which computes the proper size of the object based on the new layout<sup>2</sup>, and uses the same `MemorySource` and `LockSource`. Conversely, using a different garbage collector means substituting the `MemorySource` while using the same `ObjectSource` and `LockSource`. Hence, this structure allows different modifications to be used orthogonally: a new object layout can easily be used with or without a new garbage collector since each modification is confined to a single module.

This represents the simplest mode of extension to the Jupiter system: in-place substitution of a single module. A great many modifications can be implemented this way. It is the most desirable kind of extension, since it has the least impact on the rest of the system. However, there are more advanced modifications which cannot be implemented this way.

Consider the case in which Jupiter is to run on a multiprocessor system with non-uniform memory accesses (NUMA), such as our cluster of workstations which supports shared memory in software. In such a system, accesses to local memory are faster than accesses to remote memory. Hence, it is desirable to take advantage of local memory whenever possible.

The memory allocator on a NUMA system may take a node number as an argument and allocate 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 does not directly utilize a `nodeNumber` argument. We do so by having one `MemorySource` object for each node in

---

<sup>2</sup>Of course there is more to changing object layout than computing the new size, but this is not pertinent to our example.

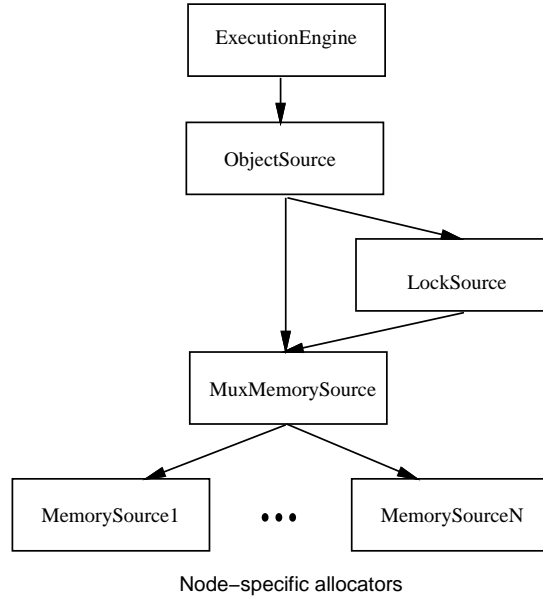


Figure 3: Node-specific memory sources. Locality decisions are made in **MuxMemorySource** invisibly to the rest of the system.

the system. We then choose the memory module to allocate an object in by calling upon its associated **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 multiple **MemorySources** into one interface. This is shown in Figure 3. **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 modifications.

A second possibility is to manage locality at the **ObjectSource** level, as shown in Figure 4. **MuxObjectSource** is similar to **MuxMemorySource**, in that it uses some heuristics to determine the memory module in which to allocate an object. We can use the same node-specific **MemorySource** code as in the previous configuration from Figure 3. We can also use the same **ObjectSource** and **LockSource** classes as in the original configuration (Figure 2); 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. In this configuration, shown in Figure 5, 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 **LockSource** classes, and the node-specific **MemorySource** class is the same one from the previous configurations.

The above configurations are examples of our second mode of extension, which involves changing the interconnection of the modules. In most cases, the modules need not be modified at all; only their relations with respect to other modules change.

