Jupiter: A Modular and Extensible Java Virtual Machine Framework

by

Patrick Doyle

A Thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science,
Graduate Department of Electrical and Computer Engineering,
in the University of Toronto

Jupiter: A Modular and Extensible Java Virtual Machine Framework

Patrick Doyle
Master of Applied Science, 2002
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

We present and evaluate the design and implementation of a flexible and efficient Java Virtual Machine (JVM) framework called Jupiter. This framework employs a modular, object-oriented building-block architecture to provide the flexibility required to explore a wide variety of design ideas with relatively little effort. In addition, the Jupiter framework is designed to provide the performance necessary to properly evaluate the impact of various implementations and combinations of ideas. Evaluation of Jupiter's modular structure through modification experiments indicate that it is highly flexible and extensible. Evaluation of the Jupiter interpreter's performance using the standard SPECjvm98 benchmark suite shows that it provides good performance: it is 2.65 times faster than Kaffe, a freely available JVM, and 2.20 times slower than Sun's highly-optimized JDK interpreter. We believe that Jupiter's unique flexibility and good performance make it an excellent vehicle for research into JVM scalability issues for high-performance computing on large multiprocessors.

Acknowledgments

I extend my sincerest thanks to all who provided the inspiration, advice, and assistance I needed to make Jupiter work. To Tarek Abdelrahman, for his invaluable feedback and high standards. If they are not reflected in this document, the fault is entirely mine. To Carlos Cavanna, for his help with multithreading issues, for guiding Jupiter on its first steps toward scalability, and for the many painful hours he has spent finding and fixing my mistakes. And to Mathew Zaleski, for giving me the insights only Mathew can give.

Thanks also to those who have worked on the projects that laid the foundations upon which Jupiter was built. To the Classpath team and to Hans Boehm, for writing 90% of the code in the system I blithely refer to simply as "Jupiter." And to the Kaffe team, for proof by example when I needed it most, that a JVM can indeed be written.

Thanks most of all to my family and friends who have provided continuing love and support. To my parents, for their innumerable words of wisdom, and their continuing enthusiasm for documents with my name on them. Finally, to my wife Trixie, for periodically placing food in front of me to keep me alive; and for her inexhaustible patience, kindness, and devotion. This document represents as much work of hers as of mine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of the Java programming language has steadily increased over the past few years, in large part because of its portability. A Java program is first compiled into *bytecode* and stored in a *classfile*, a portable executable format targeted to an abstract platform specification known as the *Java Virtual Machine*, or JVM. The compiled program is then executed by a run-time system that emulates the JVM[1]. This approach allows programs written in Java to execute on a wide variety of platforms, wherever a JVM run-time system has been implemented.

However, in spite of its popularity, the use of Java remains limited in high-performance computing, mainly because of the performance degradation caused by the need to emulate a machine different from the host architecture. To address this issue, a wide variety of technologies have been developed to improve virtual machine performance. The most significant such technology is *just-in-time* (or "JIT") compilation [AAB+00, IKY+99, MOS+98, Mar01], which translates bytecode into equivalent code native to the host machine immediately prior to execution, allowing that code to execute without the performance penalty of emulation.

An independent and complementary approach to improving the performance of Java programs is to execute the JVM in parallel on a large number of processors. Ideally, a JVM that takes this approach should be *scalable*; that is, it should not impede the Java

[1]The run-time system itself it commonly referred to simply as a "JVM;" hence, we adopt this terminology throughout this document.

program's ability to make use of all the processing resources available. JVM scalability is a difficult problem, even for a small number of processors, and a great deal of work has been done on the use of small-scale multiprocessors [AAB+00, DAK00], and even small clusters of workstations [AWC+01, MWLX99, PZ97, AFT99, ABH+01, YC97, KHBB01], to improve the performance of Java programs.

The goal of the Jupiter project at the University of Toronto [Jup02] is to develop a JVM that scales to a large number of processors[2]. It is envisioned that, for the project to achieve this goal, it will be necessary for the JVM to examine several design issues, including locality-enhancing memory allocation, exploitation of relaxed memory consistency models, parallel garbage collection, and efficient threads and synchronization [DA01]. To examine alternative approaches to addressing these issues, it is necessary to have a flexible JVM framework that facilitates the exploration of different strategies by multiple researchers, and to multiply their individual efforts by easily combining their contributions.

Regrettably, existing JVMs whose source code is available for research do not offer the level of flexibility to prototype ideas quickly, and hence tend to impede progress. Some, such as Sun's JDK [Sun02] or Kaffe [Wil02] are not designed primarily to be flexible research platforms, making experimentation on these systems difficult. Others are designed to be flexible only in certain specific areas, such as Jalapeño [AAB+00, Jal02] and OpenJIT [MOS+98, Mar01], which are designed for research into JIT compilation. In order to pursue the above research goals, a JVM with pervasive, fine-grained flexibility is needed.

It is this need that motivated our development of the Jupiter Java Virtual Machine (henceforth referred to simply as "Jupiter"), a modular and extensible framework designed to support experimentation with a wide variety of techniques for improving performance and scalability.

[2]The ultimate target of the current work is a cluster of up to 128 processors.

1.1 The Jupiter Java Virtual Machine Framework

Jupiter is a JVM framework based on a highly flexible building-block architecture that allows JVM implementations to be assembled out of component parts with a small investment of effort. The component modules themselves are designed to insulate design decisions from each other, allowing them to be combined in powerful ways, thus multiplying the efforts of individual researchers. Furthermore, because the underlying motivation for Jupiter's flexibility is to achieve good performance, the module interfaces have been carefully designed to allow for efficient implementations.

The current implementation of Jupiter is a functional JVM that provides the basic facilities required to execute Java programs. It has a single-threaded interpreter, with multithreading capabilities currently under development by other researchers [Cav02]. Jupiter gives Java programs access to the Java standard class libraries via a customized version of the GNU Classpath library [CP02], and is capable of invoking native code through the Java Native Interface [JNI02]. Nonetheless, 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 has been tuned to the point that it is comparable with commercial and research interpreters (as discussed in Section 6.2) while still maintaining a high degree of flexibility.

Jupiter is written in an object-oriented style, even though it is implemented in C. Although languages such as Java or C++ would provide more support for an object-oriented programming style, with facilities for encapsulation, inheritance, and polymorphism, we have chosen C because we did not have confidence in the ability of other languages to provide the facilities required to achieve good performance. C provides the trade-off between ease of use and performance that we feel is most conducive to future JVM research. It is a proven tool for writing high-performance systems, and by the use of design techniques covered later in this thesis, we have achieved the level of flexibility. Despite the lack of object-oriented features in the implementation language, we still refer to Jupiter using object-oriented terminology throughout this document. Such Jupiter concepts as "object," "class," and "interface" are all analogous to their

Java counterparts.

The use of flexible, efficient building-blocks to construct a high-performance scalable system is a technique that has proven successful on other systems [K4201, FSUZ88, Gam99, KS97], and we expect the same benefits to emerge from its use in Jupiter.

1.2 Contributions

Our contribution is the design and implementation of a flexible JVM framework that delivers acceptable performance. Jupiter's building-block architecture and careful module design allow modifications to be made with relatively little effort, supporting the rapid exploration of design alternatives. Experimentation with our framework demonstrates this flexibility, by easily implementing a number of sophisticated modifications, some of which are difficult to accomplish using Kaffe [Wil02], another freely-available JVM.

This flexible architecture comes with a small enough performance cost that future experimental results acquired from Jupiter can be expected to be applicable even to JVMs with cutting-edge performance. Performance measurements using the single-threaded SPECjvm98 benchmarks [SPE02] show that the performance of Jupiter's interpreter exceeds that of Kaffe by a factor of 2.65, and is comparable with that of Sun's JDK [Sun02]—a highly optimized commercial JVM—within a factor of 2.20.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of Jupiter's architecture, along with its design philosophy, and a survey of the major system components. Chapter 3 discusses implementation, describing the contents of the components with the goal of clarifying their purpose, as well as describing the work that has gone into achieving Jupiter's current level of functionality. Chapter 4 discusses flexibility, presenting a number of design techniques that have allowed us to achieve flexibility within Jupiter. Chapter 5 discusses performance, presenting the design considerations that have given Jupiter its current level of performance despite its

flexibility. Chapter 6 demonstrates the impact of all these design decisions by presenting evidence of Jupiter's flexibility and performance. Chapter 7 presents concluding remarks, and discusses opportunities for future work.

Chapter 2

Architecture

Before the design of a system can be addressed, it needs to have an architectural vision that supports the underlying philosophy of the system. For Jupiter, the requirement of flexibility led us to choose a building-block architecture, viewing the system as an assemblage of building-block facilities managing the resources of the running Java program. In this chapter, we present these concepts, and provide a series of examples that demonstrate how our goals can be achieved by this choice of architecture. Afterward, we finish the chapter with a brief review of related work.

2.1 System Structure

It is indicative of a clear, unified, cohesive system that its purpose can be described by single short sentence. For Jupiter, this sentence reads as follows: "A JVM is a device to manipulate computational resources on behalf of a Java program." This philosophy is depicted in Figure 2.1, which gives a highly simplified view of Jupiter's conceptual structure. In the center is the `ExecutionEngine`, the control center of the JVM which decodes the Java program's instructions and determines how to manipulate the program's 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`

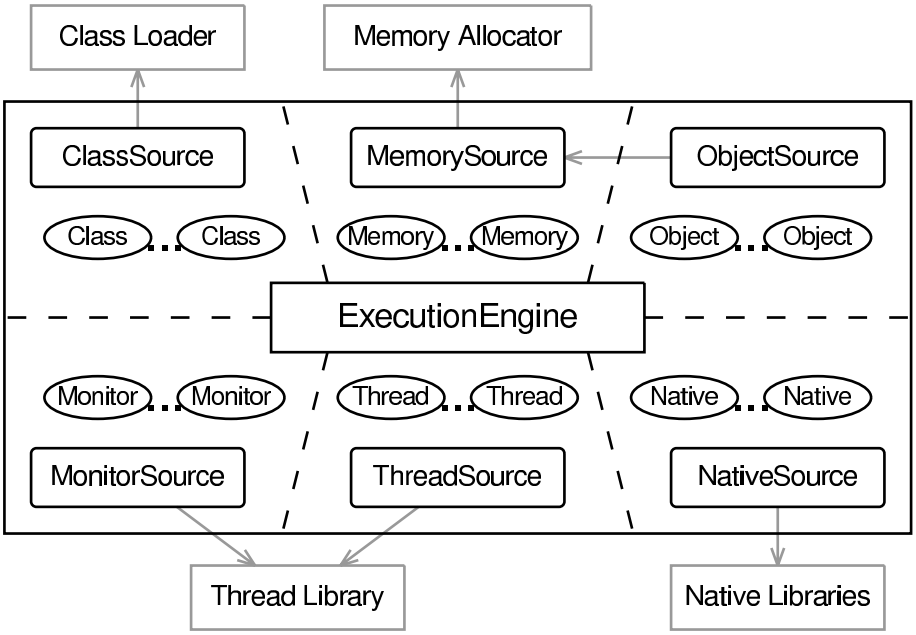

Figure 2.1: Jupiter's conceptual structure. Resource management responsibility is divided and delegated to number of Source modules, leaving the ExecutionEngine with a "pure," simplified view of the system's resources.

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* [KS97]. As each Source is instantiated, its constructor takes references to other Sources it needs in order to function. For instance, as shown in Figure 2.1, 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.

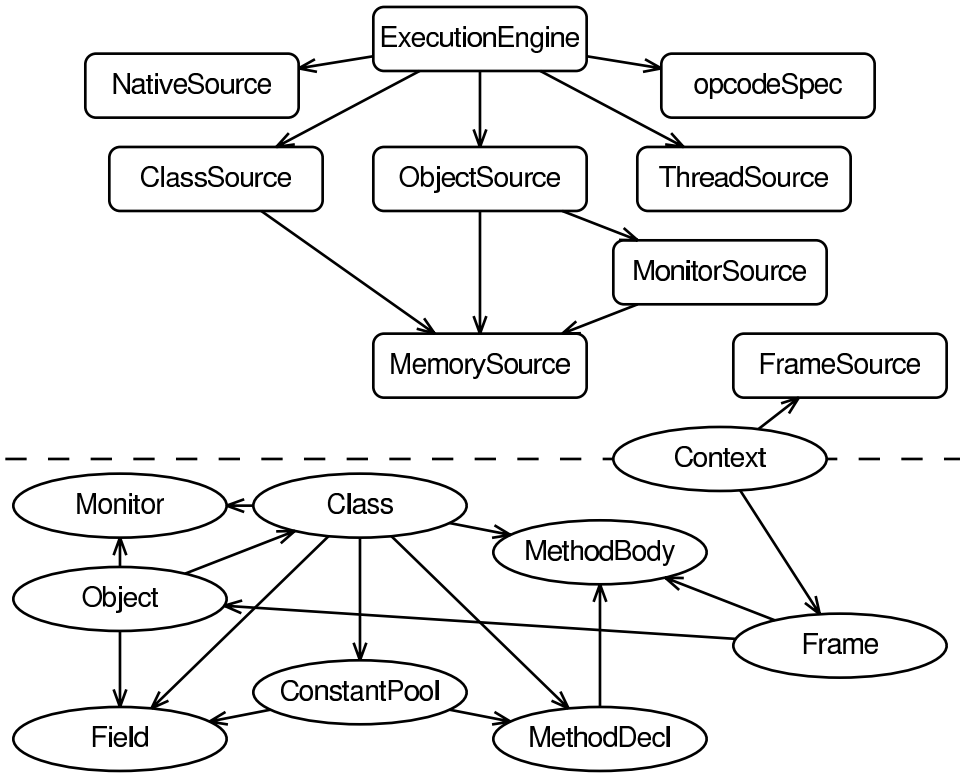

Figure 2.2: A typical building-block structure. Sources are shown above the dashed line, and resources below. The Context object acts as a link between the two when the Java program first begins to execute.

The assembly of a JVM our 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 2.2 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 appropriate sources and resources in the appropriate way, Jupiter is able to perform the indicated operations, thereby executing the Java program.

2.2 Base Interfaces

In any system, some interfaces are fundamental, while others are artifacts of particular implementation decisions. Fundamental interfaces are unlikely to change often, while less fundamental ones may change as new implementation decisions are made. All interfaces fall into a spectrum, with the most fundamental and stable at one end, and the most experimental and volatile at the other. Jupiter represents this spectrum explicitly in its module structures, with the most fundamental modules grouped together in a package known as the *base directory*[1].

These interfaces are the foundation upon which Jupiter is constructed; together, they define those aspects of the system that do not change despite Jupiter's tremendous flexibility. The `base` directory includes interfaces to all the facilities that any JVM is required to possess, no matter what configuration, permutation, or modification might be employed. Hence, any code that depends only on these interfaces is completely "portable" in the sense that it can be used within any incarnation of Jupiter.

One central design challenge in Jupiter has been to choose the right interfaces for inclusion in the `base` directory. If too much functionality is exposed through these interfaces, the range of implementation possibilities becomes constrained by the need to provide the required semantics. If too little, then implementors will be forced to depend on non-`base` interfaces to access the desired functionality, thus increasing module interdependencies throughout the system, and defeating the purpose of having `base` interfaces in the first place. For each interface, therefore, we must as the question "does

[1]The package hierarchy arising from the interface stability spectrum is depicted in Figure 3.3.

every JVM possess this functionality?" If the answer is "no," the interface must be redesigned to eliminate constraints on its implementation.

The remainder of this section gives brief tour of the base interfaces of Jupiter.

2.2.1 MemorySource

`MemorySource` can be considered one of the most fundamental of the base interfaces, since it does not depend on any other interface, and practically every module in the system makes use of memory in one way or another. It is deliberately as simple as possible, in order to accommodate many different implementations. It provides just one function, called "`getMemory`," which takes the size of the memory block required, and returns the resulting block.

Jupiter provides no interface between the memory allocator and the garbage collector. Both of these facilities, and all their interactions, are hidden behind the `MemorySource` interface. See Section 7.1 for a discussion of the limitations of this interface.

2.2.2 Class Metadata

Jupiter creates metadata resource objects to represent the Java program itself. These objects take the form of `Classes`, `MethodDecls`, `MethodBodies`[2], and `Fields`. Jupiter accesses classes by name through the `ClassSource` interface, using a function called `getClass`. Once a `Class` has been acquired, its `Fields` and `MethodDecls` and `MethodBodies` can be accessed in order to perform the operations required by the running Java program.

In accordance with Java semantics, `ClassSource` keeps track of the `Class` returned for each name, so that future references to the same name return the same `Class` object.

[2]For more information on the relation between `MethodDecl` and `MethodBody`, see Section 3.7.

2.2.3 Object and ObjectSource

The `Object` interface represents the objects in a running Java program, hiding their internal representation from the rest of Jupiter. There are two kinds of objects: arrays, and instances of non-array classes. In Jupiter, these are referred to as `Arrays` and `Instances`, respectively. Both are descended from a common "superclass" called `Object`, and both can be allocated by `ObjectSource` through the `getInstance` and `getArray` functions. Each function takes the `Class` being instantiated, plus any other necessary information (such as the dimensions for an array), and returns the new `Object`.

2.2.4 Context, Frame, and FrameSource

Each Java thread has a number of data items associated with it, aside from the shared object heap. These items are known collectively as a *Context*, and are accessible through the `Context` interface. A `Context` is simply the call stack for a thread; specifically, it is a collection of individual stack frames for the methods in progress within that thread. The frames themselves are represented by the `Frame` interface, which provides access to the components of each Java stack frame, including the local variables, operand stack, program counter, etc.

When a method is invoked, a new `Frame` is needed. The `Frame` is acquired by the `Context` from the `FrameSource` interface, through the `getFrame` function.

Representing the Java context entirely as a data structure allows Java threads to be migrated, simply by executing a given `Context` on a different native thread. This stands in contrast to the more straightforward scheme that implements method invocation by recursion within the execution engine [Wil02], causing context information to be stored on the native stack, and precluding this kind of migration.

2.2.5 ExecutionEngine

The `ExecutionEngine` class decodes the bytecodes and performs the actions necessary to implement each instruction. Its interface is quite simple, consisting of a single

function called `ee_executeTo` that takes a `Context` as an argument, and executes the method whose frame is on top of the `Context`'s call stack.

This interface is intended to be as flexible as possible, to allow for a variety of execution scenarios. For ideal flexibility, the function would execute a single instruction and then return, giving the caller complete control over the execution process. However, this would add enormous overhead to the cost of executing each instruction, and is thus impractical. Another possibility would be to execute until a method is invoked or returned; however, again, this would add considerable overhead to programs that make many short-lived method invocations.

To provide the desired degree of control, the `ee_executeTo` function takes an additional argument called the *stop frame*; execution proceeds until the stop frame is found to be at the top of the call stack. This allows for large quantities of code to be executed with a single function call, yet still permits the desired control over execution.

For example, there are situations in which the JVM needs to invoke Java methods "spontaneously," without being instructed to do so by the Java program itself, such as when a newly-loaded class is initialized. In such cases, the stop-frame technique allows an existing `Context` to be re-used: a new frame is pushed onto the call stack for the class initializer, and then `ee_executeTo` is called with the previous top frame as the stop frame. When that frame is eventually found to be the topmost frame on the call stack, that means the class initializer has terminated (either by returning normally or throwing an exception). At that time, `ee_executeTo` will return, which is exactly the desired effect.

2.2.6 Thread and Monitor

Jupiter provides two resources that cooperate to implement Java's multithreading capabilities: `Thread` and `Monitor`. The `Thread` interface provides the means for executing a `Context` concurrently with others, and the `Monitor` interface provides the functionality required to implement Java's synchronization semantics [LY99]. These resources are created and managed through the `ThreadSource` and `MonitorSource` interfaces.

Our initial implementation of Jupiter is single-threaded. Others are pursing a mul-

tithreaded implementation [Cav02], and although it is largely functional at the time of writing, multithreading is outside the scope of our own work, and so we do not discuss the `Thread` and `Monitor` interfaces in detail in this document.

2.2.7 Native and NativeSource

A `Native` represents the body of a single native method, and embodies everything needed to call that method, including the calling conventions and any extra implicit arguments required. The `NativeSource` interface provides a `getNative` function that provides the appropriate `Native` for a given native method. Typically, it looks up the `Native` in a shared library or DLL using whatever introspective capabilities are provided by the system. However, `NativeSource` could also be used to model a JIT compiler, which could fabricate a `Native` for each method as needed.

2.2.8 OpcodeSpec

Assuming that Jupiter's base interfaces achieve their goal of providing access to all the essential JVM functionality in a very general way, it becomes possible to describe the actions performed by each Java opcode entirely in terms of those base interfaces. This is the role of the `opcodeSpec` module, which specifies each opcode in a manner that is independent of the implementation decisions made by any particular incarnation of the Jupiter system. The task of building a JVM is largely reduced to that of providing implementations for each base interface. Once the base interfaces are implemented, `opcodeSpec` ensures that the JVM will have the correct behaviour according to the Java specification.

2.3 Configuration Examples

In this section, we demonstrate Jupiter's flexibility by examining several configurations of the system's building-block modules. We focus mainly on a recurring example—the object creation subsystem—though some other examples are also provided to illustrate how the same kind flexibility can be found throughout the system. Through examples,

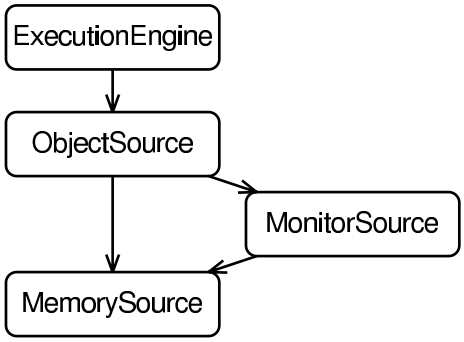

Figure 2.3: A simple object allocation building-block structure.

we present several ways in which Jupiter can be modified (which we refer to as "modes of extension").

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 [BW88, Boe02]. 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 2.3, where arrows indicate the *uses* relation between the 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. 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.

2.3.1 In-place Substitution

Having established the basic structure of the object allocation subsystem, there are many modifications that can be explored. To begin with, suppose an alternative layout for the object fields is desired [GH01]. Such a modification simply substitutes a new `ObjectSource` which computes the proper size of the object based on the new

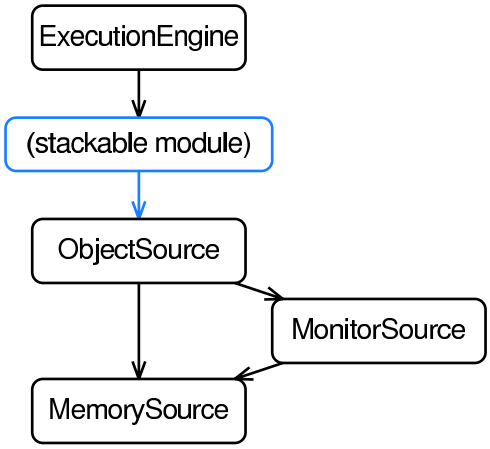

Figure 2.4: Interposition of a stackable ObjectSource module.

layout[3], and uses the same `MemorySource` and `MonitorSource`. Conversely, using a different garbage collector means substituting the `MemorySource` while using the same `ObjectSource` and `MonitorSource`. 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.