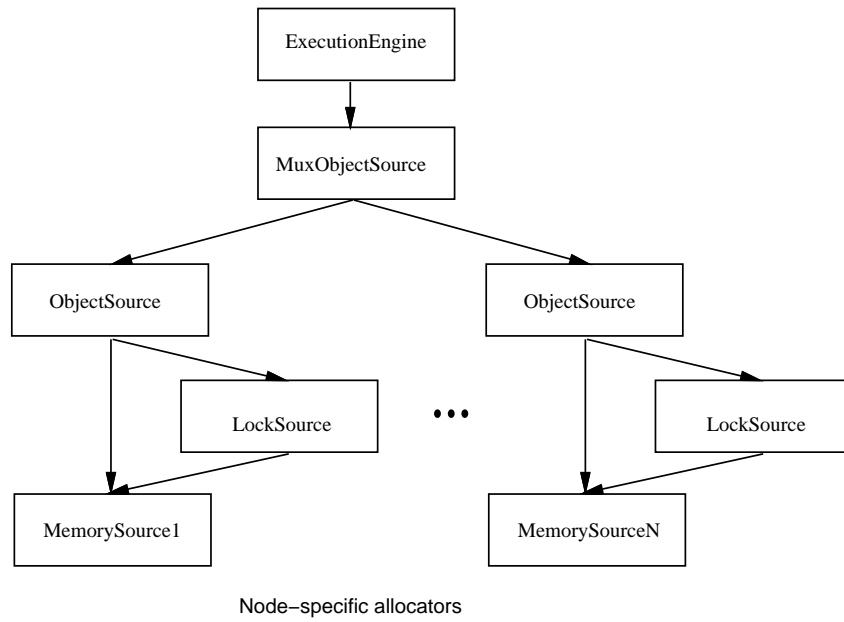


Figure 4: Node-specific object sources. Locality decisions are made in `MuxObjectSource`.

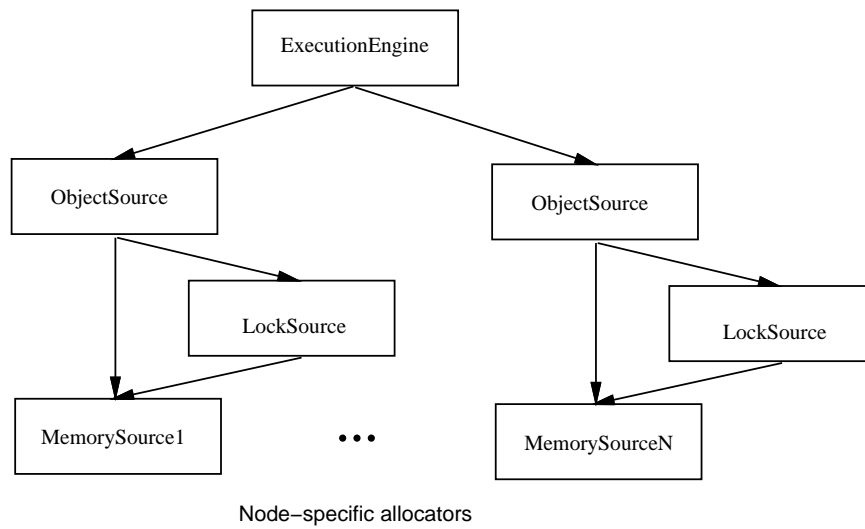


Figure 5: Locality decisions are made in the `ExecutionEngine` itself.

## 3.2 Performance

At first glance, it would appear that our flexible structure results in much object duplication within the JVM. Consider our final memory allocation configuration, illustrated in figure 5. This structure requires that `ObjectSources`, `LockSources`, and `MemorySources` be duplicated, one for each node. In a system with many nodes, this could amount to hundreds of small objects. A JVM that does not provide the flexibility or modularity of Jupiter would make calls to the `nodeAlloc` interface directly from the `ExecutionEngine` with no need to create and maintain additional objects. Hence, it would appear that our system incurs overhead to maintain these objects, resulting in extra CPU and memory usage, and poor cache locality. For a researcher interested in performance, it would be tempting to bypass the module structure entirely, thereby degrading the extensibility of the system.

However, this overhead can be eliminated by exploiting the *immutability* of the data contained in these objects. Immutable data can be freely replicated, unlike mutable data which needs careful coordination to ensure consistency among the various copies. In the present situation, because the data is immutable, it is possible to avoid creating and manipulating hundreds of small objects. Instead, they can be constructed on demand, perhaps on the stack or even in registers; passed around the system by value; and discarded when they are no longer needed. This allows us to regain a high level of performance while still enjoying the benefits of Jupiter's modular design.