2.3.2 Stacking

A second mode of extension involves stacking new modules on an already-working building-block structure. The enabling characteristic is that the stacking module must import and export the same interface. Anywhere that interface is used, a stackable module can be interposed transparently between caller and callee, thus extending the functionality of that interface in some specific way.

One application for module stacking is profiling: by interposing a data-collecting

[3]Of course there is more to changing the object layout than simply computing the proper object size, but this is not pertinent to our example.

layer between caller and callee, information can be gathered regarding the usage of that interface. For example, this technique allows us to do memory profiling using the configuration shown in Figure 2.4. The `Tracer` object can record information in the memory chunks it allocates, by requesting slightly larger chunks from the underlying `MemorySource`. Then, when the garbage collector does its traversal of the memory contents, it can also report statistics on memory usage.

Module stacking can also be used to assist the testing process. To be effective, tests should cover as much of the code as possible. Ideally, the test suite should cover every line of code, but this is difficult because of error-handling code which only executes under unusual circumstances. It is sometimes difficult to reproduce circumstances such as out-of-memory conditions, especially on modern virtual-memory operating systems, whose performance becomes unacceptably poor long before virtual memory is exhausted. Further, even if such conditions could be synthesized by limiting the memory available to the JVM, it may still be difficult to cause error conditions to occur at all the appropriate places to exercise all error-handling code.

This problem can be addressed by interposing a module whose only purpose is to cause errors to occur at predictable "error injection sites" in the code. For example, an `ErrorMemorySource` can be added to check the source code coordinates of the `getMemory` call, again using the configuration shown in Figure 2.4. Most calls are forwarded to the underlying `MemorySource`. However, when the coordinates match that of an error injection site, `getMemory` will report an error, causing the appropriate error-handling code to be exercised[4]. This same technique could be applied to any interface in the system, to check the system's handling of errors encountered by that interface.

2.3.3 Reconfiguration

A third mode of extension involves changing the configuration of the modules. In many cases, the modules need not be modified at all; only their relations with respect to other

[4]In Jupiter's implementation of this scheme, the injections sites are established by calls from the Java code itself, through native methods provided by a special `Util` class. This allows all the conditions required for the test to be controlled from within the Java code.

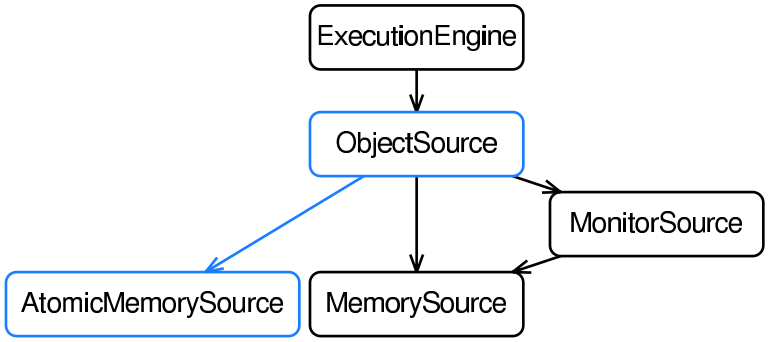

Figure 2.5: Reconfiguration to allow atomic memory allocation.

modules change. This mode of extension permits a variety of modifications that are not possible with stacking and/or in-place substitution alone.

For example, imagine the `MemorySource` being used by our system is a conservative garbage collector, such as Boehm's. This type of collector has difficulty dealing with large arrays of scalar (non-pointer) data. If the data in the array is sufficiently large, and its contents are sufficiently random, it becomes likely that there will exist a word-size sequence of bytes which are indistinguishable from a pointer. Due to the conservative nature of the garbage collector, such false pointers cause it to retain objects which are actually unreachable. If more than one large array exists, the problem is compounded, because the large size of the arrays makes it likely that a false pointer in one array will point somewhere within another. For instance, on a 32-bit architecture, the existence of two 300KB arrays of random bytes will result in a 99% chance that each will have a false pointer to the other[5]. If either array is reachable, both will be retained, which is all the more harmful precisely because the arrays are large.

To prevent a breakdown of the garbage collector in such cases—and to avoid the unnecessary scanning of large scalar arrays in the first place—it is desirable to inform the collector that certain areas of memory contain no pointers, and need not be scanned; i.e. that they are *atomic*[6]. It may appear at first that the simplistic `MemorySource` interface

[5]This assumes arrays of uniformly distributed 4-byte pointers, covering a 4-gigabyte address space.

[6]The distinction between "scalar" and "atomic" is the distinction between policy and mechanism: "scalar" describes data that contains no pointers, while "atomic" refers to memory blocks that are not scanned for pointers by the garbage collector. Note that scalar data can be stored in memory which is not atomic.

provides no mechanism for passing this kind of information to the underlying memory manager; however, the building-block structure can be reconfigured to accomplish this. The result is shown in Figure 2.5. Two `MemorySources` are provided: a new one (the `AtomicMemorySource`) for allocating *only* memory which will not contain pointers, and the original one with conservative semantics. The `ObjectSource` now makes use of the `AtomicMemorySource` to allocate scalar arrays, and the regular `MemorySource` for everything else. Thus, a modification that appears not to be feasible using Jupiter's interfaces can be implemented easily by reconfiguring the building blocks.

A similar mode of extension is useful to address Jupiter's scalability on a multiprocessor system with non-uniform memory accesses (*NUMA*). 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.

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 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 memory module in 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 2.6. `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, as shown in Figure 2.7. `MuxObjectSource` is similar to `MuxMemorySource`, in that it uses some

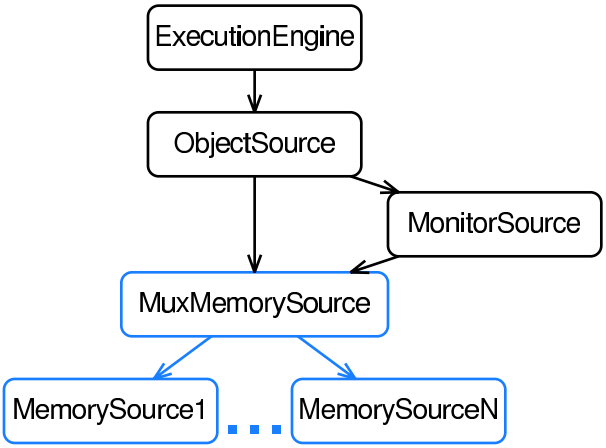

Figure 2.6: Locality decisions made at the MemorySource level.

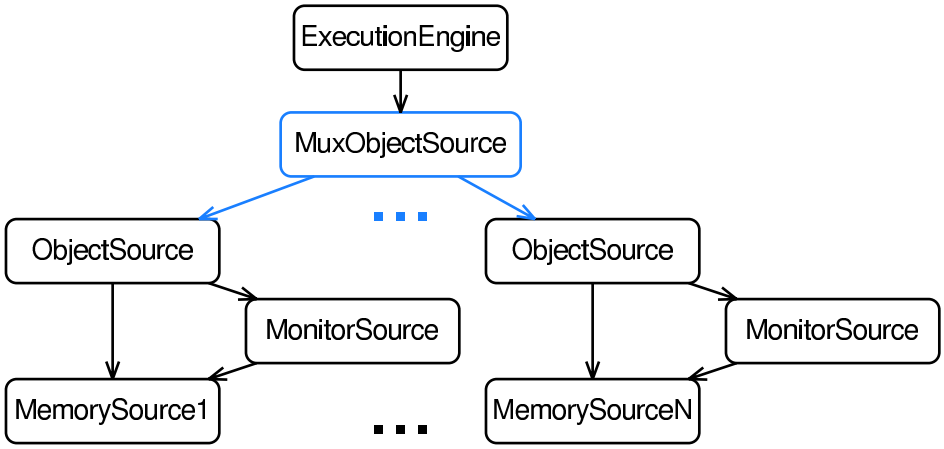

Figure 2.7: Locality decisions made at the ObjectSource level.

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 2.6. We can also use the same `ObjectSource` and `LockSource` classes as in the original configuration (Figure 2.3); 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. In this configuration, shown in Figure 2.8, 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`.

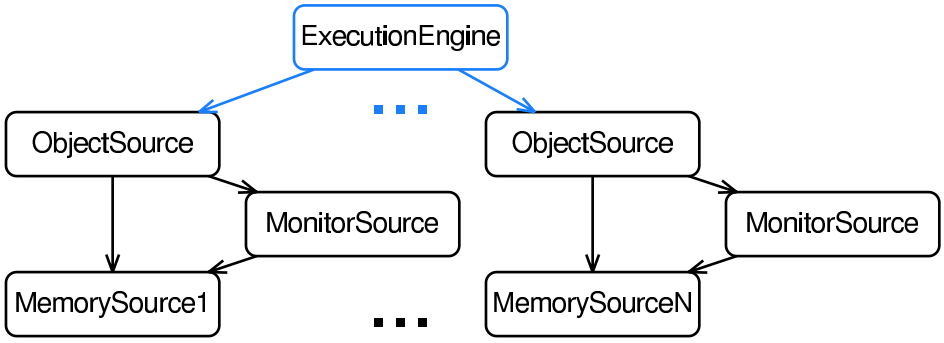

Figure 2.8: Locality decisions made by the ExecutionEngine itself.

Again, we have not changed `ObjectSource` or `LockSource` classes, and the node-specific `MemorySource` class is the same one from the previous configurations.

These examples illustrate the rich variety of system configurations which can be implemented with minimal effort due to the flexibility of the building-block architecture.