Consider for example the Jupiter configuration with the locality management in `MuxMemorySource`, which was shown in Figure 3. For each `MemorySource`, the node number is fixed. The header file `MemorySource.h` defines the `MemorySource` class, using the following definitions:

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

However, a developer could provide her own implementation of this header file (thus *overriding* the `Base` implementation) by substituting the above definitions with:

```
typedef struct ms_struct {
    int nodeNumber;
} MemorySource;

static inline MemorySource ms_forNode(int nodeNumber) {
    /* Returns the MemorySource for the given node */
    MemorySource result = { nodeNumber };
    return result;
}
```

Since the node number of any given `MemorySource` is fixed, it can be passed by value. The result is that there is no need to create all the `MemorySources` ahead of time; instead, they are created as temporaries whenever they are needed. The effect is much like *currying*, whereby a function call with multiple arguments is transformed into a series of function calls with one argument. In our case, recall that memory allocation on our cluster system looks like this:

```
void *ptr = nodeAlloc(nodeNumber, size);
```

With currying, code in Jupiter can achieve the same semantics with a call like this:

```
void *ptr = ms_getMemory(ms_forNode(nodeNumber), size);
```

Notice that we have “sneaked” an extra parameter (the node number) into the `ms_getMemory` call by packing it into the `this` object. Any number of extra arguments could be passed this way. Such code has the advantage that it still conforms to Jupiter’s `Base` class interface for `MemorySource`, and so it can still be used by other parts of the system which are unaware of this scheme of memory allocation. Thus, the information hiding properties of the modules are preserved.

We cannot overlook the fact that our example in Figure 3 also uses a second type of `MemorySource`: the `MuxMemorySource`. To be treated as a true `MemorySource` by the rest of the system, the `MuxMemorySource` must use the interface defined in `MemorySource.h`. This could be achieved in our case by representing the `MuxMemorySource` by an invalid node number, say -1, and treating it specially using an `if` statement. It is reasonable to expect the `if` statement to be optimized away whenever the node number is known at compile time, which should almost always be the case.

If the C compiler cannot put structs in registers, making our `MemorySource` implementation too slow, we could go the final step and simply declare `MemorySource` to be an `int`. (We then lose some type safety, because C’s type system will not distinguish a `MemorySource` from any other integer, but we gain performance.) Our final `MemorySource.h` would look like this:

```
typedef int MemorySource;
static inline MemorySource ms_forNode(int nodeNumber){ return nodeNumber; }
static inline MemorySource ms_mux(){ return -1; } /* The MuxMemorySource */

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

At this level there is no performance penalty for using Jupiter’s `MemorySource` interface; the `MemorySources` are just integers, and as far as the compiler is concerned, the signature for `ms_getMemory` looks exactly like that of `nodeAlloc`. Further, as wildly different as they are, these definitions are source-code compatible with the original `Base` version of `MemorySource.h`: code recompiled with these new definitions will work as it always did. That we can produce an implementation which is source-code compatible with the existing module, yet which suffers no performance penalty from the module interface, demonstrates the remarkable flexibility of the Jupiter system.

## 4 JIT Compilation

It may appear that the information hiding inherent to Jupiter’s modular structure makes JIT compilation difficult, since the JIT compiler needs this information in order to generate fast code. For example, consider Jupiter’s `Frame` and `Object` classes. `Object` represents an instance of a Java class, and `Frame` represents a stack frame for an executing Java method. Below is a portion of the interfaces of these classes. They provide the interpreter with the ability to get and set object fields, and to access the operand stack.

```
Value ob_field(Object this, Field fd);
void ob_setField(Object this, Field fd, Value vl);

void fr_pushOperand(Frame this, Value vl, Type tp);
Value fr_popOperand(Frame this, Type tp);
```

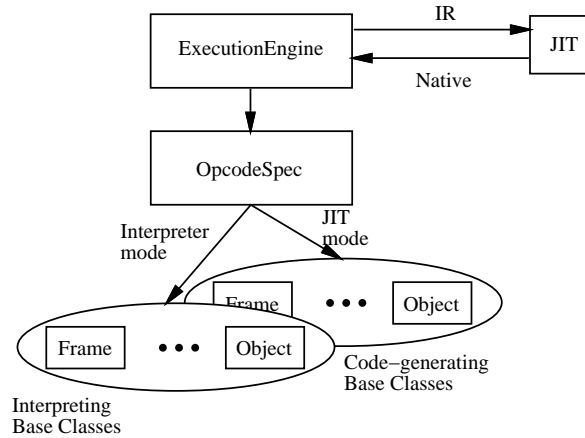


Figure 6: Jupiter's JIT compilation structure.

From a modularity point of view, these are good interfaces: they completely hide the object and stack layouts from the rest of the system. However, they make the JIT compiler's job difficult; the compiler does not have enough information to generate efficient data-access code. Of course, it would be unacceptable to generate code that uses these interfaces, when simple memory loads and stores would often suffice.

To circumvent this problem, the **ExecutionEngine** makes use of an **OpcodeSpec** module, which contains code that specifies the operations performed by each bytecode instruction in terms of Jupiter's **Base** class operations. The **ExecutionEngine** always executes this specification code regardless of whether it is interpreting or compiling bytecode. When the **ExecutionEngine** is interpreting bytecode, it uses an implementation of the **Base** classes which performs the expected actions. However, when the **ExecutionEngine** chooses to generate native code, it invokes an implementation of the same **Base** classes that generates an intermediate representation (IR). This IR is then passed to the JIT compiler to generate optimized native code. This scheme is shown in Figure 6.

For example, the specification of the **putfield** opcode, which stores a value from the operand stack into a field of an object, uses the **Frame** and **Object** classes as follows:

```

1  case PUTFIELD:
2      Field fd = ctp_field(constantPool, CODE_USHORT(1));
3      Value v1 = fr_popOperand(curFrame, fd_type(fd));
4      Object ob = fr_popOperand(curFrame, cl_type(fd_class(fd)));
5      ob_setField(ob, fd, v1);
6      break;

```

The call to **ctp\_field** in line 2 looks up the appropriate field in the constant pool. The two calls to **fr\_popOperand** in lines 3 and 4 take the new field value and the target object reference off the stack. Finally, the call to **ob\_setField** in line 5 sets the appropriate field of the target object to its new value.

When the **ExecutionEngine** is interpreting bytecode, calls to **fr\_popOperand**, and **ob\_setField** use the interpreter's operand stack to perform their actions. In contrast, when the **ExecutionEngine** is generating native code, calls to **fr\_popOperand** in lines 3 and 4 simply return numbers of virtual registers storing the corresponding values. These numbers are saved in the variables **v1** and **ob**, respectively. The call to **ob\_setField** in line 5 then uses the register numbers to generate the store instruction which will actually set the field's value. Thus, the execution of the **putfield** opcode is accomplished with a single instruction that directly accesses its operands.

This approach to the design of the execution engine and the JIT compiler allows the sharing of code between the interpreter and the JIT compiler. A similar approach to JIT design has proven successful in Kaffe [11], and we expect it to provide the same benefits to Jupiter.

As in Kaffe, this approach requires a certain discipline when writing the `ExecutionEngine` code, since that code must behave very differently with the JIT compared to the interpreter. To allow for this, it must only perform actions which can occur at JIT-compile time, calling functions to perform actions which occur at run time. For instance, when specifying the `iadd` opcode, it may be tempting to write code like this:

```
1  case IADD:
2      int v1, v2;
3      v1 = fr_popOperand(curFrame, INT);
4      v2 = fr_popOperand(curFrame, INT);
5      fr_pushOperand(curFrame, v1+v2, INT); /* oops */
6      break;
```

The actual addition takes place in line 5. However, this makes it difficult to use this same code for the JIT, because that addition must occur at run time, not at compile time. Instead, the interpreter must call a function that adds the two values. For the JIT, another function can be substituted which generates the appropriate add instruction.

## 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 uni-processor performance. However, such improvements carry over to multiprocessors, and we expect them to be easily integrated into the Jupiter framework.

There exist a number of projects investigating JVM on parallel machines, including clusters of workstations [12, 13, 14, 15, 16, 17, 18]. The goals of these projects were to provide a JVM-based system image and to provide an environment for parallel execution of multi-threaded Java applications on clusters. Although these projects differ in their approach to implementing Java on clusters of workstations, they all target only a small number of processors (8 to 16). In contrast, our work aims to enable efficient execution of parallel Java programs on large numbers of processors (64 to 128). In additions, while these projects have built JVMs to explore their respective approaches to performance, they have not implemented their JVMs in such manner to allow exploring alternative approaches. In contrast, we are building a modular and extensible JVM that will allow us, and others, to explore multiple approaches to JVM scalability.

## 6 Concluding Remarks

In this paper, we presented the basic design of a modular and extensible JVM called Jupiter, composed of many building-block modules with small interfaces. Jupiter's design will provide the JVM with the power and simplicity that Unix shell pipelines have provided to operating systems for decades. Yet, since our module composition occurs at compile-time, this approach has the potential to reduce runtime overhead to zero when required. We expect this infrastructure to facilitate our research into JVM design for parallel scalability.

## References

- [1] B. Alpern and et al., “The Jalapeño virtual machine,” *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.
- [2] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganama, T. Onodera, H. Komatsu, and T. Nakatani, “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 *Proceeding of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pp. 1–17, 1999.
- [4] P. Wu, S. Midkiff, J. Moreira, and M. Gupta, “Efficient support for complex numbers in Java,” in *Java Grande*, pp. 109–118, 1999.
- [5] T. Domani and et al., “Implementing an on-the-fly garbage collector for Java,” in *Proceedings of the ACM SIGPLAN Symposium 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 *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.
- [7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. Addison-Wesley, 2000.
- [8] W. Pugh, <http://www.cs.umd.edu/~pugh/java/>.
- [9] E. Gagnon and L. Hendren, “SableVM: A research framework for the efficient execution of Java bytecode,” in *Proceedings of the USENIX JVM Research and Technology Symposium*, pp. 27–39, 2001.
- [10] Sun Microsystems, <http://www.java.sun.com/>.
- [11] T. Wilkinson, “Kaffe - a virtual machine to run Java code,” <http://www.kaffe.org/>.
- [12] W. Yu and A. L. Cox, “Java/DSM: A platform for heterogeneous computing,” *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1213–1224, 1997.
- [13] Y. Aridor, M. Factor, and A. Teperman, “cJVM: A single system image of a JVM on a cluster,” in *Proceedings of the International Conference on Parallel Processing*, pp. 21–24, 1999.
- [14] J. Andersson and et al., “Kaffemik—a distributed JVM featuring a single address space architecture,” in *Proceedings of the USENIX JVM Research and Technology Symposium Work-in-progress Session*, 2001.
- [15] M. Ma, C. Wang, F. Lau, and Z. Xu, “JESSICA: Java-enabled single system image computing architecture,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 2781–2787, 1999.
- [16] M. Philippsen and M. Zenger, “JavaParty—transparent remote objects in Java,” *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1225–1242, 1997.
- [17] G. Antoniu, L. Bouge, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst, “The hyperion system: Compiling multithreaded Java bytecode for distributed execution,” *Parallel Computing*, to appear, 2001.
- [18] T. Kielmann, P. Hatcher, L. Bouge, and H. Bal, “Enabling Java for high-performance computing: Exploiting distributed shared memory and remote method invocation,” *Communications of the ACM*, to appear, 2001.