2.3.4 Efficiency Issues

At first glance, it would appear that our flexible structure results in much object duplication within the JVM. All the proposed memory allocator structures have portions that are 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 could 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 2.6. 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 a new implementation of this header file (thus *overriding* the base implementation) by substituting the above definitions with the following:

```
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* [CHS72], 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 2.6 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.

2.4 Related Work

Previous work in a number of areas has influenced the design and implementation of Jupiter. We have strived to combine the best existing theory and practice with a few original ideas on how to create flexible systems, and apply them to the construction of a highly flexible JVM. Therefore, Jupiter's influences stem from two different sources: JVM implementations, and modularity research.

2.4.1 JVM Implementations

There are a number of JVM framework projects designed for flexibility. However, they all address a particular dimension of JVM design. In contrast, Jupiter's flexibility is intended to be pervasive and fine-grained, allowing straightforward modification of any aspect of the system.

The original JVM, Sun's JDK [Sun02], is available in source-code form for research purposes. It is modular in a way that permits experimentation in certain dimensions, with certain portions of the system designed to be interchangeable. In contrast, Jupiter has taken on modularity as a fundamental architectural feature, and is intended to provide flexibility in many more dimensions, at a smaller level of granularity. Rather than view the system as a unit with certain interchangeable parts, Jupiter views the system as composed of nothing but interchangeable parts. Instead of prescribing that certain parts of the system be flexible, Jupiter's flexibility emerges from its pervasive, fine-grained module structure.

The Kaffe project [Wil02] is an freely available JVM which claims modularity as one of its features. Like the JDK, Kaffe's modularity provides flexibility in certain dimensions, specifically: threading, memory management, native method interfacing, and native system calls. However, much like the JDK, Kaffe does not share Jupiter's pervasive, fine-grained approach toward flexibility.

In addition, there are a number of other JVM projects that are intended for research.

IBM's Jalapeño JVM[AAB+00, Jal02] (now part of the Jikes Research Virtual Machine project[Jik02]), is designed to explore the implementation of an industrial-grade server JVM written in Java. Though its object-oriented design undoubtedly possesses some degree of inherent flexibility, it appears to embody certain implementation decisions, such as object layout, stack layout, method dispatch, etc. It is not clear whether these aspects of the system can be easily changed in Jalapeño, while Jupiter allows them to be changed with minimal effort.

Perhaps the closest relative to Jupiter is the Joeq JVM [Wha02] developed during the same time period as Jupiter by John Whaley, one of the original designers of the Jalapeño project. Like Jupiter, Joeq is designed to be flexible to facilitate research into JVM implementation techniques. It is written in Java in an object-oriented style, so it is likely to be inherently flexible. However, it does not arrange its facilities into the building-block architecture that Jupiter exploits to achieve its high degree of modularity.

The OpenJIT project [MOS+98, Mar01] is an object-oriented framework for experimenting with JIT compiler designs, written in Java. It is implemented as a JIT-compiler plug-in to Sun's JDK. Hence, it limited to JIT compiler research specifically, while Jupiter is intended to facilitate all aspects of JVM research.

The SableVM project [GH01] is a framework for research into efficient bytecode interpretation. It does not possess the flexibility of Jupiter's building-block architecture. While SableVM's primary focus is on performance, Jupiter's focus on modularity has led to a strict partitioning of design decisions into separately-modifiable modules; this concern is not reflected in SableVM's structure, which has roughly same quantity of code as Jupiter in 1/3 as many files. Also, despite its performance goals, it is not clear what work would be needed to incorporate a JIT compiler into SableVM, while Jupiter's separation of opcode specification from interpretation provides a clear road-map for JIT compiler implementation.

A number of projects such as cJVM [AFT99], Hyperion [ABH+01], Java/DSM [YC97], Kaffemik [AWC+01], and Jessica [MWLX99] provide a single JVM image on a cluster. It is our ultimate goal that Jupiter will likewise be extended

to operate on a cluster. While these other projects have chosen a particular approach to provide a single system image, our intent is for Jupiter to be flexible enough to experiment with any number of approaches.

2.4.2 Modularity Research

There are numerous modularity-related works that have affected Jupiter's design, and their application to the construction of a JVM is believed to be original.

The continuing research into modular operating systems at the University of Toronto, from Hurricane [FSUZ88] to Tornado [Gam99] to K42 [K4202], have inspired the building-block structure of Jupiter. The K42 publications make the point that building-block operating system composition allows customizations to be realized simply by rearranging the blocks [K4201], making system-level customizations as easy as application-level programming. While Jupiter has no division between application- and system-level *per se*, we believe the increased ease with which powerful customizations can be implemented is comparable.

Orran Krieger's Hurricane File System [Kri94, KS97] is exemplary of the kind of building-block flexibility that Jupiter has strived for. In particular, its stackable building blocks are the direct ancestor of Jupiter's own stackable modules, such as the `Tracer` and `ErrorMemorySource` (described in Section 3.2).

A number of other writings on coding practices have combined to enhance Jupiter's flexibility greatly, in a multitude of subtle ways, whose combined effect is as important as that of the building-block architecture itself.

At the design level, the building blocks themselves been crafted with careful attention to Design by Contract, along with its corollary, Command-Query Separation, both of which feature prominently in Bertrand Meyer's *Eiffel* programming language [Mey97]. The naming conventions were influenced both by Meyer's work, and by Tim Ottinger's naming rules [Ott02]. The assignment of functionality to modules were strongly influenced by careful consideration of information hiding, as prescribed by David Parnas [Par72]. Altogether, these influences have given to Jupiter's interfaces clear, predictable semantics, which are a precondition for flexibility, and a required property of

a successful component-based system.

At the implementation level, the greatest single influence is from John Lakos' work on the practical issues involved in writing large systems [Lak96]. Though Jupiter is not as large as the systems for which Lakos' disciplines were developed, nonetheless they have contributed significantly to its success. His admonition against circular dependencies greatly contributed to Jupiter's hierarchical coarse-grained module dependence structure (shown in Figure 3.3), though Jupiter's provision of a highly stable lowest-level "base" package, upon which other any module may freely depend, is believed to be original. In addition, Lakos' disciplined use of module prefixes has been employed, with some modifications, to great effect in Jupiter.

Java's scheme of exception handling [GJSB00], which is among the best-conceived to be found in object-oriented languages, greatly influenced the design of Jupiter's error handling system. This is not because, as one might assume, that being a JVM, Jupiter is in fact required to implement Java exceptions. Rather, we found certain properties of Java's exceptions (described in Section 4.5) to be desirable for robust systems, regardless of their functionality. The resulting error-handling idioms employed in Jupiter, which translate the benefits of Java's exception handling to C code, are original.

2.5 Conclusion

In this chapter, we have presented the building-block philosophy behind Jupiter's architecture, introduced the cast of modules that participate in the execution of the system, and demonstrated the resulting flexibility with a number of sample building-block configurations. We have briefly touched on the themes addressed by Jupiter's design: first dealing with the flexibility of its design, then with the organization of its modules (particularly the `base` directory), and finally in the performance of its implementation. Having established the direction in which the remainder of this document will proceed, we shall use the subsequent chapters to elaborate on each of these themes in an effort to convey Jupiter's contribution to JVM design.

Chapter 3

Implementation

Jupiter's primary goal is flexibility, and its flexible architecture allows for a wide variety of implementations. The particular implementation chosen for each module is not, in itself, the central concern of this thesis. However, it is helpful to take a tour of Jupiter's implementation in order to clarify the role of each module, and to present the work that has gone into achieving Jupiter's current level of functionality and performance. Further, the implementations described here may serve to explain why certain design decisions were made.

The data structures and algorithms presented in this chapter are not the only (nor even the best) possible. For expediency, our implementations generally use the simplest schemes that achieve the desired qualities in the final system (particularly performance). Being relatively simple, these schemes should provide clear examples of how the modules can be implemented.

Each implementation is first presented as though modularity were not a concern; afterward, we take a step back to examine how each implementation fits into Jupiter's module structure, focusing on the degree of coupling required between the modules.

Diagrams are used heavily to depict the modules and data structures being discussed. Thus, we begin by describing the nature of the diagrams we will be using. Afterward, we present Jupiter's implementation in a quasi-chronological order, roughly corresponding to the sequence in which they are first executed. At the topmost level, Jupiter's `main` function does the following things:

1. Build the JVM's infrastructure from instances of the Source modules.
2. Invoke `ClassSource` to load the main class.
3. Create a `Context` for the main thread.
4. Push the initial stack frame for executing the `main` method onto a `Context`.
5. Pass the `Context` to the `ExecutionEngine` to run until the `main` method terminates.

Accordingly, this chapter presents the implementation of the activities in Jupiter in the following sequence:

- Memory Allocation.
- Classfile Parsing.
- Stack Layout.
- Object Layout.
- Method Lookup.

3.1 Overview of Diagrams

The implementation's data structures are described with the aid of object diagrams, whose format is shown in Figure 3.1. These diagrams show the entities that exist at runtime within the Jupiter system. Sometimes, objects alone do not convey the most significant aspects of the runtime behaviour, and so the object diagrams are supplemented with symbols representing the interfaces through which certain structures are accessed. (The interface symbol is also shown in Figure 3.1.) Since interfaces are not really entities that exist at runtime, and are more relevant to modularity than to object interactions, they are included only when the intended behaviour would be unclear without them.

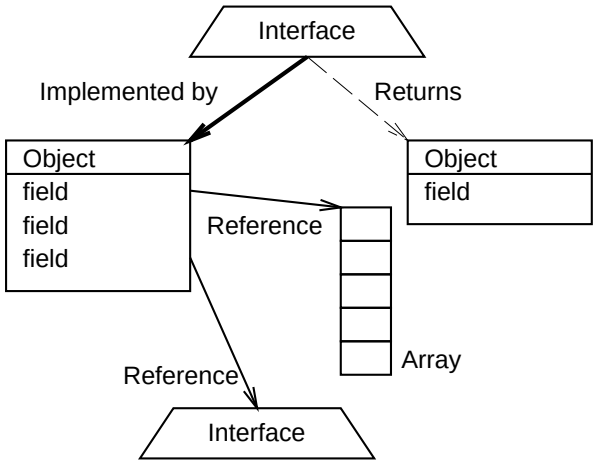

Figure 3.1: Object diagram legend. Arrows point in the direction of data reachability.

The implementations of some modules may use small amounts of special knowledge of other modules, leading to increased coupling. For example, an implementation of `ObjectSource` needs access to the constructor for `Object`; no such constructor is provided by the `base` interface of `Object`, so the `ObjectSource` must make use of additional interfaces in order to access this special functionality. This situation is much like using a subtype of a particular base interface, rather than restricting oneself to the use of the base interface alone.

We describe the coupling required by each implementation with the help of module diagrams, whose format is shown in Figure 3.2. These diagrams depict modules as ovals, with arrows indicating module dependencies. Each module consists of a `.h` file and possibly a `.c` file. The nature of each dependency arrow is discussed, and the impact on the modularity of the system is evaluated.

Every module in Jupiter resides in a certain directory, and the directories are arranged in a dependency hierarchy shown in Figure 3.3. Modules in the upper directories tend to depend on those in the lower directories, but never vice-versa. Most of the code in the system resides in the "Main Sequence" directories, which have a straightforward linear dependency hierarchy. The other *peripheral* directories contain modules that do not fit neatly into this sequence; they tend to be grouped by function (e.g. parsing, native methods, etc.).

The peripheral directories are not fundamental to the JVM's structure; for exam-

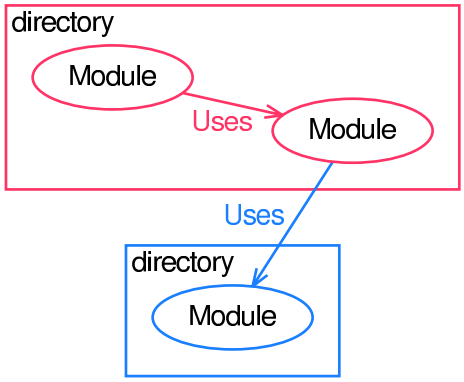

Figure 3.2: Module diagram legend. Arrows point in the direction of module dependence.

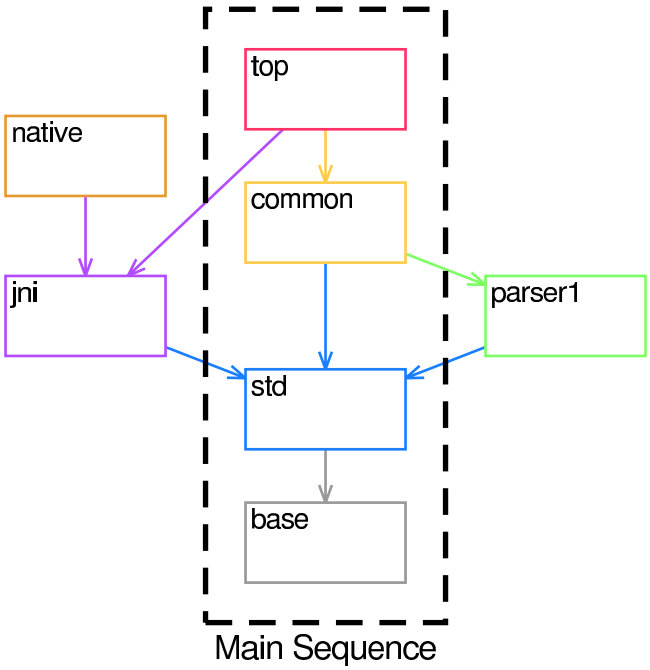

Figure 3.3: The module directory dependency hierarchy.

ple, while most JVMs would have a classfile parser, the particular implementation provided by the `parser1` directory could be entirely replaced by another parser (perhaps "`parser2`"), or by a completely new scheme that builds metadata by some mechanism other than parsing classfiles. Likewise, while Jupiter currently supports JNI (the Java Native Interface [JNI02]), other native interfacing schemes could be used instead, such as the older NMI (Native Method Interface), or the CNI (Cygnus Native Interface) used by `gcj`, the GNU Java compiler [CNI02, GCJ02]. Such modifications would involve re-

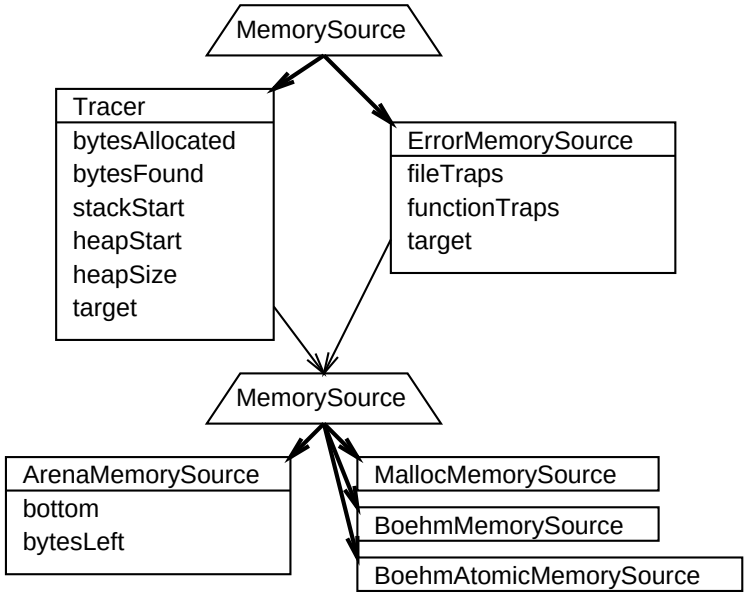

Figure 3.4: The memory allocation objects. The stackable building blocks are shown between two MemorySource interfaces.

placing the entire `jni` directory, indicating that this directory is not fundamental to the JVM's structure.

Every module has numerous dependencies on the `base` modules. Since those modules are so fundamental (and hence stable), such dependencies are quite benign, and they are often omitted to avoid cluttering the diagram with gray arrows. Arrows that are not gray indicate dependencies on module features that are not fundamental to Jupiter; they represent extra coupling between implementation modules.

3.2 Memory Allocation

Memory allocation in Jupiter is achieved by the `MemorySource` interface (introduced in Section 2.2.1). The `MemorySource` interface currently has more implementations than any other interface in the system. Because `MemorySource` is so highly polymorphic, and because its performance is usually not critical, a generic implementation is provided that uses a full-blown jump table system, much like the virtual method tables of C++, to achieve the utmost flexibility. This way, other modules can implement additional `MemorySource`s simply by supplying the appropriate function pointers.

Figure 3.4 shows the objects involved in memory allocation. A real running incarnation of Jupiter would not necessarily use *all* these objects, though there is nothing preventing it. The system's view of the memory allocator is through the `MemorySource` interface shown at the top of the diagram, while the actual memory allocators are at the bottom.

Starting at the bottom-right, three `MemorySource`s are provided which make use of external memory allocators:

- `MallocMemorySource` calls the standard C `malloc` function to allocate memory. This was used during early development, before the Boehm garbage collector was added. It is also used by some regression test programs that do not need or want garbage collection.

- `BoehmMemorySource` calls the Boehm conservative garbage collector [BW88, Boe02].

- `BoehmAtomicMemorySource` also calls the Boehm collector, but it marks the resulting memory chunks as being pointer-free (i.e. *atomic*). This is used for allocating arrays of non-pointer data. With the data marked as pointer-free, the garbage collector can avoid wasting time scanning potentially large areas of memory for pointers that do not exist, as described in Section 2.3.3.

The fourth allocator, shown on the bottom-left, is the `ArenaMemorySource`. It doles out chunks of memory from a given contiguous block, called an *arena*. It can be used to alleviate the load on the garbage collector for short-lived memory chunks by allocating them from a block that is managed by other means; e.g. by explicit freeing, or by placement on the stack.

Besides the allocators themselves, two additional `MemorySource`s appear in the diagram. They are shown between two `MemorySource` interfaces, since they both use and implement this interface: they are stackable building blocks that can be added to an existing memory source to provide additional functionality (as described in Section 2.3.2).

The first stackable `MemorySource` is the `ErrorMemorySource`, which reports a allocation error when called from specified points within the Jupiter source code. This is

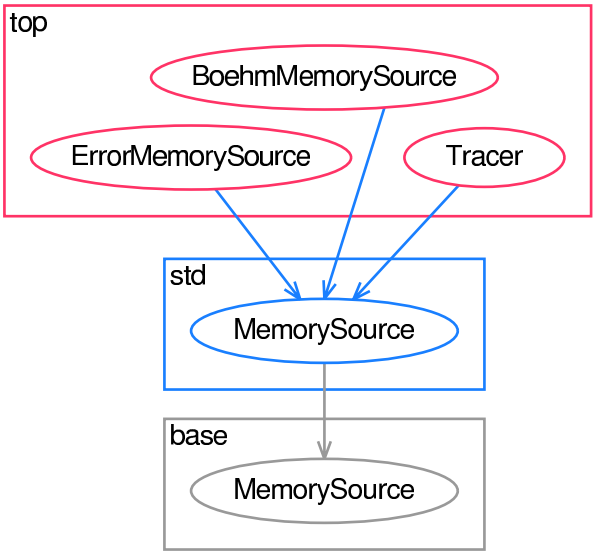

Figure 3.5: The memory allocation modules.

useful for regression-testing Jupiter's error handlers, by injecting errors into Jupiter that are otherwise difficult to reproduce. Error injection points can be specified with file-name/line-number pairs, or with function-name/line-number pairs. These are stored in hash tables that are consulted on every memory allocation to determine whether an error should be reported; if not, the request is simply delegated to the target `MemorySource` upon which the `ErrorMemorySource` is stacked.

Finally, the second stackable `MemorySource` is the `Tracer`, which 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. At desired points in the execution of a program, the `Tracer` does a conservative sweep of reachable memory, collecting statistics on memory usage that can aid diagnosis of memory leaks. The nature of the conservative sweep requires that the location of the heap is known, in order to determine whether a potential pointer is valid. `Tracer` achieves this by mandating the use of an `ArenaMemorySource`, which places the heap entirely within a known contiguous block of memory.

The `MemorySource` modules are shown in Figure 3.5. All the `MemorySource` modules have a dependency on the `std/MemorySource` module, which defines the generic jump table dispatch mechanism upon which they are all implemented.

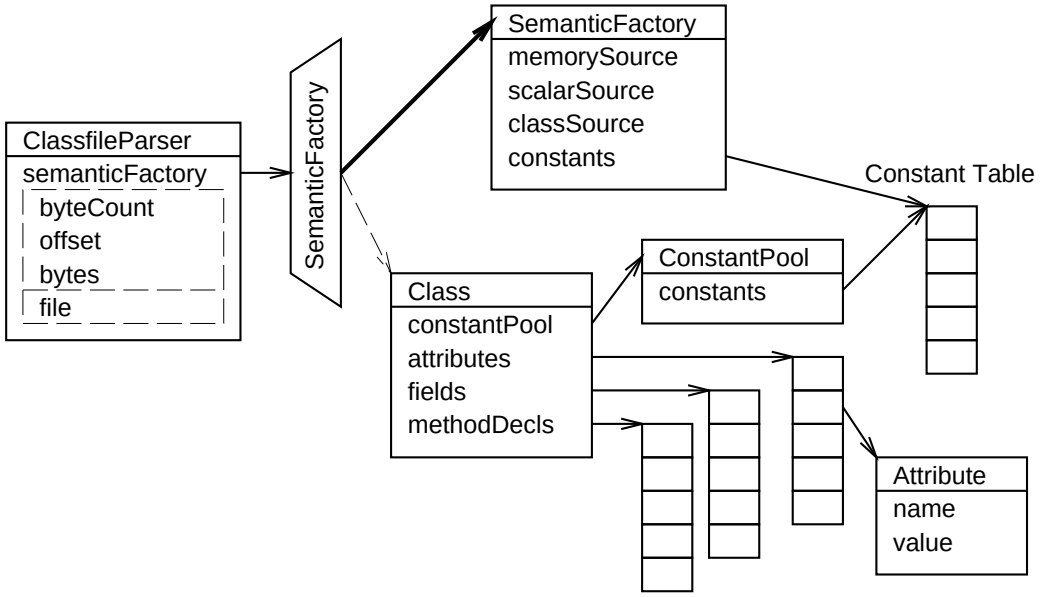

Figure 3.6: The classfile parsing objects. The metadata objects being constructed are shown underneath the SemanticFactory object.

3.3 Classfile Parsing

The `ClassfileParser` is a recursive-descent parser, generated automatically from a machine-readable classfile format specification by a tool called "Codegen" which produces C code [Doy02]. During the parsing process, a Jupiter *metadata object* is created to represent each construct parsed. The objects are created in postorder; that is, the object for each construct is created after the objects for its constituent parts. This guarantees that the parser always has a complete set of parts to pass to the object's constructor[1].

The objects involved in parsing a classfile are shown in Figure 3.6. At the left of the diagram is the `ClassfileParser` itself, which is used by the system whenever a new class is to be loaded. The `ClassfileParser` directs the construction of Jupiter's metadata according to the contents of a classfile, which may be located in memory or in a file.

To build each metadata object, the `ClassfileParser` calls upon the

[1]A few constructs have a back-pointer to their containing construct. Those are added after the containing construct is built.

`SemanticFactory` interface. While the `ClassfileParser` is responsible for deciding which objects to build, the `SemanticFactory` is responsible for how they are built. It provides a collection of *Factory Methods* [GHJV94] that the `ClassfileParser` calls to build an object for each construct found in the classfile. The `SemanticFactory` object that implements this interface is shown at the top of the diagram, and the metadata objects being constructed are shown underneath it. While the `ClassfileParser` is machine-generated from a classfile format specification to handle the complex, tedious, and error-prone aspects of parsing, the `SemanticFactory` is hand-coded and quite simple.

The `SemanticFactory` object has references to all the facility classes required to allocate the resources it needs to build the metadata. It has two `MemorySources`: one for scalar (non-pointer) data such as strings and arrays of bytes, and another for regular mixed data. This allows it, if desired, to take advantage of specialized facilities (such as the `BoehmAtomicMemorySource`) to allocate pointer-free data. It also has a `ClassSource` which is used to resolve superclasses and superinterfaces.

In addition, the `SemanticFactory` contains a reference to a constant pool in the form of an array of `Constant` objects we refer to as a *constant table* (shown on the far right of the diagram). When parsing first begins, the table is empty, and it gradually fills as the classfile's constant pool is parsed. Once the constant pool has been parsed, the constant table is used to assist with the remainder of the parsing process. Constant pool indices found in the classfile for things like field and method names are resolved by the `SemanticFactory` itself. This way, instead of passing a constant pool index to the `Field` or `MethodDecl` constructor, it can provide the resolved name string instead, which is generally much more useful. Note also that the `ConstantPool` metadata object that is eventually created to represent the constant pool itself is constructed rather economically from the same constant table used during the parsing process, as shown in the diagram.

When the classfile is parsed, its attributes are represented in metadata as `Attribute` objects, and no attempt is made to parse the data inside each attribute. This is because the structure of each attribute depends on the attribute's name. Checking the name

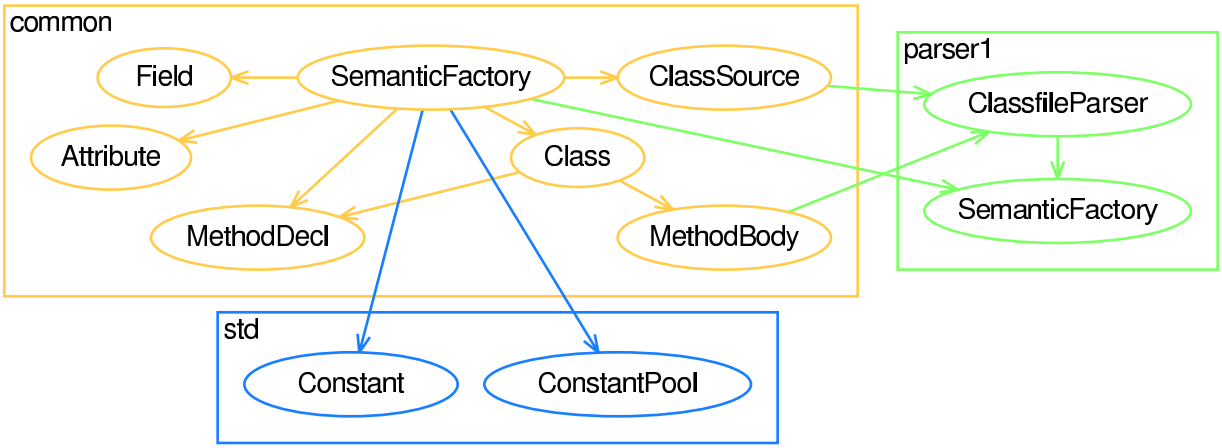

Figure 3.7: The classfile parsing modules.

of each attribute as it is parsed would complicate the parser, and would waste time inspecting attributes that may never be used by the JVM. Instead, the `ClassfileParser` interface simply provides functions for parsing attributes whose structure is known to the parser. After the parsing of the classfile is complete, individual attributes can be parsed as needed.

The modules that implement classfile parsing are shown in Figure 3.7. The most important thing to note regarding this diagram is that `ClassfileParser` has no dependencies outside the `parser1` directory, meaning that it is insulated from implementation decisions made in the rest of the system. It makes use of the `SemanticFactory` interface, which is implemented by a similarly named module in the `common` directory. This arrangement allows the parser to construct all the data structures that describe a classfile's contents without having any dependence on the data structures themselves: it is dependent only on the format of the classfile.

At the center of the `common` directory is a `SemanticFactory` implementation module which depends on most of the other modules in the diagram. It uses constructors from each of these modules to build the objects specified by the `ClassfileParser`.

Since `ClassSource` is the `base` interface responsible for managing classes, it is the primary client of the `ClassfileParser` interface. However, the `MethodBody` module also uses the parser's functionality to parse the `Code` attribute associated with each method. (The `Code` attribute is the one that contains the bytecodes for a method.)

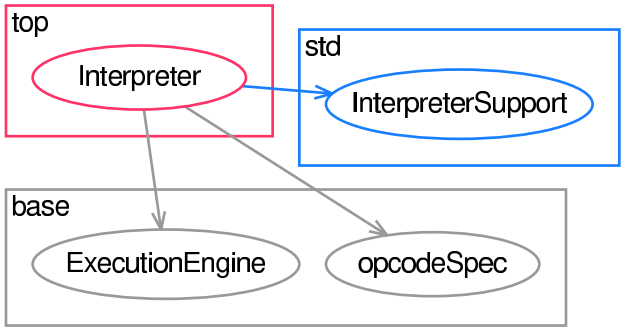

Figure 3.8: The bytecode execution modules.

This illustrates the value of a parser that can read both from files and from memory: while classfiles are usually read from files, attributes are parsed after they have already been read into memory by the classfile parsing process.

3.4 Bytecode Interpretation

The execution of the Java program is choreographed by the `ExecutionEngine`. It is the `ExecutionEngine` that determines the overall execution paradigm of the JVM, whether it be a simple interpreter loop, a threaded interpreter (described below), or a just-in-time (JIT) compiler.

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 3.8. These modules are each responsible for implementing a portion of the `ExecutionEngine` functionality:

- 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`" into the interpreter module. Since it depends only on the `base` interfaces, it is a very stable module, so it too resides in the `base` directory. 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. Being stable, though not truly fundamental, it resides in the `std` directory.

- The `Interpreter` module implements the `ExecutionEngine` interface, making use of the `opcodeSpec` and `InterpreterSupport` modules as necessary. This module changes relatively often—mainly to add or change mechanisms for debugging Java code, such as tracing facilities or statistics gathering—so it resides in the `top` directory.

The `opcodeSpec.h` header contains a piece of code for each Java opcode, describing how that opcode is implemented. Each piece of code is surrounded by macros, which can be defined by the including module to expand to whatever control structure is required. For instance, a typical interpreter would cause these macros to expand to `case` and `break` statements for inclusion inside a switch statement. This way, as each opcode is encountered, the switch statement will branch to the appropriate implementation code.

The current `ExecutionEngine` implementation is a *threaded interpreter*, meaning that, after executing one opcode, it branches directly to the code for executing the next opcode [GH01]. 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.

Since the target of each branch depends upon the opcode to be executed, the static `goto` facility provided by the C language is not sufficient. Hence, the direct branch is implemented using `gcc`'s *computed goto* facility, which allows branch labels to be treated as first-class objects. During initialization, the interpreter builds an array of the branch targets indexed by opcode number. To dispatch an opcode, the interpreter looks up the corresponding branch target from the array, and then performs a computed `goto` to jump to that branch target.

Having opcodes dispatched through a branch target table also allows them to be overridden to provide additional functionality. For instance, the current implementation makes use of this capability to help with bytecode substitution, modifying a method's bytecodes to contain new opcodes defined by Jupiter to perform the same task in a more

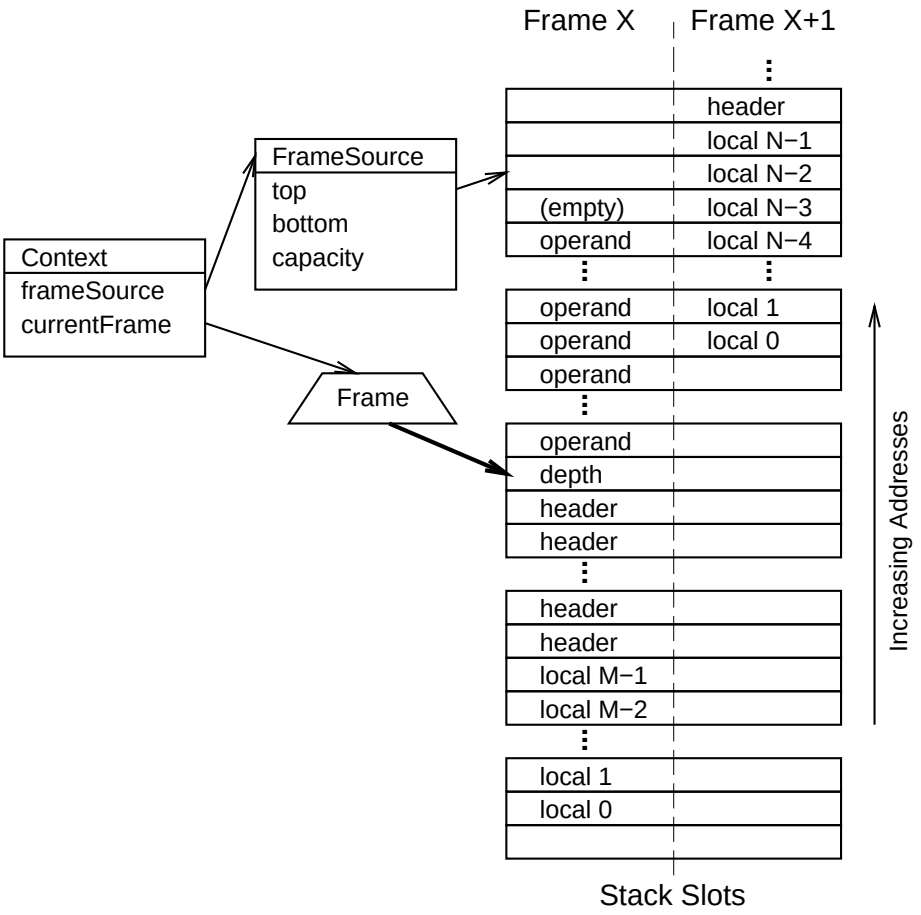

Figure 3.9: The stack layout. Each stack slot is labelled twice, for its role in the two overlapping frames. The slot marked "`(empty)`" is the portion of the operand stack space which does not currently contain data.

efficient manner. The details of the implementation are presented in Section 6.1.3.

3.5 Stack Layout

The Java execution stack consists of a number of *frames*, each of which holds data for use by a single executing method. The top frame is for the method currently executing, while the rest are for methods which are in progress, waiting for an `invoke` opcode to finish executing. This situation is modeled in Jupiter by the `Context` interface, which represents the execution stack as a whole, and provides access to `Frame` objects that represent individual stack frames.

The objects representing the execution stack are shown in Figure 3.9. 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 datatypes require two adjacent slots, as prescribed by the Java specification [LY99].

One object that is notably absent from this picture is the `Frame` object. The reason is that `Frame`s are not self-contained objects in the usual sense. A number of interface design techniques provide the illusion that `Frame`s are just like any other objects (see Section 4.1), but in reality, each `Frame` is stored in a contiguous group of slots within the slot array. A particular frame's slots contain the frame's operand stack, local variables, and a header that includes such things as the frame's class and method, its program counter, and its current operand stack depth. A `Frame` pointer actually points to the operand stack depth, which resides at the end of the frame's header. Knowing the location of the header, the locations of the rest of the frame's data can be computed. The operand stack depth was chosen as the target of the `Frame` pointer because, among other reasons, it is the most frequently accessed element of the frame. With the `Frame` pointer pointing directly to it, the stack depth can be accessed efficiently, with no pointer arithmetic.

When a Java method is called, a new frame is created, and the method's arguments are transferred from the caller's operand stack to the callee's local variables. Storing frames as contiguous groups of slots allows an important optimization to be implemented: the slots used by two adjacent frames can be overlapped. Specifically, the caller's operand stack is overlapped with the callee's local variables. This has the effect of transferring the argument values without the need to copy any data. The resulting layout of two adjacent stack frames is shown in the diagram, with overlapping slots labelled twice to indicate their role in each of the frames.

The `FrameSource` is responsible for allocating frames. It tracks the location of the first unused stack slot, so the location of the next allocated frame can be computed. The diagram depicts the state of the `FrameSource` after frame X has been allocated, but before frame X+1, with the `top` pointer pointing to the first available slot beyond

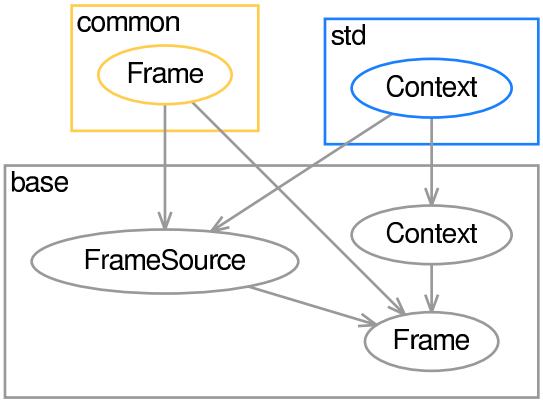

Figure 3.10: The stack-related modules.

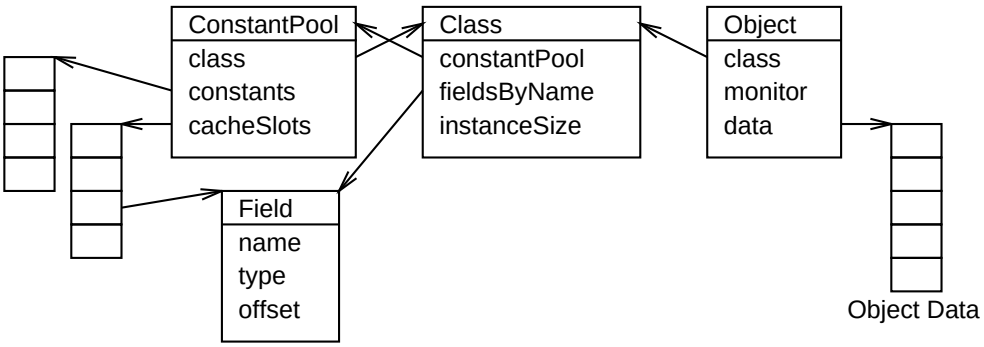

Figure 3.11: The Jupiter objects responsible for the layout of Java objects.

frame X's operand stack.

The modules that manage the execution stack are shown in Figure 3.10. The `Frame` and `FrameSource` implementations are both located in the `Frame` module (for reasons discussed in Section 4.4.1). Note the very low degree of coupling: both the `Frame` and `Context` modules depend only on `base` headers. Note, in particular, that `Context` depends only on the interface of `FrameSource`, not on its implementation. This makes `Context` a highly stable module, hence its placement in the `std` directory.

3.6 Object Layout

Jupiter's object implementation is shown in Figure 3.11. Accessing the fields of an object proceeds in two stages: first, a `Field` is acquired that represents the desired field; second, the `Field` is used to look up the field's value within the target object.

The structures for the first stage are shown on the left of the diagram, while those for the second stage is shown on the right.

At the outset, the system has only the object reference and the constant pool index. The index is passed to the `ConstantPool`, whose job is to find the `Field` which corresponds to that index. The first time a given field is accessed, the `ConstantPool` calls upon the `Class` to find a `Field` with the given name, using its `fieldsByName` hash table. The `ConstantPool` then caches the `Field` so subsequent accesses need not use the hash table.

Having found the desired `Field`, it is a simple matter to extract the value of that field from the target object. Every object has an associated data area containing the values of each field, and every `Field` stores the offset of that field within the data area. Knowing the field's offset, the value of that field can be extracted from the object's data area is a simple matter of array indexing.

Jupiter assigns offsets to the fields of each class when that class is loaded. Fields are arranged in such a way that the start of every object's data area is laid out as an instance of its superclass. This allows an object to be treated as an instance of any of its ancestor classes, as required by the Java specification [GJSB00]. When a class is loaded, its first field is assigned an offset equal to the `instanceSize` of the superclass, and subsequent fields are placed afterward at an offset determined by the preceding fields' sizes.

The current layout scheme is a simplistic one. Fields in the object are laid out in the same order in which they appear in the classfile. To ensure that each field is aligned as it should be, Jupiter simply aligns all fields to word boundaries, thus wasting some space within each object. More efficient schemes are possible; for instance, if fields were sorted in descending order of size, no padding would be necessary between fields to achieve the desired alignment. However, some padding would be needed between the between the last field of the superclass and the first of the subclass. Since this complicates the algorithm somewhat, and since object size has not been a problem thus far, we have chosen to keep the simpler scheme for the time being.

For arrays, the elements are not padded. Since they are all of the same size, aligning

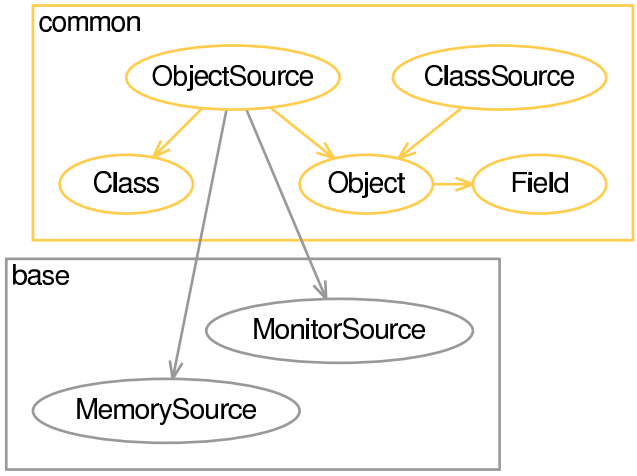

Figure 3.12: The object layout modules.

the first element automatically aligns the rest.

The modules involved in object management are shown in Figure 3.12. There are a number of relatively minor dependencies among the `common` modules:

- `ObjectSource` uses a constructor from `Object`, for obvious reasons.
- `ObjectSource` queries `Class` to find its `instanceSize`, in order to determine how much memory to allocate for a new object's data area.
- `ClassSource` calls upon the `Object` module to layout the fields of each new class as it is loaded. This encapsulates all field layout decisions within the `Object` module.
- `Object` uses `Field` to store its own offset.

None of these dependencies represents a serious breach of information hiding, since the leaked information is applicable to any scheme which allocates object data in a single contiguous block of memory. Given this relatively minor constraint, object layout decisions are completely encapsulated by the `Object` module. Even if a non-contiguous layout scheme were desired, only a handful of modules would be affected, and all but `Object` would require only trivial changes.

3.7 Method Lookup

To invoke a method, Java code issues one of the four `invoke` opcodes, all of which operate in a similar fashion: the instruction specifies a location within the constant pool that contains the name and type signature of the desired method. This information (collectively called the *method selector*) is used to locate the appropriate method body, which is then executed.

The method name and type signature are specified as strings. If every `invoke` instruction needed string manipulation to locate the appropriate method body, method invocation would be very slow indeed. In order to achieve acceptable performance, Java's method lookup rules are carefully designed to allow for much more efficient implementations: the string manipulation can be pre-computed, and the results can be stored efficiently in the constant pool, and in the class itself.

Jupiter's approach to dispatching a method is conceptually quite similar to looking up a field value (described in Section 3.6), though the implementation is more complex. Recall that field lookup first acquires a `Field` object from the `ConstantPool`, and then uses the `Field` to locate the field value. The two-stage process allows the `Field` object to be cached to avoid processing the field name on every access. Similarly, method dispatch first acquires a `MethodDecl` from the `ConstantPool`, and then uses the `MethodDecl` to locate the appropriate `MethodBody`. As with field lookup, this two-stage process allows the `MethodDecl` object to be cached to avoid processing the method selector on every invocation.

Jupiter's implementation is fairly typical: jump tables are built for each class, with one entry corresponding to each method selector, indicating which `MethodBody` is to be used for that selector. In the first stage, the constant pool maps the given selector to the corresponding `MethodDecl`, which contains the index of that selector's jump table entry[2]. In the second stage, the `Class`' jump table is accessed using the `MethodDecl`'s index to locate the desired `MethodBody`. After the first invocation of a particular selector, subsequent invocations no longer need to do any string manipulations: a method

[2]The `MethodDecl` also contains enough information to determine which jump table to use. This is important for interface methods, each of which has its own jump table.

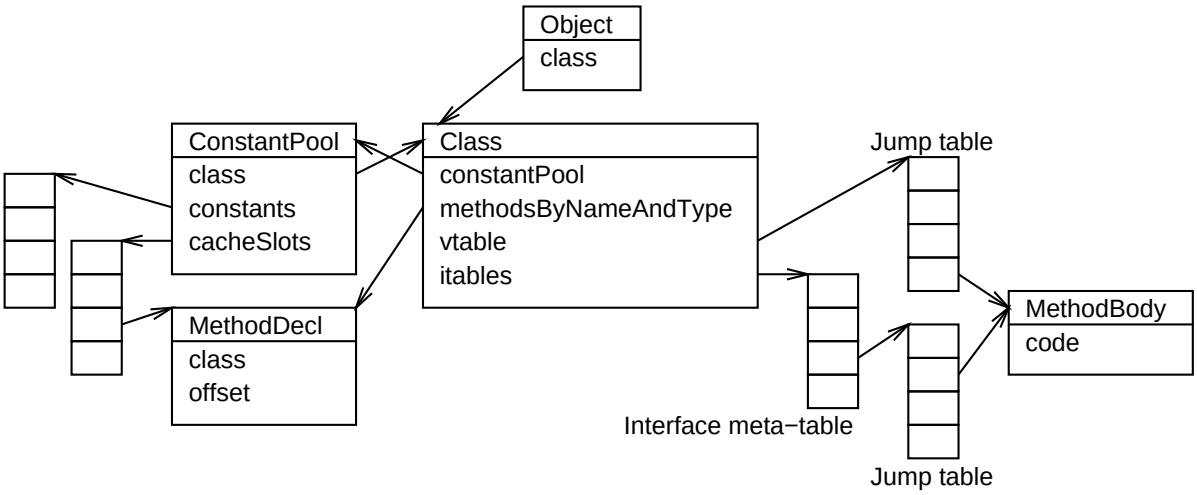

Figure 3.13: The method lookup objects. Lookup proceeds first as shown on the left to acquire a MethodDecl, then on the right to acquire a MethodBody. Note the similarity of the left side with that of Figure 3.11.

lookup is reduced to a pair of array lookups, first to get the `MethodDecl` from the `ConstantPool`, and then to get the `MethodBody` from the jump table.

The first stage is independent of the position of the target object's class within the inheritance hierarchy: it depends only on the declared type of the variable through which the method was invoked. The second stage is the one responsible for *dynamic dispatch*: it returns the `MethodBody` appropriate for particular run-time class of the target object.

The data structures involved in this process are shown in Figure 3.13. The structures for acquiring the `MethodDecl` are shown on the left, while those for acquiring the `MethodBody` are on the right.

The first step toward invoking a method on an object is to retrieve the `ConstantPool` for its class. Next, the constant pool index supplied by the `invoke` instruction is passed to the `ConstantPool`, whose job is to return the appropriate `MethodDecl` for that index. The `ConstantPool` already has the name and type signature for each index, since that information was supplied during the classfile parsing process. However, the task remains to find which `MethodDecl` corresponds to that name and type signature.

The first time a particular index is used, the `ConstantPool` consults the `Class`, which uses its `methodsByNameAndType` hash table to find the appropriate `MethodDecl`.

That `MethodDecl` is then cached by the `ConstantPool` so that it can subsequently return the `MethodDecl` directly given only the constant pool index.

Acquiring the `MethodDecl` marks the end of the first stage of the method invocation. The operation of the second phase depends on whether or not the method is being invoked through an interface class (by `invokeinterface`) or a regular class. For regular classes, Java supports only single inheritance, so all method lookup can be achieved by using a single jump table for each class, known as the *vtable*. As with field layout, Java's single inheritance makes vtable layout simple: each vtable starts with a copy of the vtable of the superclass (with the appropriate entries overridden), allowing any class to be substituted for any of its ancestor classes.

In contrast, Java supports multiple inheritance for interfaces, so a more elaborate scheme is required. Jupiter's scheme uses a two-level table system which we refer to as the *itables*. The first level is the *interface meta-table*, which contains one entry for each interface class in the system[3]. This is a sparse table, since each class implements only a subset of all the interfaces in the system; however, the total memory occupied by these tables is still not very large, though certain optimizations are possible if the sparseness is found to be problematic [SW01]. Entries corresponding to interfaces that are not implemented are left null, while the other entries point to the second-level tables. The second level tables are much like vtables: they have one entry for each method selector in the corresponding interface.

It is interesting to note that the the `itables` scheme allows for a very fast interface conformance test. It is only necessary to check the `itables` entry corresponding to the interface in question; that entry will be non-null if and only if the class conforms to that interface. In contrast, regular class conformance is achieved by walking up the inheritance hierarchy and checking whether the given class is encountered. This leads to the somewhat ironic situation that the conformance test is actually faster for interfaces than for regular classes, despite the difficulties introduced by multiple inheritance.

The Jupiter modules involved in method lookup are shown in Figure 3.14. Most of the dependencies are harmless `base` dependencies. Note in particular that

[3]Actually, it contains one entry for each interface that was loaded at the time the class was created. The `itables` are *not* needlessly enlarged every time new interface classes are loaded.

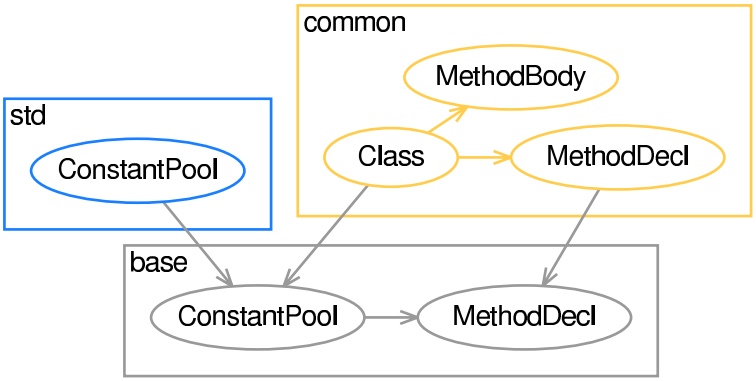

Figure 3.14: The method lookup modules.

`ConstantPool` and `Class` are completely independent. This allows programmers to modify or rewrite `Class` without concern for the implementation of `ConstantPool`, which is well-defined by the Java standard and relatively stable—hence its placement in the `std` directory.

The `Class` module is dependent on the `common` implementations of `MethodDecl` and `MethodBody`, though these dependencies are not particularly strong ones. `Class` requires each `MethodDecl` to store its own jump table offset; this reveals only that jump tables are being used, and is not a very worrisome breach of information hiding. `Class` also needs special access to `MethodBody` because it is the `Class` module which creates the `MethodBodies`. The classfile parser creates only `MethodDecls`, and the `Class` then takes every `MethodDecl` without the `ACC_ABSTRACT` attribute and constructs a `MethodBody` from it. This reveals only that `MethodBodies` can be constructed from non-abstract `MethodDecls`, which again is not a worrisome breach of information hiding. Thus, the coupling in this scheme is fairly low.

3.8 Conclusion

This chapter has provided a tour of Jupiter's implementation, describing the data structures and algorithms, as well as the modules and their dependencies. These implementations are by no means the only ones possible; on the contrary, the purpose of Jupiter's design is to facilitate future research aimed at improving the state of the art of JVM implementation. However, by describing a typical implementation of each module and

interface, we hope to have clarified their role in the system, to provide a backdrop for the discussions in the remainder of this document.

Chapter 4

Design for Flexibility

Flexibility is the ease with which a system can be adapted to a variety of requirements with a relatively small investment of effort. A programmer should have the ability to conceive an idea, and then implement it with an amount of effort commensurate with the perception of the complexity of the modifications involved. Jupiter uses a building-block architecture that has the potential for tremendous flexibility, but a flexible architecture alone does not make for a flexible system. In this section, we present a number of techniques used in the design of Jupiter's modules which allow the flexibility of the building-block architecture to reach its full potential.

It is too much to ask that a system could be so flexible as to make every possible modification simple. A more realistic goal is that the programmer should not be taken by surprise by the amount of effort required to implement an idea. In order to achieve this goal, a design should possess the following properties:

- *Simplicity.* The modules of the system should have the simplest possible design, to allow the programmer to construct an accurate mental model of the system with which to estimate the effort involved in performing a modification.

- *Orthogonality and information hiding.* System functionality should be partitioned and encapsulated behind independent interfaces. This reduces the chances that modifying one part of the system would necessitate further unexpected modifications to seemingly unrelated parts.

- *Reliability and robustness.* Though it may appear to work properly, a system of unreliable components has the potential to fail under novel conditions in which they have not been tested, requiring additional unexpected effort to make a new idea work.

The desire to provide Jupiter with these properties has affected its design in innumerable ways, both dramatic and subtle. While it would be impossible to describe every design decision that has contributed to Jupiter's flexibility, this section presents a sampling of five techniques which have been used with great success repeatedly throughout the system. The techniques are the following:

- *Interface coding conventions.* Careful consideration of what information to hide and provide makes Jupiter's interfaces flexible and understandable.
- *Design by Contract.* The focus on the clear delineation of responsibilities among modules makes the system more robust and simpler to comprehend.
- *Splitting over-constrained interfaces.* A single problematic interface is replaced by two, each of which concentrates on fewer criteria, thus making it easier to achieve a good design.
- *Modularizing by maintenance characteristics.* Code is divided into modules based on the circumstances under which it may need to be changed, thus focusing the effort of modification on a well-delineated set of modules.
- *Pervasive error handling.* Jupiter's error handling idiom makes robust error handling simple, thus encouraging its disciplined use throughout the system.

Some of these techniques achieve flexibility by simplicity, some by orthogonality, some by reliability, and so on. Most techniques simultaneously provide several of these properties, and so we have not attempted to subdivide the section by property. Instead, we present each technique, describe its effects, and provide a number of examples.

One example which recurs throughout this chapter is that of the Java execution stack, modeled by the `Context`, `Frame`, `FrameSource` interfaces. The stack is accessed

by almost every opcode executed, sometimes several times. Hence, any inefficiency will be multiplied enormously during the execution of a program. Furthermore, because the stack's efficiency is so important, the interfaces must allow for a variety of implementations in order to explore alternative trade-offs. Thus, of all the interfaces in the system, the execution stack epitomizes the design tension between flexibility and performance.

Being "soft" requirements, such qualities as flexibility and simplicity are difficult to quantify, unlike "hard" requirements like performance and efficiency. However, we shall present arguments that each technique provides a substantial, tangible benefit.

4.1 Interface coding conventions

Interfaces form the skeleton of any modular system, so their design is crucial to the system's flexibility: they must be crafted carefully to avoid placing unnecessary limitations on implementors. To address this concern, a number of coding idioms have been employed in the construction of Jupiter's header files, in order to present the least possible impediment to flexibility. In this section, we begin by presenting a typical C interface coding style, and discuss its shortcomings. We then proceed to modify the style incrementally to address the shortcomings as we encounter them, until ultimately we arrive at the style used by Jupiter.

In C, object references are implemented using pointers: references are passed from one function to another as pointers, and they are stored inside other objects as pointers. The common C idiom for achieving this uses syntax like the following:

```
/** (Something.h) **/
struct something{
  int a, b, c;
};

/** (Caller) **/
void some_function(struct something *s){
  s->a = s->b + s->c;
}
```

The lack of encapsulation associated with this idiom makes reasoning about the program's correctness difficult, since any structure could be easily modified (perhaps accidentally) by any part of the system.

To provide the required encapsulation, we make use of an idiom known as *opaque data structures* by which the contents of the struct are not declared in the header file. Instead, the header file provides only accessor function declarations. The struct definition is placed inside the module that defines the accessor functions, so only that module can access the struct directly. All other code must make use of the accessor functions.

The resulting header file appears as follows, along with a sample caller:

```
/** (Something.h) **/
struct something_struct;
int something_a(struct something_struct *s);
int something_b(struct something_struct *s);
int something_c(struct something_struct *s);
void something_set_a(struct something_struct *s, int new_value);

/** (Caller) **/
void some_function(struct something *s){
  something_set_a(s, something_b(s) + something_c(s));
}
```

This provides better encapsulation, and has the additional benefit that all struct accesses use C's function call syntax. Since the function call syntax is identical to macro call syntax, we can rest assured that, if necessary, we could replace one of the function declarations with a macro definition. Thus, there is nothing fundamentally standing in the way of eventually making this interface as efficient as necessary, without altering the call sites.

This idiom, however, is quite verbose. To begin with, it is helpful to define a typedef for the struct declaration. In addition, the Jupiter coding style assigns a prefix abbreviation to each kind of object. Specially-formatted comments are added to the header file in which the corresponding typedef appears, so that utility scripts can keep track of the prefixes that have been assigned. In the case of the `Something` example, we might assign the prefix "`st`" to the `Something` struct. The code that results from the incorporation of these idioms looks like this:

```
/** (Something.h) **/
/* Prefix: st=Something */
typedef struct st_struct Something;
int st_a(Something *st);
```

```
int st_b(Something *st);
int st_c(Something *st);
void st_set_a(Something *st, int new_value);

/** (Caller) **/
void some_function(Something *st){
  st_set_a(st, st_b(st) + st_c(st));
}
```

At this point, we have achieved an idiom which provides good encapsulation without being overly verbose. However, there are still some flexibility problems with this approach. Note that every reference to a `Something` must explicitly be declared as a pointer. This makes it impossible to implement schemes in which an object reference is not a pointer, without altering the interface. For example, following modifications to the system may prove to be desirable, but would be difficult to implement using the above interface idiom:

- If the object is immutable, we may choose to pass it by value. If explicit pointer syntax is used throughout the system, this change would be impossible without altering all the code that manipulates that object type.
- If a system were designed in which Jupiter objects were permitted to reside in different address spaces (say, on different nodes in a cluster), then references to these objects would need to be something other than a simple pointer to the struct.

To address these issues, the pointer declarations are "hidden" inside the typedefs. All code that manipulates objects is written with pass-by-value syntax, but with pass-by-reference semantics[1]. This results in the final interface coding idiom—the one actually used by Jupiter—which would appear as follows:

```
/** (Something.h) **/
/* Prefix: st=Something */
typedef struct st_struct *Something;
int st_a(Something st);
int st_b(Something st);
int st_c(Something st);
```

[1]In this respect, the code looks and behaves more like Java than like traditional C.

```
void st_set_a(Something st);

/** (Caller) **/
void some_function(Something st){
  st_set_a(st, st_b(st) + st_c(st));
}
```

Having hidden the pointer declaration inside the `typedef`, all code that uses an object need not be aware of the mechanism by which references to that object are passed around the system. The mechanism can be changed (for instance, from pointers to some sort of handle) without rewriting all the code that manipulates the object.

These coding conventions are designed to provide the utmost flexibility, not only for the implementation of each interface, but also for the interfacing mechanisms themselves. The choice between function and macro, pointer and handle, value and reference, have all been left to the implementor, due to the high degree of information hiding these conventions achieve.

4.2 Design by Contract

Jupiter has been influenced heavily by the notion of *Design by Contract*, a methodology developed by Bertrand Meyer and supported directly in his Eiffel programming language [Mey88]. This approach stands in contrast to defensive programming, a philosophy advocating that robustness be achieved by having each function in the system check that it has been invoked with legal arguments, and report an error otherwise. Instead, Design by Contract prescribes that each function be associated with a *contract*, which consists of a precondition and a postcondition[2]. The semantics of the contract require that if the caller satisfies the precondition before calling the function, then the function is responsible for satisfying the postcondition.

If the caller does not satisfy the precondition, then the caller is considered to be in error, and the callee is absolved of responsibility. Specifically, the callee has no

[2]Design by Contract also includes other assertions, such as class and loop invariants, plus rules that apply the appropriate contracts in the presence of polymorphism. Since Jupiter has shallow inheritance hierarchies and simple loops, these facilities were not needed.

responsibility to detect or report the error condition: rather, the behavior of the callee is undefined.

Allowing undefined behavior may seem to be contrary to the goal of robustness; however, just the opposite is true. Since the responsibilities of the caller are stated quite clearly in the function's precondition, it is fairly straightforward to make sure that the precondition is satisfied before calling the function. In addition, having decided to use contracts as the basis of the system's interface design paradigm, we then design functions to have the simplest possible contracts, thus making the preconditions even easier to obey.

In practice, a broken contract does not wreak undefined havoc in the system. Instead, the implementation of each function contains `assert` statements to check that its contract has been obeyed. Contract breaches are invariably considered bugs, and the system is not considered correct until it is written such a way that no contract, and hence no `assert` statement, is ever broken. In theory, as the program is debugged, the number of contract breaches should fall asymptotically toward zero[3]. Once a sufficient level of confidence in the correctness of the program is achieved, assertions can be deactivated, and performance is not impeded by unnecessary error checking.

The primary advantage of Design by Contract is that when the program fails, the responsibility for the error is assigned unambiguously to a single function; namely, whichever function it was that broke a contract. That function must be repaired by modifying either its implementation or the contract so that the two match each other.

The main disadvantage of using preconditions instead of error checking is a matter of convenience: each precondition represents additional responsibilities that have been delegated to the caller, making the function less convenient to call. For instance, if the function does not take responsibility for error checking, then error checking logic must be added at every call site. However, in many cases, even this is not significant, since the caller had to handle the error condition anyway, even if the detection itself is done by the callee.

[3]In the case of Jupiter specifically, it did not take long before contract breaches became extremely rare; in practice, any contract breaches found in Jupiter in an ongoing basis occur in newly-added code.

The question of whether to use preconditions rather than error checking has no single answer that applies in all cases: the choice must be made for each function in the system. This section presents two examples which illustrate the trade-offs involved: local variable and operand stack access in the `Frame` interface, and array element access. In each case, the design has been chosen which maximizes the flexibility of the system.

4.2.1 Local variable/operand stack access

The typical, straightforward use of contracts is exemplified by the functions that provide access to the local variables and operand stack. Through the `Frame` interface, the local variables of each stack frame can be accessed by index, counting from zero up to some known maximum. Similarly, the operand stack can also be accessed by index, counting from one up to the current stack depth. In both cases, only valid indices should be used. The goal is to choose an approach, whether by contract or by error checking, that ensures that invalid indices are not used.

The choice of approach for this case is fairly straightforward, since valid Java code never attempts to access an out-of-bounds local variable or operand stack entry: such code would fail verification. An invalid index is a sign of a bug in Jupiter, for which an assertion failure is appropriate. Hence, we chose to enforce index bounds using preconditions. The local variable and operand stack accessor functions simply require that the index is valid, rather than report an error for invalid indices.

Note that it is simple and efficient to implement an error-checking version of an accessor function that makes use of the precondition version, but not vice versa. Thus, the precondition version is flexible enough to allow for both schemes to be implemented if needed, while the error-checking version is not.

In this example, being simultaneously more flexible and more efficient, the advantages of Design by Contract are clear.

4.2.2 Array element access

Another example of the use of contracts in Jupiter is the array element accessor functions. This example demonstrates that the policy used by Jupiter's interfaces need not

be dictated by Java semantics: Jupiter's interfaces can use a precondition even when Java requires explicit error checking.

Java specifies that a particular exception be thrown if an array index is out of bounds. The natural way to model this would be for Jupiter's `Array` interface to check array indices and report an error when they are out of bounds. However, Jupiter's `Array` accessor functions take the opposite approach: they have the precondition that all indexes must be valid. As pointed out in the preceding examples, it is simple and efficient to implement the exception approach using the precondition approach, but not vice versa. Thus, Java's semantic requirements can be easily accommodated. Furthermore, the JVM's own internal accesses to array elements are guaranteed never to be out of bounds, so checking bounds inside the `Array` interface would be unnecessary and wasteful.

The use of preconditions in Jupiter's `Array` interface allows for an efficient implementation, while still allowing Java's specifications to be met with little difficulty. It achieves both with only one set of accessor functions, thus keeping the interface simple, and enhancing the system's flexibility.

4.3 Splitting over-constrained interfaces

An interface is the means by which pairs of modules communicate. Any module which is on either side of the interface—both clients and implementors—may impose certain requirements on that interface, and an ideal interface is one which meets all its requirements. In some situations, the modules may impose too many requirements on the interface, and no ideal solution exists. In such cases, the interface is *over-constrained*, and an over-constrained interface causes one or more design goals to be compromised.

For instance, the interface for accessing the operand stack presented in Section 5.1.3 describes how bottom-based indexing is more efficient than top-based. However, for most code that accesses the operand stack, top-based indexing is far more convenient. Thus, while the efficiency criterion rules out the top-based scheme, ease-of-use rules out the bottom-based one. In this way, the interface is over-constrained, and no solution exists which meets all design criteria with a single interface.

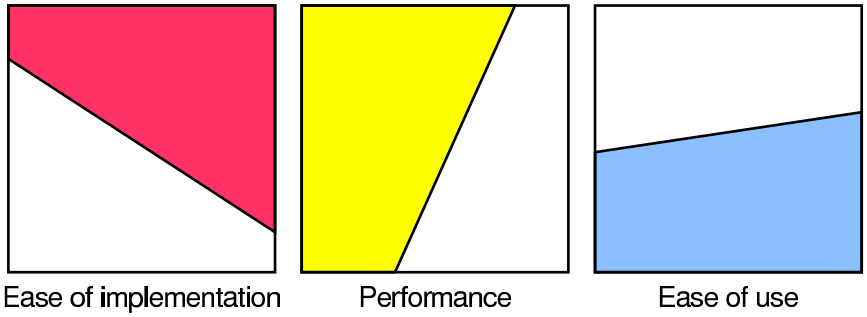

Figure 4.1: Constraints on the interface design space. Each coloured area represents designs that are ruled out by the corresponding constraint.

The solution we present in this section is to split the interface into two. In the case of the operand stack, two accessor functions are provided: the main function called `fr_operand` uses bottom-based indexing, and a helper function called `fr_topOperand` uses top-based indexing. The `fr_topOperand` function translates the top-based index into bottom-based format and passes it on to `fr_operand`. In this way, both performance and ease-of-use can be achieved: in most cases, the `fr_topOperand` helper function would be used, but when performance is important, `fr_operand` can be called directly. By splitting one interface into two, we have relieved each interface of one constraint, allowing solutions to be found for each interface separately.

Figure 4.1 depicts interface constraints graphically. The figure shows three design criteria: ease of implementation, performance, and ease of use. These three particular criteria are common ones for many of Jupiter's interfaces. For each criterion, the universe of possible interface designs represented as a plane. Each criterion rules out certain designs, represented by the shaded portions of the plane. When the criteria are superimposed, as shown in Figure 4.2, it is often the case that there remains no part of the design space left unshaded, and hence no design can possibly meet all three criteria simultaneously. Under these conditions, any design decision will necessarily be a compromise, causing the system to fall short of its design goals.

To address this issue, an over-constrained interface can often be replaced by two separate interfaces, as shown in Figure 4.3. The two new interfaces are separated by an additional insulating module that serves to free the interfaces from each other's constraints. In this example, the low-level interface need not be constrained by ease-of-use

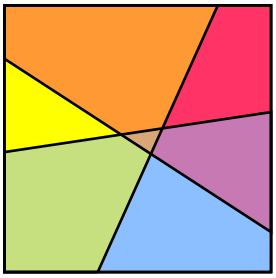

Figure 4.2: An over-constrained design space. Every possible design breaks at least one of the constraints.

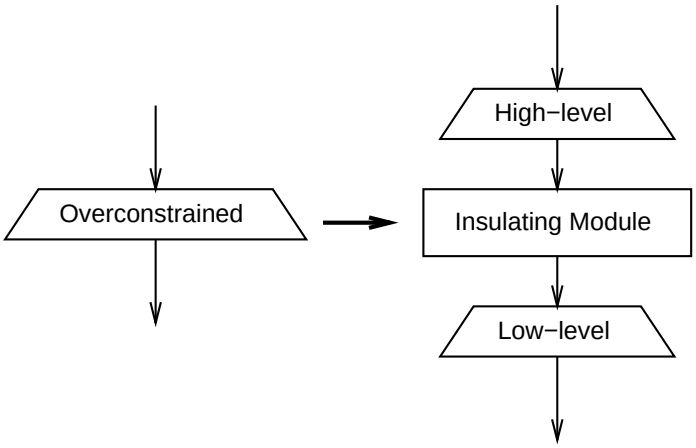

Figure 4.3: An over-constrained interface can be replaced by two less-constrained interfaces separated by an insulating layer.

considerations, because the insulating code will act as a wrapper that provides ease-of-use to the high-level interface. Likewise, the high-level interface need not be constrained by ease-of-implementation considerations, because insulating code can contain any logic required to reduce the complexity of the underlying implementation. Having removed one design constraint from each interface, solutions can now be found that do not compromise any of the design goals, as shown in Figure 4.4. Note that both interfaces are still constrained by performance; hence, interface splitting does not imply performance degradation.

As a bonus, the insulating code between the two interfaces tends to be very stable, since it usually depends only on the two Jupiter base interfaces that it splits. Insulating code often converges asymptotically toward an ideal implementation that seldom

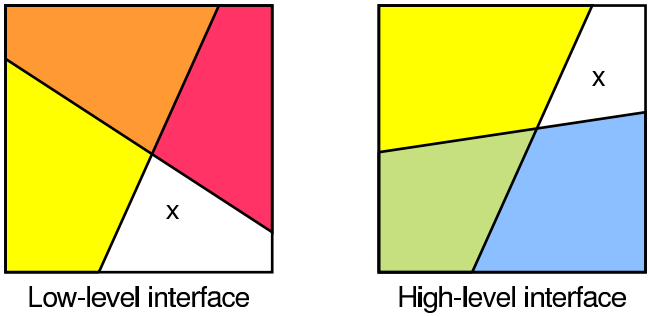

Figure 4.4: Design constraints after interface splitting. Each interface is relieved of one constraint, permitting an uncompromised solution (shown by the "x").

changes[4].

The same splitting technique can be successful with other design trade-offs besides ease-of-use versus ease-of-implementation, though that is the most common application of the technique in Jupiter. It is most effective with "soft" requirements, such as interface simplicity, flexibility, or ease of use, though it is also sometimes helpful with "hard" requirements like performance and efficiency.

To illustrate the benefits of splitting over-constrained interfaces, we present number of examples of how this technique has been employed in the design of Jupiter.

4.3.1 Context versus FrameSource

The `Context` and `FrameSource` interfaces provide a typical example of the benefits of interface splitting. Originally, the `Context` interface was all that stood between the user and the implementation of the Java call stack, which caused `Context` to become over-constrained. Adding the `FrameSource` interface relieved the constraints and allowed for an uncompromised solution.

The constraints on the original `Context` interface are exactly those shown in Figure 4.1. The simultaneous of ease-of-use and ease-of-implementation could not be satisfied simultaneously while still providing acceptable performance. Any design that made `Context` easy to use did so by burdening the implementation with responsibilities, while

[4]This phenomenon has occurred numerous times in the development of Jupiter, and such "convergent" modules are moved into the `std` directory when they are discovered, in order to indicate their stability.

any design that made `Context` simple to implement did so by exposing complexity to the caller. For example, if `Context` takes responsibility for tracking the topmost stack frame, that complicates its implementation; if not, that responsibility is left to the caller, making the interface more complicated to use.

To relieve this design tension, the `FrameSource` interface was introduced as a low-level interface to the call stack implementation, while `Context` became the high-level interface for clients that use call stack functionality. The `Context.c` implementation module acts as the insulating code in this case, implementing the `Context` semantics in terms of the `FrameSource` interface, thus relieving each interface from the constraints of the other.

The `Context` and `FrameSource` interfaces have a number of differences that illustrate their distinct design constraints:

- Stack manipulation almost always makes reference to the topmost frame, so `Context` keeps track of the current topmost frame for ease-of-use. In contrast, `FrameSource` does not keep track of the current frame, thus removing this burden from the implementation. Tracking of the topmost frame is handled by the `Context.c` module.

- When a frame is removed from the stack, that is usually because a method has returned, so the return value should be propagated to the caller's frame. `Context` takes responsibility for this operation in order to simplify client code. `FrameSource` does not, thus simplifying the implementation code. Copying the return value to the caller's frame is done by the `Context.c` module.

- Frames are added to and removed from the stack when methods start and finish executing. If the method is synchronized, the appropriate monitor must be locked and unlocked. The `Context` interface takes responsibility for this to simplify the client, while the `FrameSource` interface does not. The `Context.c` module performs the actual monitor operations.

As shown in Figure 4.4, the efficiency constraint applies to both the `Context` and

`FrameSource` interfaces[5]. Because of this, the two interfaces have some things in common; for instance, both take responsibility for copying method arguments when a new frame is pushed on the stack. It may have simplified the implementation of `FrameSource` if this requirement were removed, and the argument copying were performed by the `Context.c` module. However, this design would be unable to exploit the efficiency of the (very common) stack layout technique that causes adjacent frames to overlap to make argument copying unnecessary (as described in Section 3.5). Such a design would correspond to a point somewhere in the upper-left area of the design space, being ruled out by the efficiency constraint.

4.3.2 Threading Interfaces

Jupiter's interfaces to the threading facilities demonstrate how the demands on an interface can be especially constraining when they are imposed by forces outside the control of the system. The threading interface is one place within the JVM that is constrained by two different external forces: the semantics required of Java threads, versus the capabilities provided by the underlying thread library.

Using a single interface between these two external forces would cause that interface to become over-constrained, and finding one interface which cleanly serves both purposes would be difficult if not impossible. Whenever Java's requirements differ from the functionality of the underlying library, we are presented with three sub-optimal choices regarding the design of the interface: it could resemble Java's design rather than the library's, making it harder to implement; it could resemble the library's design, making it harder to use to meet the Java specification; or it could be a compromise, making it harder both to implement *and* to use.

To address this design tension, Jupiter contains two threading interfaces, shown in Figure 4.5. The higher-level interface consists of the `Thread` and `Monitor` abstractions, which implement the semantics of Java threads and monitors. The lower-level interface consists of the `ThinThread`, `Mutex`, and `Condition` abstractions (known collectively as the "`ThinThreads`" interface) which encapsulate the underlying thread library. The

[5] As it does with every interface in Jupiter.

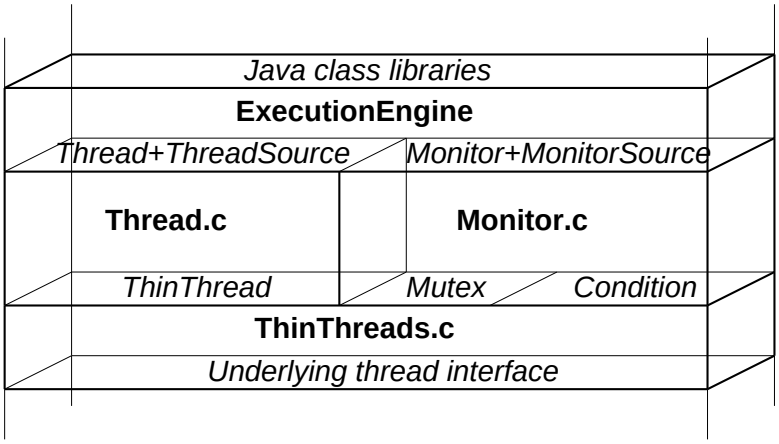

Figure 4.5: Multithreading modules and interfaces. Modules are shown as blocks divided by horizontal planes representing interfaces.

interfaces are separated by the `Thread.c` and `Monitor.c` modules.

These two interfaces are complementary 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.

The details of the threading interfaces are still under construction at the time of writing [Cav02]. However, it is expected that, once finished, they will serve as an excellent example of the tremendous flexibility benefits afforded by the interface splitting technique, since it is these interfaces that originally motivated the technique.

4.3.3 ObjectSource versus MemorySource

Most interface splitting produces a low-level interface with some desirable characteristics, and a high-level interface designed to be used by the rest of the system. However, Jupiter's object allocation system demonstrates how the low-level interface (namely `MemorySource`) that arises as a product of interface splitting may itself become as useful and important as the high-level interface.

One of the earliest design tensions to become apparent involved the memory allocation subsystem. Java programs can only use memory in the form of objects, so the allocation interface came to be known as `ObjectSource`. The design tension arose from the highly formalized view of memory that objects represent. Specifically, memory in Java is not just an unstructured array of addressable storage locations; rather, Java objects possess a great deal of additional semantics which are nontrivial to implement, such as field layout, synchronization via monitors, distinguishing arrays from non-array objects, etc. As a result, the creation of an object involves more than just the allocation of memory. Thus, the requirement to keep object allocation easy for the caller, while simultaneously keeping the memory allocator simple, would cause the single `ObjectSource` interface to become over-constrained.

The solution is to provide two interfaces, `ObjectSource` and `MemorySource`, and to implement an insulating layer between them, in the form of the `Object.c` and `ObjectSource.c` modules. The `ObjectSource` and `MemorySource` interfaces differ in a number of ways:

- It is convenient to have the quantity of memory required to hold a new object computed automatically, so `ObjectSource` provides this functionality. In contrast, `MemorySource` requires the caller to perform this computation, thus relieving the memory allocator of the responsibility.

- The fields of a new object instance are required to be initialized with zeros, and `ObjectSource` does this. `MemorySource` provides no guarantees regarding the contents of the memory blocks it provides, thus allowing implementations to do whatever is easiest and/or most efficient.

- Objects that are arrays are treated differently from objects that are instances of non-array classes. `ObjectSource` is aware of these differences, and takes responsibility for allocating and initializing both kinds of objects properly. `MemorySource` remains unaware that any such distinction exists.

Normally, having split memory allocation into two interfaces, `ObjectSource` would be the interface used by the system, while `MemorySource` would be seen only by memory allocator implementations. However, while the Java code only needs memory in the form of objects, regular memory allocation is, of course, indispensable for the JVM itself. Thus, the `MemorySource` interface is used widely to allocate memory throughout Jupiter.

4.4 Modularizing by maintenance characteristics

Information hiding has long been established as the most effective basis for decomposing a system into modules [Par72]. However, this prescription still leaves room for refinement, since it is not always clear what information should be hidden by which modules. In this section, we supplement this venerable module design criterion by considering the manner in which the information in each module may change over time, which we refer to as its *maintenance characteristics*.

The principle of module cohesion prescribes that a change to the system should require a small number of modules to be modified; preferably, only one. This suggests that code which changes under the same circumstances should appear in the same module. The complementary principle of module independence prescribes that separate changes should affect separate modules. This suggests that code which changes under different circumstances should appear in different modules. Together, the effect of these principles is that module boundaries should largely be determined by the code's maintenance characteristics.

In this section, we present two examples of module design based on maintenance characteristics. By addressing the flexibility criteria (atomicity, independence, and cohesion), this technique directly contributes to the flexibility of the system.

4.4.1 Frame and FrameSource

Jupiter's interfaces for allocating and manipulating stack frames illustrate how code with similar maintenance characteristics should appear in the same module.

The `Frame` interface are designed to insulate the rest of the system from the details of each frame's memory layout. Similarly, the `FrameSource` interface is designed insulate the rest of the system from the collective layout of all the frames on the stack. The layout of a single stack frame is very likely to affect and to be affected by the layout of the stack as a whole. This is especially true because it is common to lay out the stack in such a way that adjacent frames overlap to make argument passing more efficient; this trick only works if individual frames are designed to allow for it.

Since the layout of the stack as a whole is so tightly coupled to the layout of the individual frame, it is unlikely that they could be modified independently. Hence, combining the implementations of `Frame` and `FrameSource` into a single module is unlikely to harm independence or atomicity, while it greatly improves cohesion. Hence, the two implementations have been situated in a single module.

4.4.2 Native versus NativeSource

Jupiter's interfaces for accessing and invoking native methods illustrate how code with different maintenance characteristics should appear in different modules.

The invocation of native methods proceeds in two steps: first, a `NativeSource` is used to acquire the `Native` which implements a given `MethodBody`; second, the `Native` is invoked using its `nv_invoke` function. The `NativeSource` step is implemented by looking up the native code using whatever introspection capabilities are provided by the system (e.g. `dlsym` in Unix, `getprocaddress` in Windows). The invocation step is implemented by building a suitable native stack frame, and then branching to the start address of the native code.

The two steps have very little to do each other, and would tend to be modified under different circumstances. The `NativeSource` implementation would need to be changed to suit different operating systems, while the `nv_invoke` implementation would need to be changed to suit different calling conventions. The same operating system can execute

with different calling conventions (e.g. Linux running on x86 versus Alpha processors), and the same calling convention can be used by different operating systems.

Therefore, in contrast to `Frame` and `FrameSource`, the implementations of `Native` and `NativeSource` have different maintenance characteristics. They are implemented in two separate modules to preserve independence and atomicity.

4.5 Pervasive error handling

In order for a system to be robust and reliable, it must employ a good error handling scheme in a disciplined manner throughout the entire system. If designed properly, the error handling scheme may also contribute directly to flexibility by decoupling the module that detects an error from that which determines the appropriate response. Jupiter employs an error handling strategy that makes robustness relatively easy to achieve, inspired by Java's exception mechanism.

The typical C idiom for reporting error conditions is to use the return value of the function: a return value of zero indicates success, while various nonzero values indicate failure. This idiom is nearly the worst one possible, for a number of reasons. First, it is awkward to use, since every error-prone function call must have its own if statement that checks the return value, causing common-case code to be interleaved with error-handling code. Second, it is easily (even accidentally) circumvented, since C allows function return values to be silently ignored. Third, it necessitates the use of global variables to communicate extra error information, since the function is only capable of returning an integer error code.

In contrast, the Java error handling scheme is well-suited to writing robust, flexible software, and possesses a number of desirable features. Our goal is to find a C idiom that provides the same features, which are as follows:

- *Ease of use.* Jupiter's error handling technique must have a minimum of syntactical overhead, and leave the common-case code readable even when every possible error is properly detected and handled.

- *Precision.* The code for a function is greatly simplified if each statement can

assume that all the preceding statements have succeeded. Hence, it is desirable for the error handling technique not to allow any statement to be executed unless all preceding statements have succeeded.

- *Stack unwinding.* When an error occurs in a deeply-nested function call, all functions in progress on the call stack must be given an opportunity to do the necessary cleanup in order to restore the system to a stable state.

- *Diagnosis.* Being a JVM, Jupiter must implement the proper semantics for the running Java program. Java requires that failures occurring in the JVM must be reported to the Java program by throwing an exception object which is an instance of a specified class. The error handling technique must be capable of causing the correct exception to be thrown in all cases. In addition, Java exception objects can provide a textual error description, and so Jupiter's error handling technique should take advantage of this in order to provide as much diagnostic information as possible.

Given the advantages that exceptions provide to Java code, it is natural to try to duplicate the exception mechanism itself in C. Nonlocal control transfers in C are provided by means of the `longjmp` facility, which can pass control from one stack frame to another. However, this mechanism provides no opportunity for intermediate frames to recover from the error, making it unsuitable as a general error handling scheme by itself. While it may be possible to make use of `longjmp` as part of a suitable error handling framework, it has the additional drawback that the semantics of code containing `longjmp` calls is sometimes surprising. For example, the values of most variables (specifically, non-volatile automatic variables) are undefined after a `longjmp` [KR88].

For these reasons, we have chosen instead to pursue an error handling scheme composed only of well-defined constructs of the C language. The scheme employed by Jupiter consists of the following three conventions:

1. Each error-prone function takes an `Error` object as an extra argument, through which any error that occurs can be described.

2. Functions ensure a *standard postcondition*; namely, that they return zero (or null) if and only if an error has occurred. Besides this, no error information is contained in the return value.

3. Sequences of error-prone statements are combined into a conjunction and used as the condition in an if statement. The "then" clause of the if statement will only be executed if all the error-prone statements succeed, and the "else" clause will be executed when one of them fails.

These conventions combine to form an error handling system with all four desired properties:

- Ease of use is provided by allowing multiple statements to be combined within one if statement, which handles the cleanup for all of them.

- Precision is provided by the short-circuit semantics of C's conjunction expressions: the first statement which returns zero, indicating an error, prevents the remainder of the statements in the conjunction from being executed, and causes control to jump immediately to the cleanup code inside the "else" clause.

- Stack unwinding is achieved by placing all cleanup code inside the "else" clauses. Because no nonlocal control transfer takes place, all stack frames will have their "else" clauses executed. As a further benefit, simplicity and terseness are preserved, since code that does not require cleanup can simply omit the "else" clause.

- Diagnosis is provided by storing a description of the error in the `Error` object.

We believe that this approach is one that is generally applicable to C programs, and not just JVMs. Like most error handling schemes, it makes the system flexible by separating error detection from recovery. However, this scheme differs from others by providing a number of reliability benefits while remaining easy to use, thus encouraging its use throughout the system; indeed, it is almost as easy to write robust code with this scheme as to write careless code that ignores errors. The resulting pervasiveness of robust error handling throughout Jupiter makes its modules more reliable, and thus easier to reuse in novel system configurations.

4.6 Conclusion

In this chapter, we have presented a number of design techniques which have provided substantial flexibility benefits to the Jupiter system. The techniques allow programmers to determine the modules affected by a given modification, and help to make the required effort predictable and proportionate to the perceived complexity of the modification. These properties allow Jupiter to be modified in a straightforward way to suit the needs of future researchers.

Chapter 5

Design for Performance

Jupiter's primary design focus is flexibility, rather than performance, and the two goals are often perceived to be in conflict. However, Jupiter's design embodies a different viewpoint: that a useful platform for JVM research must be flexible enough to allow for efficient implementation. In this chapter, we explore how Jupiter's design allows these two goals to coincide, rather than compete.

The chapter is divided into two parts. First, we focus on interfaces, discussing how they have been designed to permit efficient implementations. Afterward, we focus on implementation, presenting a number of implementation techniques which further highlight the manner in which Jupiter's flexibility can be used to achieve good performance. In each case, just as with flexibility in previous chapter, the desire to achieve performance through flexibility has affected Jupiter's design innumerable small ways. Rather than attempt to be exhaustive, we provide a sampling of the more novel and significant techniques that have been employed in this pursuit.

5.1 Interface Design Techniques

Often, the most natural way to define an interface actually embodies subtle, implicit assumptions which make it difficult to implement efficiently. To permit efficient implementation, it is important to reason about the minimal computation implied by each interface. Choosing the interface with the least implied computation provides the

implementor the flexibility to construct an efficient implementation.

In all, we present three techniques which help to achieve this goal:

- *Design by Contract.* By reducing the need to check for error conditions, implementations can be streamlined for the common case.
- *Lazy computation.* Performing computations only when needed prevents unnecessary computation whose results are never used.
- *Reducing implied arithmetic.* In an efficient system, arithmetic can be costly, and avoiding unnecessary arithmetic can provide a substantial performance benefit.

We describe each technique, and provide examples of how it has been applied to Jupiter.

5.1.1 Design by Contract

Design by Contract, described in Section 4.2 allows a function to impose *preconditions*: constraints on the conditions under which the function may be called. Functions that are permitted to impose constraints on their callers are relieved from the burden of checking for erroneous usage.

In contrast, in the absence of such constraints, the function is forced to detect and respond to erroneous usage; in a well-functioning program, many such errors will never occur, and so checking for them is pure overhead. For instance, practically every function in Jupiter has at least one parameter which is a pointer, and most of them require those pointers to be non-null. If they were not permitted to make such demands, the system would be bogged down by huge quantities of null-pointer checks, the vast majority of which are not necessary.

However, if functions are permitted to prohibit callers from passing null pointers, then they can assume *a priori* that those pointers will be valid, placing the onus for pointer validity on the caller. Then, any caller which can guarantee the validity of a pointer need not check for it to be null. In such cases, when certain errors are guaranteed never to happen, no error checking is ever performed. Omitting the error checking code can improve performance.

An important example of the efficiency that can be achieved using Design by Contract is the interface for accessing elements of a Java array. In the absence of Design by Contract, a function must be prepared to deal with any unusual conditions under which it may be called, and report errors to the caller. As a result, the function for accessing array elements might look like the following:

```
bool ar_getElement(Array ar, int index, Value *result, Error er);
```

This function would return a flag indicating whether an error had occurred. The inefficiency of an interface like this arises from the implicit requirement to check for such conditions as null pointers and out-of-bounds index values in order to preserve the robustness of the system. This may appear to be benign, since the Java specification prescribes that all array accesses require null pointer checks and index checks anyway.

However, when the JVM itself manipulates arrays, it is quite common to know statically that the pointers will never be null, and/or that the index will always be in bounds. An example of this can be found in the native method `arraycopy` from `java.lang.System`, which is intended to provide an efficient way for a Java program to make a copy of an array. The implementation of this method can begin by checking that the arrays are not null and that the indices being copied are in bounds. Afterwards, the contents of the array can be copied element-by-element with no further checking.

With the `ar_getElement` interface described above, we have two rather unpleasant options for implementing `arraycopy`:

- Use `ar_getElement` to copy each element. This places the (now superfluous) null pointer and index checks inside the inner loop, leading to unnecessary performance degradation.

- Implement `arraycopy` inside the `Array` module, making use of "inside knowledge" of how arrays are laid out in order to perform the copy, thus complicating the implementation of the `Array` module.

Design by Contract provides a third alternative. The array element accessor function can impose a precondition stating that the array reference must not be null, and that the index must be in bounds. This places the responsibility on the caller to make

sure that these conditions never occur. Then, because they never occur, the accessor function does not need to check for them, and does not need to report error conditions. The simplified accessor function looks like this:

```
Value ar_element(Array ar, int index);
```

With the interface defined in this way, the `arraycopy` method can call this function inside its inner loop without the unnecessary overhead imposed by checking for error conditions that will never occur. This allows `arraycopy` to be written in a way that is independent of the `Array` module's implementation without compromising performance.

This example demonstrates that a function which fails to impose appropriate preconditions in its caller incurs the overhead required to check for erroneous usage. By proper application of Design by Contract, unnecessary error checking can be eliminated.

5.1.2 Lazy computation

Lazy computation defers the execution of a function until the results of that function are actually needed. This stands in contrast to eager computation, which executes code as soon as it is logically correct and convenient to do so. In this section, we examine the benefits of lazy computation by describing its use in the `FrameSource` interface.

Most computation in an imperative language like C is done eagerly, since lazy computation typically requires an explicitly-coded mechanism, necessitating an inconvenient amount of additional effort and complexity. However, when cases arise in which lazy computation is reasonably straightforward to implement, its benefits are worth considering.

For instance, consider the `FrameSource` interface, used to allocate stack frames for executing Java methods. This interface provides a function called `frs_getFrame` which is declared as follows:

```
Frame frs_getFrame(
  FrameSource frs,
  MethodBody mb,
  Frame caller,
  Error er
);
```

This function returns a `Frame` suitable for executing the given `MethodBody`. In addition to frame creation, `frs_getFrame` serves another important purpose: it communicates to the `FrameSource` which of the previously-allocated frames may be recycled, and which must be retained. The "`caller`" argument is considered to be the last frame that needs to be preserved; any frames which had been allocated after `caller` may be discarded by the `FrameSource`. When using this interface, frames are not explicitly discarded when no longer needed; rather, they are discarded implicitly when the next allocation occurs. This scheme is referred to as *lazy frame discarding*.

The potential disadvantage of lazy discarding is that frames may remain allocated after the associated method has finished executing, thus unnecessarily occupying memory. However, since method invocation is common in Java, we expect that the return of one method will usually be followed shortly by the invocation of another, causing a new frame to be allocated. When this occurs, all old frames will be discarded. Therefore, old frames are not long-lived, even with lazy discarding, meaning there is very little memory wasted by lazy discarding.

To understand the advantages of lazy discarding, consider the implications of eager discarding, using another function such as `frs_discardFrame`. First, `frs_discardFrame` would need to be called for every discarded frame, while the lazy scheme can discard several frames with a single call. Second, the lazy scheme relieves `FrameSource` of the obligation to keep track of the current top frame, a duty which then falls unambiguously to the `Context` interface. Third, all else being equal, an interface that requires explicit cleanup is harder to use than one that provides automatic cleanup. One or more of these issues could be addressed by other means, but lazy discarding is a simple way to achieve all three simultaneously[1].

On top of the efficiency advantages provided by this interface's lazy approach, it has the unusual property that, far from complicating the interface or implementation of `FrameSource`, it actually *simplifies* them. Obviating the `fr_discardFrame` function

[1]It is conceivable that a future researcher may find it desirable to discard frames more promptly. In that case, the `FrameSource` interface could be augmented with a conceptually redundant `frs_discardFrame` function. However, `frs_getFrame` should still be left with the ability to discard frames lazily, for the reasons mentioned above.

means one less function to implement, and one less resource to clean up manually.

This example illustrates that, if the implementation of a lazy computation strategy doesn't unduly complicate the code, laziness can be an effective means of preventing unnecessary computation, thus improving the efficiency of the system.

5.1.3 Reducing implied arithmetic

Some interfaces prescribe a certain amount of unavoidable computation. All else being equal, such interfaces should be avoided, and interfaces should be chosen that perform no more computation than required by the caller.

To illustrate, consider the function provided by the `Frame` interface to access an element of the operand stack:

```
Value fr_operand(Frame fr, int offset, Type tp);
```

Since most of the Java opcodes access data near the top of the stack, it may be natural to choose a convention whereby `offset` represents the distance of the desired item from the top of the stack. The top item itself would be represented by an offset of zero, and deeper items would correspond to larger offsets.

The trouble with this interface becomes apparent when we note that the top of the stack is not fixed; rather, it moves every time data is added to or removed from the stack. Hence, locating a particular item on the operand stack will require the interpreter to perform a calculation resembling this:

```
location = stackBottom + stackDepth - offset
```

Values for `stackBottom` and `offset` may be known to the interpreter at compile time, but `stackDepth` is not, so a certain amount of run-time arithmetic is unavoidable.

In contrast, consider a different indexing convention which counts elements starting from the bottom of stack. With such a convention, locating a particular item would require a calculation like this:

```
location = stackBottom + offset
```

When the interpreter uses the bottom-based scheme to access an element from the *top* of the stack, the bottom-based `offset` cannot be known statically, so some run-time arithmetic is still required. In such cases, the bottom-based indexing scheme provides no benefit. However, the advantage of this scheme becomes apparent in cases when the interpreter already knows the bottom-based offset. All the quantities required by the bottom-based scheme—the `stackBottom` and the `offset`—are both known statically, and so no computation is required at runtime. In contrast, the top-based scheme would still require run-time arithmetic to be performed.

Furthermore, the bottom-based scheme can be beneficial even when a bottom-based offset is not known statically. For instance, since the stack depth changes very frequently, it may be desirable for the interpreter to cache it in a register. Calls to `fr_operand` in a top-based scheme look like the following:

```
fr_operand(fr, offsetFromTop, tp);
```

This interface provides no way to take advantage of the cached stack-depth value. However, with bottom-based offsets, an element of the operand stack could be accessed like this:

```
fr_operand(fr, stackDepth-offsetFromTop, tp);
```

With `stackDepth` in a register, and `offset` most often being a compile-time constant, accessing the operand stack becomes very efficient indeed, performing much as it would if hand-coded in assembly.

Clearly, interfaces which require less computation will be more efficient. Determining what computation is implied by a given interface, as opposed to a particular implementation of that interface, can be subtle, as illustrated by the operand stack example above. However, by careful consideration of exactly which values can be known at compile time, and which require run-time arithmetic, an interface can be designed which minimizes arithmetic across any practical implementations.

5.2 Implementation Techniques

Having considered techniques that make interfaces flexible enough to permit efficient implementations, the task remains to produce the implementation itself, achieving the

desired performance for the critical modules. Performance tuning generally requires careful consideration of each module; however, some techniques apply broadly across all the modules in the system. In this section, we present two techniques which have greatly contributed to achieving the performance of the interfaces which we have so carefully designed:

- *Promoting function inlining.* The use of careful coding techniques, combined with Jupiter's `IncludeGen` tool (described in Section 5.2.1.2), allows the compiler to inline many functions automatically, avoiding the cost of a function call.

- *Exploiting immutability.* Immutable data can be freely shared or replicated without concern for consistency among the various copies, allowing a wide selection of low-overhead implementation options.

5.2.1 Promoting Function Inlining

With function calls as pervasive as they are because of Jupiter's fine-grained modularity, is important to make them as efficient as possible. The option always exists to replace a function declaration with a macro to achieve good performance; however, if acceptable performance can be achieved without macros, the resulting code is cleaner and easier to understand. To achieve the performance of macros while using function calls, it is necessary to make heavy use of function inlining.

Inlining can reduce the overhead of function calls, first by eliminating the overhead introduced by the branch instructions, and then by enabling a host of optimizations to occur across what used to be a function call boundary. As a general rule, function inlining is hard for compilers to do, and it is not desirable to depend entirely on cutting-edge compilers to achieve good performance. Hence, we have developed techniques that give an average compiler the help it needs to recognize and exploit as many inlining opportunities as possible.

This section presents the two implementation techniques that allow substantial function inlining to occur, even when a compiler (`gcc`) is used that does not have cutting-edge inlining capabilities:

- Minimizing common-case code size allows the most frequently called functions to be inlined.
- The use of a preprocessing tool called `IncludeGen` enables function inlining across module boundaries.

The availability of pervasive function inlining relieves the programmer from concerns of performance penalties for cross-module function calls, allowing module boundaries to be based on uncompromised software engineering principles.

5.2.1.1 Minimizing common-case code size

When a function is inlined, its body is replicated in each call site. Thus, inlining large functions can dramatically increase the size of the executable, leading to a number of problems including increased paging and increased instruction cache pressure. To address these problems, compilers typically have a size threshold, beyond which a function will not be inlined. However, there are cases within Jupiter where this kind of inlining threshold might cause the compiler to miss important optimization opportunities, resulting in significant performance degradation. To address this issue, Jupiter functions are written in a manner which produces small functions to handle common cases, thus making sure that they can be inlined when it matters most.

An example of this technique can be found in Jupiter's `ConstantPool` implementation. In a JVM, a constant pool acts as a cache of information used by the methods of its class, and may be accessed very frequently. The body of the function for accessing a `METHODREF` constant pool entry might look something like this:

```
MethodDecl ctp_method(ConstantPool ctp, int index){
  if(ctp->cache[index] == EMPTY){
    /* Lengthy, complex code to resolve a METHODREF */
  }
  return = ctp->cache[index];
}
```

In the common case, when a constant pool entry has already been resolved, the desired value is simply read from the cache. This operation is fast, making the overhead of a function call significant. Hence, we would like these accessor functions to be inlined.

The difficulty occurs when the resolving code inside the if statement is so long that it prevents the surrounding function from being inlined. The function could be marked explicitly as `inline`, but that would cause the exact code-bloating problems that the compiler was trying to avoid by *not* inlining the function in the first place.

To get the benefits of inlining without the code bloat, we split this code into two separate functions as follows:

```
static void ctp_fillMethod(ConstantPool ctp, int index){
  /* Lengthy, complex code to resolve a METHODREF */
}

MethodDecl ctp_method(ConstantPool ctp, int index){
  if(ctp->cache[index] == EMPTY){
    ctp_fillMethod(ctp, index);
  }
  return = ctp->cache[index];
}
```

The `ctp_method` function now has a small, fixed-size body, and can be inlined. Meanwhile, the lengthy, complex resolution code is in a separate function, for which the compiler can make an independent inlining decision. This technique avoids the function call overhead in the common case, while simultaneously preventing undesirable code expansion.

5.2.1.2 IncludeGen

Unless function inlining is performed by the linker[2], C's separate compilation model prevents inlining across compilation-unit boundaries. With the typical coding style having each module in a separate compilation unit, inlining across module boundaries does not occur, resulting in high system overhead due only to modularity. To address this issue, we have developed a tool called "`IncludeGen`" which preprocesses C code to place all modules in a single compilation unit, thus enabling cross-module function inlining.

`IncludeGen` facilitates inlining by generating a `.c` file containing one `#include` directive for each module in the system, which puts all the code into a single compilation

[2] Or by equivalent capabilities in the compiler.

unit. This allows the compiler to "see" all the function definitions in the entire system during its optimization phase, and enables implicit inlining of cross-module function calls, plus all the additional optimizations that are possible after inlining occurs.

One minor drawback to this technique is that all identifiers in the system must be globally unique, because they will all be visible in the same compilation unit. This is not a big problem for Jupiter, because all public names (and most private ones) make use of module prefixes, as described in Section 4.1, automatically making them globally unique. The handful of name clashes that occur between private names from separate modules are easily fixed by renaming the offending identifiers, usually just by adding the appropriate module prefix.

A more significant drawback is the difficulty of maintaining the proper ordering of the `#include` statements. In our testing, we used the `gcc` compiler, and discovered that it requires a function definition to appear before its first use if it is to be inlined. Thus, the ordering of the `#include` directives is crucial to performance.

Unfortunately, as important as they are, the ordering constraints identified by the programmer do not appear explicitly in the file; instead, they are implicit in the final ordering of the `#include` directives. This makes maintenance problematic. When new modules are added to the system, the programmer may discover that certain important functions are not being inlined, and may choose to remedy the problem by rearranging the `#include` directives. In doing so, the desirable properties of the existing ordering may be lost, and the system may perform very poorly as a result. Hence, despite the attractiveness of the `#include` file for allowing inter-module inlining and optimizations, maintaining such a file would be prohibitively problematic.

Therefore, Jupiter does not leave the maintenance of the `#include` file up to the programmer. Instead, a utility called `IncludeGen` is provided, which automatically generates the file based on a list of module names passed in on the command line[3]. To get the ordering right, each source file is scanned for specially-formatted comments which specify ordering constraints on that module. The resulting dependence graph is first checked for circularities, which are reported as errors, alerting the programmer to con-

[3]The module names usually come from the `Makefile`, which has a complete list of all modules in the system.

flicts in the ordering requirements. (This stands in contrast to the manually-maintained `#include` file, which would simply produce an executable with poor performance.) Finally, the graph is traversed in topological order to generate the `#include` directives.

The `IncludeGen` utility recognizes two directives for specifying the dependency edges for a module: the `Uses` directive indicates that the containing module should appear after another; and the `UsedBy` directive indicates that the containing module should appear before another. Though the system would be conceptually complete with only one of these two directives, both are provided so that lower-level modules never need to refer to higher-level ones. For instance, `Context.c` is a low-level standard module which is independent of the implementation of any other module (as described in Section 4.3.1). However, `Context.c` makes use of "callback" functions implemented in `Frame.c`, a high-level module, and these functions should be inlined. Placing a "`Uses Frame.c`" directive in `Context.c` would introduce a reverse dependency by making a low-level module refer to a higher-level one. Instead, we place a "`UsedBy Context.c`" directive in `Frame.c`, preserving the module hierarchy.

In addition to `Uses` and `UsedBy`, `IncludeGen` provides two more directives called `UsesAll` and `UsedByAll`. `UsesAll` is for top-level modules such as the main interpreter loop. It saves the programmer from having to name all the modules in the system explicitly with a long list of `Uses` directives: a scheme that would lead to maintenance problems when modules are added, removed, or renamed. `UsedByAll` is for fundamental modules like `MemorySource.c` and `Error.c` which are used heavily throughout the system. It saves the programmer from having to add `Uses` directives to almost every module.

Modules that employ `UsedByAll` and `UsesAll` directives still belong to the dependence graph, meaning that they can still have ordering constraints relative to one another. For instance, one module in the `UsesAll` subset can have a `Uses` directive to make sure it appears after another module which is also in the `UsesAll` subset.

The `IncludeGen` utility allows ordering dependencies to be given explicitly and be checked for inconsistencies, greatly simplifying the task of producing a good module ordering. This degree of control over the module ordering allows Jupiter to achieve

good performance with a standard, commonly-available compiler (namely `gcc`).

5.2.2 Exploiting Immutability

Data that does not change after it has been constructed is referred to as being *immutable*. Besides the tremendous ergonomic benefit afforded by the ability to reason about immutable data, there are a number of additional benefits:

- The data can be freely replicated without concern for consistency among various copies, because the copies will never change. This allows an implementation to make use of processor caches (and even registers) as well as other mechanisms that enhance performance by making extra copies of data.
- Because immutable objects have no state changes through which to reveal their identity, it is sometimes possible to discard the object, and then re-create it later, transparently to the users of that object. This may relieve the implementation of the burden of storing objects when they can easily be recomputed.
- No synchronization is required when accessing the data from multiple threads because no thread ever changes it.

To illustrate the efficiency benefits of immutable data, this section presents three examples of Jupiter interfaces whose immutability easily allows for certain efficient implementations which would be difficult to construct if the system were based on mutable data abstractions. Each example demonstrates more than merely the ability to pass data by value instead of by reference, since each has some additional special feature:

- `Value`. We have seen that immutable data can be passed by value instead of by reference; the `Value` interface demonstrates that the opposite change—passing by reference instead of by value—can also be beneficial.
- `Type`. Immutable data can be passed both by value and by reference on a case-by-case basis, providing the benefits of both approaches simultaneously.

- `MemorySource`. Even when an interface is mutable, the immutable portions can be isolated, and the benefits of immutable data can still be realized.

Together, these examples illustrate the wide range of situations in which immutable data may confer some benefit.

5.2.2.1 Value

Immutable data can be beneficial because it allows data to be passed by value when it would normally be passed by reference. However, the converse is also true: data normally passed by value can also be passed by reference, as demonstrated by the `Value` abstraction.

`Value` represents a single value of any of the Java types. Since the behaviour of the Java opcodes is independent of the JVM implementation, it is desirable for both interpreters and JIT compilers to share the code that specifies the opcode behaviour. However, JIT compilers are fundamentally different from interpreters in that they do not have access to the data values, since those values will not exist until the generated code is executed.

To address this issue, Jupiter treats `Values` differently depending on whether they are used by an interpreter or a compiler. Interpreters simply treat a `Value` as a C union of all the possible data types, while JIT compilers must treat `Values` as placeholders that indicate where the data *will be located* when the generated code is executed. The placeholder may be a virtual register number, or a memory location, or anything else that the JIT compiler uses to represent a data location.

As the implementor of the JIT compiler, it may be desirable to describe a data location using a data structure. If the data structure is immutable, then `Value` may be defined as a pointer to such a structure. Users of the `Value` interface may do whatever they normally would have done with a `Value`, and the fact that it has been implemented as a pointer to immutable data, rather than a simple value, will go undetected.

Thus, an immutable structure can be passed by reference, and callers expecting data to be passed by value will continue to operate as expected. In this way, immutable data provides yet another degree of freedom for the system implementor.

5.2.2.2 Type

Jupiter's `Type` abstraction models the information contained in Java's type descriptor strings. A given descriptor string never changes, and so the `Type` abstraction is immutable, and hence amenable to being passed by value. In particular, the `Type` example demonstrates how a single data type can even be passed by value in some cases and by reference in others. This results in a highly efficient implementation, allowing the use of the `Type` abstraction to expand beyond the cases where Java uses type descriptor strings, to become the ubiquitous way of manipulating type information throughout the system.

To illustrate the efficiency permitted by the immutability of `Type`s, consider what happens in a fairly typical opcode, `ICONST_0`, which pushes an `int` constant with the value of zero onto the operand stack. The implementation of this opcode in Jupiter's `opcodeSpec` module looks like the following:

```
fr_pushValue(curFrame, 0, tp_new(tpc_int));
```

The call to `tp_new` constructs a `Type` object which is used to indicate the type of the pushed data to the `fr_pushValue` function. In this case, an `int` is being pushed, so a `Type` representing the `int` type is passed[4]. Creating a new `Type` object every time this code is executed could pose a significant efficiency problem, since time spent allocating the `Type` object could be much greater than that spent actually pushing the value on the stack. Even caching previously-created `Type` objects may prove prohibitively costly, since the actual computation being performed here—the pushing of a value onto the operand stack—is very small. It may appear that using a `Type` object for this purpose would not be practical; however, the immutability of the `Type` abstraction allows it to be implemented in a highly efficient manner that eliminates the object-creation overhead.

To begin to address this problem, we divide type descriptors into two categories: the *complex* types, and the *simple* types. Complex types have structure; for instance, an array type is not complete without specifying the type of the array's elements. One cannot simply say that the type of an object is "an array" without providing the element type as well. Similarly, a method type is not complete without the types of its

[4]Specifically, it represents a Java type descriptor of `"I"`.

arguments and return value, and a class type is not complete without specifying the name of the class.

In contrast, simple types can be completely described as an element of an enumeration, without any additional data. There is a finite list of simple types that includes such types as `int`, `long`, `void`, and so on. This enumeration is known as the `TypeChar` enumeration, because each type is described by a certain character: `'I'` for `int`, `'J'` for `long`, `'V'` for `void`, etc. These are the same character values assigned by the Java specification.

Having made the distinction between simple and complex types, the solution to the efficiency problem is to pass simple types by value, while passing complex types by reference. Simple types are more transient and more prevalent, so the performance benefit is significant.

To accomplish this, `Type`'s constructor (called `tp_new`) looks like this:

```
Type tp_new(MemorySource ms, String descriptor, Error er){
  TypeChar tpc = descriptor[0];
  if(tpc_isSimple(tpc)){
    result = (Type)typeChar;
  }else{
    result = tp_newComplex(ms, descriptor, er);
  }
}
```

It begins by extracting the `TypeChar` from the type descriptor string. When that `TypeChar` indicates a complex type, an internal `tp_newComplex` function is called to construct the appropriate data structure. However, for the case of a simple type, the type character itself is returned. Type characters can be distinguished from pointers since the ASCII codes are always between 1 and 255, while pointer values (in most operating systems) are never in this range: the memory page starting at virtual address zero is marked invalid by the memory manager to detect null pointer dereferences.

A number of "core" accessor functions for the `Type` interface are implemented like this:

```
TypeChar tp_char(Type tp){
  if((unsigned)tp > 0xFF)
    return tp->typeChar; /* tp is a struct */
  else
```

```
    return (TypeChar)tp; /* tp is the TypeChar itself */
}
```

The remainder of the accessor functions call the core functions, and don't themselves need to distinguish pointers from `TypeChars`.

Finally, returning to the `ICONST_0` example, we can see how this implementation of the `Type` interface makes for highly efficient code. As shown above, the implementation of `ICONST_0` looks like this:

```
fr_pushValue(curFrame, 0, tp_new(tpc_int));
```

Once `tp_new` is inlined, this code becomes the following:

```
fr_pushValue(curFrame, 0, (Type)tpc_int);
```

Inside `fr_pushValue` are a number of cases for handling specific data types. In fact, the implementation of `fr_pushValue` can be thought of as a `switch` statement on the data type. Since the data type has been resolved to a known compile-time constant, the `switch` statement is optimized away, leaving only the specific case appropriate for pushing an `int` value on the stack. The resulting code is exactly the same as if the type-specific code were called directly, while at the source-code level, the clean object-oriented interface remains intact.

This example demonstrates how immutability can permit a given data type to be passed by value in some cases and by reference in others. This occurs with no impact on the source code, which may continue to be written under the assumption that all data is passed by reference.

5.2.2.3 MemorySource

Section 2.3.4 describes a scheme in which `MemorySources` are passed by value rather than by reference, permitting the code to be optimized substantially. An interesting thing to note regarding this example is that, on first glance, the `MemorySource` interface does not appear to be immutable. In particular, `ms_getMemory` is not referentially transparent: every time `ms_getMemory` is invoked, it may return a different value, thus revealing that some sort of state change must be occurring inside the `MemorySource`.

The reason immutable-data techniques can be applied to `MemorySource` is that its functionality can be divided into a mutable and an immutable part. The actual memory management process requires mutable data; however, the particular memory pool associated with a given `MemorySource` never changes. Keeping track of which memory pool is associated with which `MemorySource` involves only immutable data, and is therefore amenable to immutable-data techniques. This explains why, when applying these techniques to `MemorySource` in Section 2.3.4, it was the node number—effectively the memory pool number—that was passed by value.

Thus, even when dealing with mutable interfaces, it is often possible to isolate specific immutable data that can be passed by value to improve performance.

5.3 Conclusion

In this chapter, we presented a sampling of the techniques which have helped Jupiter's interfaces and implementations to be as efficient as possible. By using these techniques, we have achieved both performance and flexibility simultaneously, making Jupiter a good platform for future JVM research.

Chapter 6

Experimental Evaluation

In this chapter, we discuss experiments performed to evaluate the success of Jupiter's design. We have employed a number of techniques to ensure both flexibility and good performance, as was described in Chapters 4 and 5. Regrettably, it is difficult to quantify the effect of each individual technique on the flexibility and performance of the system. Because these are design techniques, changing them would require redesigning large parts of the system using inferior techniques, and the nature and degree of the inferiority that should be introduced is debatable. Even though some of the techniques are amenable to a direct comparison—for instance, the same interface could be designed with and without lazy computation—the benefit of these techniques arises from the accumulation of small improvements throughout the system, and undoing these techniques would involve modifying large quantities of code. Furthermore, even ignoring these issues, it is not clear that re-implementing parts of the system with an obviously inferior design approach, and then benchmarking the result, would provide any useful insight.

Therefore, rather than try to quantify the impact of these techniques individually, we attempt to convey the nature of the improvement they produce by demonstrating the flexibility and performance of the resulting system. Specifically, we first carry out 3 modifications to the Jupiter JVM, and demonstrate the degree of flexibility Jupiter provides. Afterward, we evaluate the performance of Jupiter using standard benchmarks, and show that it provides adequate performance.

6.1 Flexibility

Jupiter's primary goal as a research platform is flexibility, and although we believe we have achieved this goal, a system's flexibility is hard to quantify: a wide variety of software metrics have been developed in an attempt to do so, but the success of such metrics is questionable [Pow98]. Instead, we characterize the impact of a number of modifications according to three criteria:

- *Demarcation*: the clarity with which Jupiter's interfaces identify the modules that will need to be changed. Good demarcation is important before the modification is performed, to facilitate the estimation of the work required, and to enhance confidence in the correctness and completeness of the proposed changes.

- *Localization*: the degree to which Jupiter's interfaces confine the required changes to a small quantity of code. Good localization is important during the modification process, to reduce the amount of code that must be understood and altered by the programmer.

- *Insulation*: the degree to which Jupiter's interfaces conceal the effects of one modification from others, allowing them to be combined trivially. Good insulation is important after the modification is complete, to allow modifications contributed by multiple programmers to be combined, thus multiplying their individual efforts.

In all, we present three modifications:

- *Stack reversal.* The original implementation of the Java stack grew downward; that is, newer frames occupied lower memory addresses than older ones. During the development of Jupiter, the stack implementation was altered so that the stack grew upward, toward higher addresses. Such a fundamental change would be challenging in a JVM that did not encapsulate the stack allocation details from the rest of the system.

- *Scalar array allocation.* Large arrays of non-pointer data can lead to performance problems unless the garbage collector is told that it does not need to scan such

arrays for pointers. Hence, it is useful to make use of a `MemorySource` specially designed for allocating scalar (i.e. non-pointer) data, which communicates this fact to the garbage collector. Adding this new `MemorySource` to Jupiter proved to be quite straightforward.

- *Quick opcodes.* Certain Java opcodes are hard to implement efficiently for a variety of reasons. A well-known solution to this problem is to rewrite the bytecodes upon the first execution of a given opcode, to replace it with another opcode that has more efficient semantics. Jupiter's structure makes such bytecode replacement straightforward.

This section motivates and describes each modification, and then evaluates the demarcation, localization, and insulation of that modification achieved by Jupiter's module structure. The modifications are not the easiest possible ones that fit nicely into Jupiter's module structure. Instead, we have strived to choose modifications that might be difficult in other JVMs, but were relatively simple in Jupiter. Some of the modifications presented are not necessarily well demarcated, localized, and/or isolated by Jupiter's design, but maintenance effort is a product of all three of these factors, and extraordinary success in one can make up for mediocrity in others. For instance, a modification that is localized to half a dozen lines of code will be reasonably easy to perform and to combine with others even if, strictly speaking, it is not well demarcated or isolated.

6.1.1 Stack reversal

The Java call stack can be implemented in a number of different ways, embodying different trade-offs, entirely hidden behind the `Frame` and `FrameSource` interfaces. One such trade-off is the direction in which the stack grows as new stack frames are added. The new frames may occupy lower memory addresses than previous frames, in which case the stack is said to "grow downward;" or else new frames may occupy higher memory addresses, in which case it is said to "grow upward."

Either growth direction may be desirable. Downward-growing stacks are the norm for processes running in many modern operating systems, including x86 Linux and

Windows. Having the Java call stack stack grow downward may allow it to exploit operating system facilities provided especially for downward-growing regions of memory, such as Linux's `MAP_GROWSDOWN` flag for the `mmap` system call [JT98].

On the other hand, there are reasons to prefer an upward-growing stack. The semantics of Java make it much simpler and more efficient to have each frame's operand stack grow in the same direction as the call stack as a whole, and to lay out the local variable array in the same direction as the operand stack grows. Therefore, a downward-growing call stack is heavily biased toward a downward-growing operand stack and a downward-ordered local variable array[1]. With all these things oriented downward, toward lower memory addresses, all operand stack and local variable indexes would need to be negated on every access, harming the efficiency of these operations.

During the development of Jupiter, we switched from the downward-growing scheme to the upward-growing one. The call stack implementation is clearly demarcated by the `Frame` and `FrameSource` interfaces, whose implementation is localized to a single `Frame.c` module (as described in Section 4.4.1). This module consists of 534 lines of code, of which roughly half had to be changed. Aside from this, no other code in the system needed to be altered, meaning this modification is isolated from any other that does not impact the stack layout. Therefore, Jupiter's module structure succeeded in facilitating the stack reversal modification.

In contrast, a JVM that has not been carefully designed for modularity could easily make this modification problematic. For example, in Kaffe [Wil02], the main interpreter function is called `runVirtualMachine`, and is declared as follows:

```
void runVirtualMachine(
  methods *meth,
  slots *lcl,
  slots *sp,
  uintp npc,
  slots *retval,
  volatile vmException *mjbuf,
  Hjava_lang_Thread *tid
);
```

Note that the local variables ("`lcl`") and stack pointer ("`sp`") are passed as a pointer to an array of `slots`. Being non-opaque array pointers, there is no way to know what

[1]This is due to the way frames can be made to overlap. See Section 3.5 for more details.

parts of Kaffe depend on the stack layout. For instance, a cursory search for the word "slots" revealed the following code from the soft_multianewarray function, which allocates multidimensional arrays:

```
/* stack grows up, so move to the first dimension */
args -= dims-1;
```

In addition, the code that calls runVirtualMachine appears as follows:

```
/* Allocate stack space and locals. */
lcl = alloca(sizeof(slots) * (meth->localsz + meth->stacksz));
(...)
sp = &lcl[meth->localsz - 1];
runVirtualMachine(meth, lcl, sp, npc, retval, &mjbuf, tid);
```

In this code, the stack pointer is initialized to point past the end of the local variables, before the method execution begins[2]. Thus, information regarding stack layout has not been hidden behind any interface, allowing undesirable dependencies to form easily. Furthermore, soft_multianewarray shows that such dependencies are, in fact, found in a number of places. Hence, the stack reversal modification is not well delineated within Kaffe.

Jupiter does, of course, contain equivalent code; however, in keeping with Jupiter's coding conventions, this code is located inside Frame.c, along with *all* the other code in the system which has information regarding stack layout. This serves to demarcate, localize, and isolate stack layout modifications to a small amount of code.

6.1.2 Scalar Array Allocation

The MemorySource interface is one that lends itself to a wide variety of implementations for various purposes. Jupiter currently contains no fewer than six different MemorySources (shown in Section 3.2), and does not yet implement the locality-enhancing schemes described in Section 2.3.3.

With the MemorySource interface as polymorphic as it is, we found it desirable early on to write a generic implementation that uses a jump table to dispatch the

[2]Note that it also allocates Java stack frames on the C stack, which not only precludes the overlapping-frame optimization discussed in Section 3.5, but also ties the two kinds of stacks together, precluding certain forms of thread migration as described in Section 2.2.4.

`ms_getMemory` call to the appropriate implementation function, much like C++'s virtual method tables. Afterward, additional `MemorySource`s could be added without changing *any* existing code, except for the one location where the `MemorySource` is instantiated. This represents the ultimate demarcation and localization, by confining the modifications to a single line of existing code in Jupiter's `main` function. It also completely isolates the new `MemorySource`s from any other system modification, since practically the entire modification occurs in a newly created file—the new `MemorySource`'s own implementation file—upon which there can be no prior dependencies because the file didn't exist.

This ideal applies only to cases when one `MemorySource` is to be substituted for another, for all memory allocation purposes. For instance, it allows us to use the `ErrorMemorySource` for regression testing, or the `Tracer` for memory profiling, by changing a single line of code. However, in some cases, it is desirable to substitute another `MemorySource` for some subset of memory allocation. In those cases, the change is not quite so ideal, though we believe it still to be very close to minimal.

To illustrate, consider the allocation of large arrays of non-pointer data, and the problems this causes for conservative garbage collectors (described in detail in Section 2.3.3). A new `MemorySource` is needed to communicate to the garbage collector the scalar nature of the memory being allocated: this is the `BoehmAtomicMemorySource`, described in Section 3.2.

Clearly, it cannot simply be substituted via the ideal one-line change, because that would cause it to be used for all memory allocation throughout the system. This would cause pointer-containing memory blocks to be marked as pointer-free, leading to unpredictable results. Instead, the scope of the modification must be expanded to include those modules that are responsible for allocating scalar arrays in the first place.

At the time of the modification, Jupiter had two such modules: `ObjectSource`, responsible for allocating Java arrays, which (of course) are sometimes arrays of scalars; and `SemanticFactory`, used during classfile parsing to allocate arrays for storing method bytecodes, interface lists and exception lists. Each of these modules was augmented to use two `MemorySources` instead of one: the first is used for blocks which will

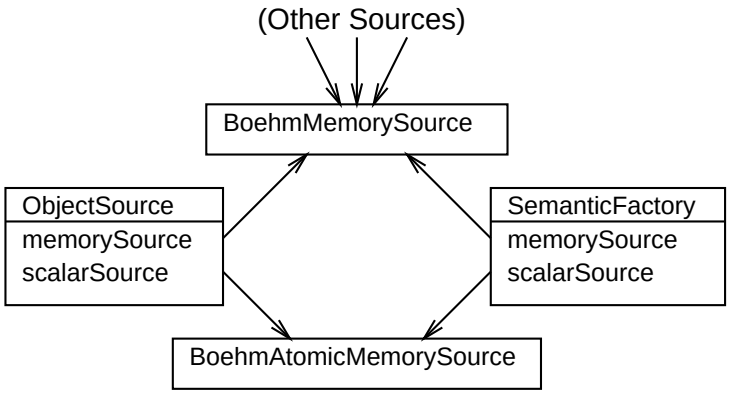

Figure 6.1: Using a separate MemorySource to allocate scalar data.

contain mixed pointer and scalar data, while the other is only used to allocate blocks that will contain no pointers. This scheme is depicted in Figure 6.1.

This kind of modification is only needed when it is discovered that large scalar arrays are causing stress on the garbage collector. In such cases, it is simple to use the Boehm collector's memory profiling capabilities to find the source of the problematic arrays. With this knowledge, Jupiter's interfaces clearly demarcate the code required to modify the allocation policy at a known call site. The allocation of Java arrays is clearly hidden behind the `ObjectSource` interface, while the construction of metadata for parsed classfiles is hidden behind `SemanticFactory`.

The modification is localized to the `ObjectSource` and `SemanticFactory` implementation modules, plus the modules that instantiate them. The quantity of code involved is a dozen lines for `ObjectSource` and five lines for `SemanticFactory`, all of which are trivial modifications that simply add or substitute a `scalarMemorySource` pointer for the usual `memorySource`. This, plus the constructor calls, brings the total number of affected lines to around twenty.

Despite the small line count, it may appear that this modification has grown to encompass an unexpectedly large number of modules: not only the new `BoehmAtomicMemorySource` module, but `ObjectSource` and `SemanticFactory`, and the modules that construct them. However, all this work is merely refactoring that does not make these latter modules dependent on the `BoehmAtomicMemorySource` in any way. On the contrary, they are only aware that they are using two `MemorySource`

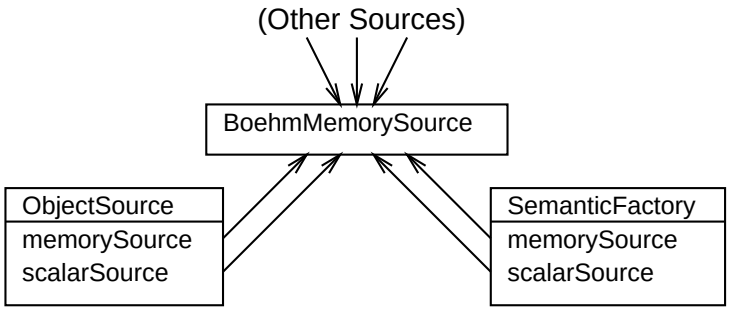

Figure 6.2: A trivial change reverts to allocating scalar data from the original MemorySource.

references instead of one. The entire modification could be reversed simply by pointing both references to the original `MemorySource`, as shown in Figure 6.2; a modification which would affect only a single line of code in Jupiter's `main` function. Therefore, after the necessary refactoring, the ideal localization has been achieved for this modification.

This modification is isolated from others that do not change the `ObjectSource`'s or `SemanticFactory`'s policy for selecting a `MemorySource`. It is not surprising that two changes to a `MemorySource` selection policy would not be isolated from each other. However, the extremely small number of lines involved, and the simple nature of the alterations required, mean that the effort required to combine `MemorySource` selection policies is very small indeed.

Therefore, Jupiter's module structure succeeded in facilitating the special treatment of scalar array allocation.

6.1.3 Quick Opcodes

Some of Java's opcodes are hard to execute quickly by their very nature. One remedy for this is to alter a method's bytecodes to replace slow instructions with more efficient ones defined internally by the JVM. In particular, we have used this technique to improve the performance of the `getfield` and `putfield` opcodes, which are inherently slow for two reasons:

- *Overloading.* Some instructions can operate under a number of different conditions, forcing the implementation to check the current conditions every time that

opcode is encountered, even if the conditions for a given instruction never change. In particular, `getfield` and `putfield` can be called before or after the field is resolved. The implementation must check whether the field has been resolved every time, despite the fact that the field accessed by a given instruction only needs to be resolved once. Also, the same opcodes are used for both category-1 and category-2 (4-byte and 8-byte) fields, so the implementation must check the field type in order to determine the number of bytes to read or write. These checks could be eliminated by rewriting the instruction, after resolving the field, with a new one that simply assumes the field has been resolved, and is specialized for the appropriate field size.

- *Data retrieval.* Some instructions access data that is stored in structures outside of the bytecode stream, and traversing these structures reduces performance. In particular, `getfield` and `putfield` need to find the field's offset within the object, while the bytecode stream provides only the index of the field reference within the constant pool. Thus, the constant pool must be accessed, and the field descriptor must be consulted to find the field's offset. This overhead can be eliminated by rewriting the instruction so that the field offset is stored directly in the bytecode stream, resulting in faster access.

In Jupiter, new "quick" opcodes called `qgetfield` and `qputfield` were added which assume the field has been resolved, and which store the field offset as a parameter directly within the bytecode stream, thus addressing both of the above issues. The implementations of `getfield` and `putfield` were changed so as to replace themselves with their respective quick versions the first time they execute. Thereafter, subsequent executions of the same bytecode will find the quick instructions, and will execute without the overhead.

The first step in implementing the quick opcodes was to specify their semantics in the same manner as in the `opcodeSpec` module (described in Section 3.4), though the new opcodes were placed in a separate module to keep the standard Java opcodes separate from those defined by Jupiter. Next, opcode numbers were chosen from among those left unassigned by the Java spec. This was done by adding the new opcodes to

the `bytecode.h` file that defines all the opcode number assignments.

To minimize overhead, the quick instructions store the field offset directly in the bytecode stream. Acquiring the offset of a given field, and accessing object contents by offset, are both operations that are not supported by the `base` interfaces. Hence, the `common` interface headers for `Field` and `Object` were augmented with functions that provide the needed functionality, and these headers were "`#included`" into the interpreter[3].

Finally, the bytecode substitution mechanism itself was implemented. To replace the implementations of the original opcodes, the associated entry in the branch target array (described in Section 3.4) was changed to point to the new bytecode-substituting code.

In all, the modification was confined to the `Interpreter` and `bytecode` modules, plus the other modules directly involved in the implementation of the new opcode itself (namely `Field` and `Object`). A total of 8 lines of code were added—no existing lines were changed—plus approximately another 80 to implement the new opcodes and the bytecode substitution mechanism. Hence, this modification was quite well demarcated and localized by Jupiter's module structure.

As for isolation, that depends largely on the requirements of the new opcode being implemented, though all new opcodes require a modification to the `bytecode.h` header. However, this change is exceedingly simple: the name of the new opcode must be added to a list. Therefore, though the isolation in this case is strictly not ideal, the effort required to combine multiple new opcodes is trivial, and so they are effectively isolated.

6.2 Performance

To test Jupiter's functionality and performance, we have used it to run the single-threaded applications from SPECjvm98 benchmark suite [SPE02]. In this chapter, we present the execution times consumed by these benchmarks running on Jupiter, and

[3]Note how even the additional dependencies resulting from breaches of information hiding are demarcated by Jupiter's module structure: it is clear from inspecting the `#include` directives in the interpreter that it is making use of non-`base` interfaces.

compare them with results from Kaffe 1.0.6, and from Sun's JDK v1.2.2-L. We find that Jupiter is faster than Kaffe and slower than JDK. We believe that Jupiter's performance is sufficient to allow JVM research to proceed unimpeded. Afterward, we investigate the reasons that Jupiter's performance is slower than that of JDK.

Since multithreading is outside the scope of this work[4], we have used the subset of the SPECjvm98 benchmarks which are single-threaded, which includes the following programs:

- `201_compress` is a modified Lempel-Ziv compression algorithm. It manipulates large arrays of bytes.
- `202_jess` is an expert system that applies rules to a database of facts in order to solve puzzles. It stresses the memory manager by allocating many short-lived objects.
- `209_db` performs a number of queries on a memory-resident database.
- `213_javac` is Sun's Java compiler. It does a lot of object creation and method invocation.
- `222_mpegaudio` decompresses mp3-encoded data. It performs floating-point multiplication and addition on large arrays.
- `228_jack` is a parser generator. It stresses the JVM's exception handling mechanism by throwing large quantities of `ArrayIndexOutOfBoundsException`s.

Table 6.1 compares the execution times of each benchmark run on the various JVMs. All times were measured on a 533MHz Pentium III with 512MB 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—as described in Section 5.2.1.2—to facilitate function inlining. The times were reported by the Unix "`time`" program, and therefore include all JVM initialization. All benchmarks were run

[4]Jupiter has, of course, been designed to support multithreading, and a working implementation of the concurrency mechanism has been added by others. Thus, we believe the benchmark results to be very close to Jupiter's actual performance once multithreading is fully debugged and operational.

	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 6.1: Execution time, in seconds, of each benchmark 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.

with verification and JIT compilation disabled, since Jupiter did not yet possess either of these features at the time the tests were run[5]. The results are sorted in order of Jupiter's performance relative to the JDK's.

Averaged across all benchmarks, Jupiter was 2.20 times slower than JDK, and 2.65 times faster than Kaffe[6]. Jupiter's flexibility has allowed us to build a JVM from the ground up with an interpreter at this level of performance in well under one man-year of work, presumably orders of magnitude less than has been done on JDK and Kaffe. This performance is a result of our ability to experiment with a number of different optimizations quickly, with little effort.

With this foundation in place, a JIT compiler can now be added to Jupiter that can achieve the level of performance expected of a JIT compiler. The interpreter is fast enough that we can expect, once the JIT compiler is added, that the time spent in the interpreter will not affect overall performance. We believe this to be a reasonable expectation, since experience has shown that runtime systems with both an interpreter

[5]The effect of verification on performance was less than our experimental error, affecting execution time by a fraction of one percent. JIT compilation, of course, has a substantial impact.

[6]We have chosen to present geometric means for the following reason. Kaffe/JDK ratios can be computed as the product of Kaffe/Jupiter times Jupiter/JDK, and could be placed in a column of their own. Because the geometric mean of the products equals the product of the geometric means, such a table would be internally consistent. This would not be the case for other kinds of averages.

and a compiler spend a large majority of the execution time running the generated code; for instance, the Dynamo runtime system [BDB00] typically spends only 1.5% of its time in the interpreter, with the remainder spent executing generated code. If Jupiter exhibits a similar execution time ratio to Dynamo, then our interpreter's 120% slowdown relative to JDK translates to a mere 1.8% performance penalty for the JVM overall. Hence, even at its present level, the performance of our current interpreter implementation is fast enough not to be a factor in Jupiter's ultimate performance.

Nonetheless, it is interesting to investigate the reasons for the performance difference between Jupiter and JDK. We can consider three possible causes for this difference:

- *Implementation.* Jupiter's interpreter may be slower simply because it uses inferior implementation techniques. To the degree that this is the problem, continuing optimization work can bring Jupiter's performance closer to that of JDK.

- *Compiler.* Jupiter's code frequently appears to be beyond the capability of `gcc` to optimize well. If Sun has used a superior compiler to build JDK, Jupiter might benefit immediately from using a similarly powerful compiler.

- *Flexibility.* The design and implementations decisions that have given Jupiter its flexibility may directly impede performance.

In the remainder of this section, we investigate the compiler and implementation issues, and attempt to gauge the magnitude of their performance effect. The third issue, flexibility, is the most relevant, considering Jupiter's focus on flexibility; unfortunately, is also the most difficult to quantify. First, we would need to construct a non-flexible version of Jupiter for comparison; and since Jupiter's flexibility is ubiquitous, this would essentially mean writing an entirely new JVM. Second, even if it were feasible to construct a non-flexible version, decisions on exactly what design aspects are primarily due to flexibility are far from clear. Thus, such an experiment is not only infeasible, but also ill-defined, and there is no reason to believe the resulting conclusions would have any merit.

Therefore, the following subsections concentrate on the implementation and compiler issues; flexibility can be inferred to be the cause of any slowdown not attributed to these

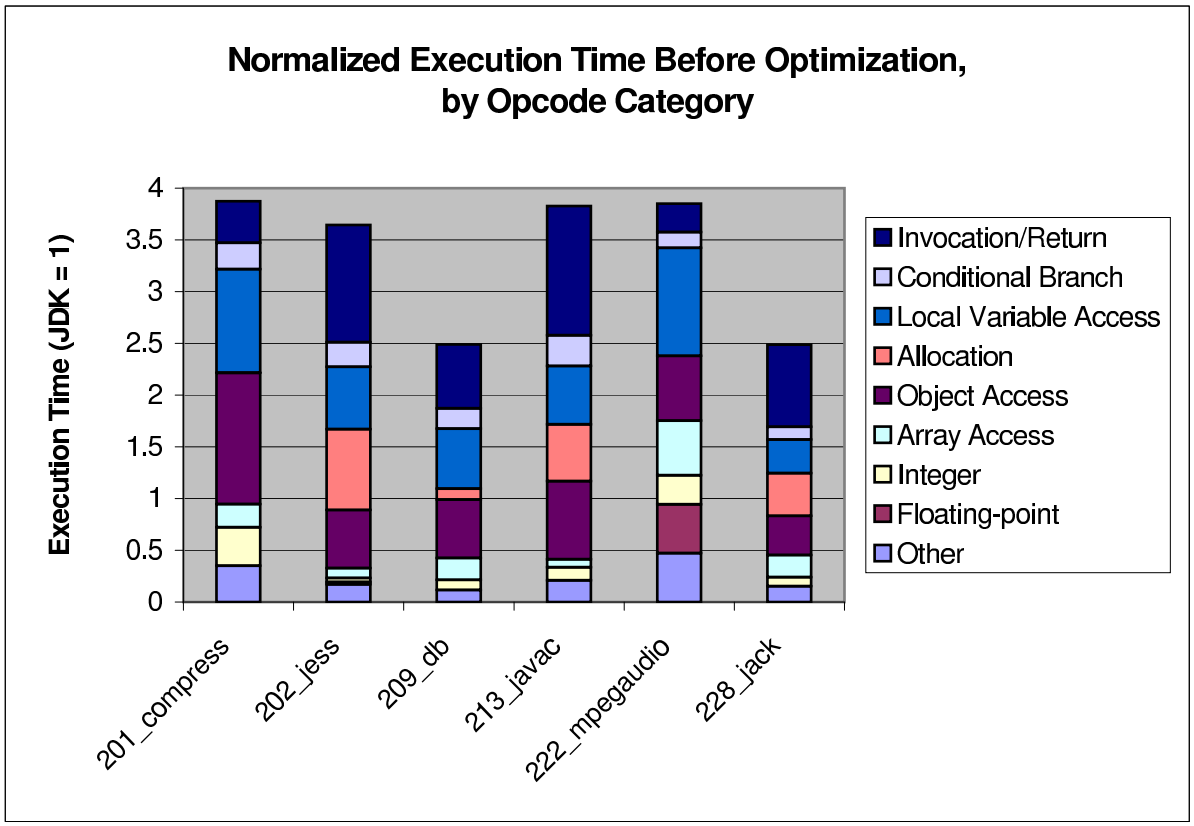

Figure 6.3: Execution profile before applying optimizations.

other two causes.

6.2.1 The Implementation

To help locate performance bottlenecks in Jupiter's implementation, we have included a sample-based profiling facility that measures the amount of time consumed executing each kind of opcode. It is implemented using a Unix signal delivered to the interpreter at regular intervals (50 times per second) with a signal handler that records which opcode was being executed at that moment. A sample result is shown in Figure 6.3, which groups the opcodes into the following categories:

- *Invocation/Return* includes all opcodes that change the call stack: the four `invoke` opcodes, the six `return` opcodes, and Jupiter's optimized `qinvokevirtual`.
- *Conditional Branch* includes the 16 `if` opcodes, plus `lookupswitch` and `tableswitch`.

- *Local Variable Access* includes the 50 variations of `load` and `store` opcodes.
- *Allocation* includes opcodes that allocate new objects: `new`, `newarray`, `anewarray`, and `multianewarray`.
- *Object Access* includes `getfield` and `putfield`, as well as Jupiter's optimized `qgetfield` and `qputfield`.
- *Array Access* includes the 16 array loads and stores.
- *Integer* includes the 13 32-bit integer mathematical and bitwise operators, such as `imul`, `ixor`, `iinc` and `ineg`.
- *Floating-point* includes the 16 operators for manipulating and comparing `float` and `double` datatypes, such as `fmul` and `dcmpl`.
- *Other* includes the other 74 opcodes[7].

The times are normalized, with JDK's execution time on the same benchmark as 1.0. This graph clearly shows the opportunities for optimization; for example, the large bars representing object access indicate that these opcodes are likely to benefit from optimization effort.

In response to this profile data and consequent investigation of the code, both at the C and assembly level, we have implemented a number of optimizations. The impact of these optimizations is shown in Figure 6.4, which charts the execution-time reduction due to each optimization:

- *bottombased*: The `Frame` interface was changed to use bottom-based operand stack indexing, as described in Section 5.1.3.
- *threaded*: The "loop-and-switch" interpreter was replaced by a threaded interpreter, as described in Section 3.4.

[7]The `wide` prefix opcode is counted separately, in the "other" category, as its own opcode (though the point is moot, since it did not appear in any of the benchmarks).

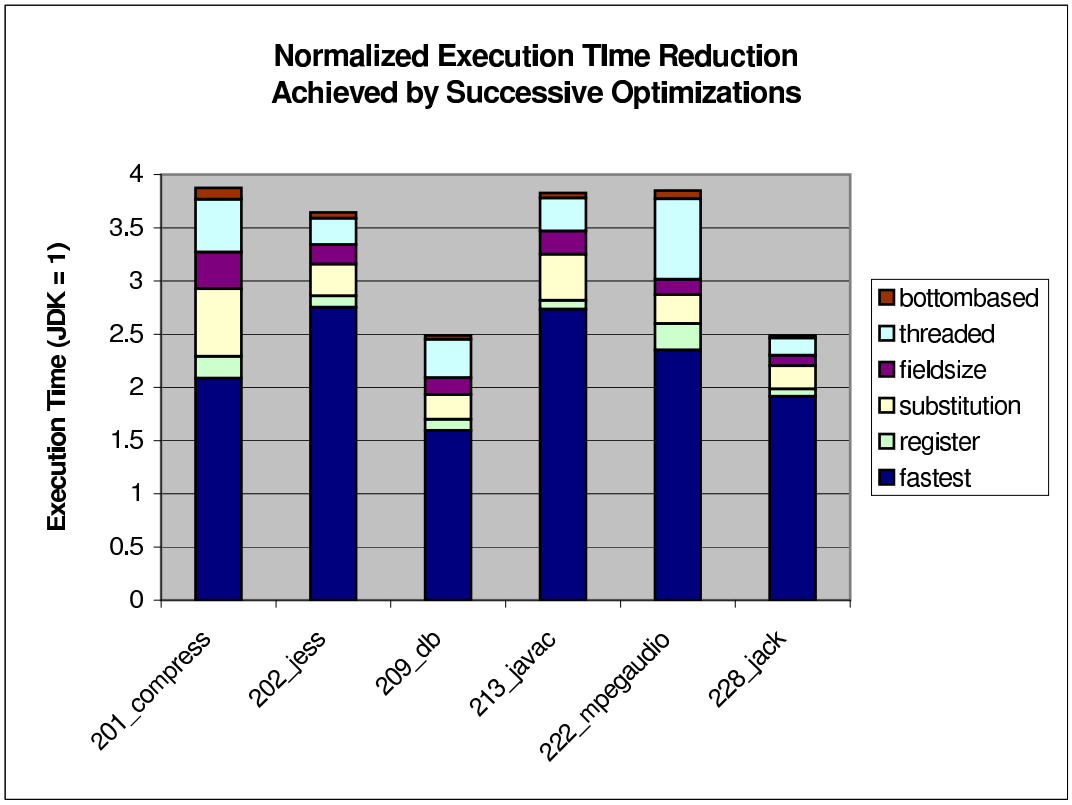

Figure 6.4: Effect of each optimization on benchmark execution times.

- *fieldsize*: The field's size (either 4 or 8 bytes) was cached inside the `Field` pointer, relieving the interpreter from having to traverse Jupiter object references to find this information.

- *substitution*: Bytecode substitution was implemented, as described in Section 6.1.3, using the Jupiter-defined `qgetfield`, `qputfield`, and `qinvokevirtual` opcodes.

- *register*: A CPU register was assigned to storing the location of the currently-executing instruction.

The resulting execution profile after performing all optimizations is shown in Figure 6.5, which illustrates the effect of these optimizations by contrast with Figure 6.3. Some of the optimizations were more beneficial than others, but all were simple to implement, and were largely independent, so evaluating their respective merits was quite straightforward, further demonstrating the flexibility Jupiter offers.

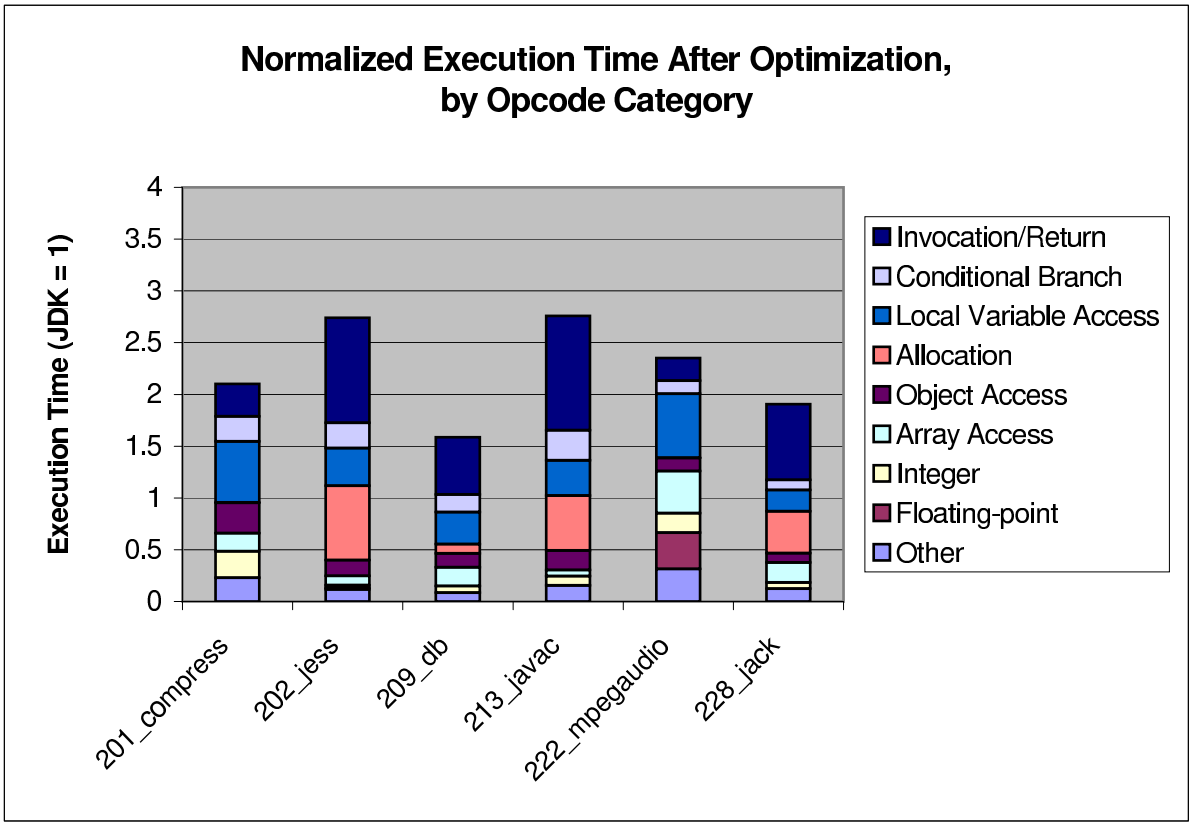

Figure 6.5: Execution profile after applying optimizations.

It is clear from the final profile chart that optimization 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. Using this information, we could continue the optimization process, and proceed to investigate these opcodes for optimization opportunities. However, because the interpreter's performance is sufficiently fast that we expect it not to affect Jupiter's ultimate performance once a JIT compiler is added, the continuation of the optimization process is left as future work.

6.2.2 The Compiler

Jupiter's coding style relies heavily on function inlining to achieve good performance, so any weakness in the compiler's inlining ability can have a substantial impact. For example, `gcc` running at the `-O3` optimization level translated the implementation of the `qgetfield` opcode into the assembly code shown in Figure 6.6. There are a number

of simple ways in which this code could have been better optimized:

- Lines 16 and 17 load the operand depth into the `%ecx` register, even though `%ebx` already contains the operand depth from line 2.
- Line 15 computes a value which is always zero, because of line 3. Line 18 then subtracts this zero from the `%ecx` register, which has no effect. These three lines could all be removed.
- Line 20 reads, modifies, and writes a value that is then immediately read again in line 21, requiring 3 memory accesses. Instead, this value could be saved from line 12. Modifying it inside the register, and then writing it to memory, requires only one memory access instead of three.

These improvements require nothing more than common subexpression elimination, an optimization that is not beyond the capability of modern optimizers [Muc97]. The exact reason that `gcc` did not successfully perform these optimizations is hard to determine, due to the heuristic nature of compiler optimization techniques. However, it appears that the implementation of `qgetfield` is too stressful on the inlining facility of `gcc`, which causes subsequent optimization steps to produce code that is clearly sub-optimal.

Making the improvements manually in the assembly code would eliminate 5 instructions from the common-case path, or 26% of the original total, producing the code shown in Figure 6.7. Our measurements seem to indicate somewhere between 12% and 15% improvement to the performance of `qgetfield`. However, the actual performance impact is difficult to measure, since object access accounts for only 11% of the `201_compress` benchmark (and even less of the others). Hence, even a 26% improvement would amount to less than 3% overall decrease in execution time, which is not much more than experimental error.

The `qgetfield` instruction is not exceptional: most of the opcode implementations contain the same kind of sub-optimal code. In fact, `qgetfield` could be expected to be easier to optimize than other instructions, since it was specifically designed for this purpose (as described in Section 6.1.3). Thus, we expect that this kind of optimization

```
      /* qgetfield */

      // Get operand depth
1     movl -96(%ebp),%esi
2     movl (%esi),%ebx

      // This %edi value is never used
3     movl %ebx,%edi

      // Get the "this" pointer
4     movl (%esi,%ebx,4),%eax
5     movl %eax,%esi

      // Report error if null
6     testl %eax,%eax
7     jne .L87489
      movl 20(%ebp),%edi
      movl $.LC318,(%edi)
      movl $0,4(%edi)
      jmp .L23250

    .L87489:
      // Get the field offset from the bytecode stream
8     movl -108(%ebp),%eax
9     movzbl 1(%eax),%ecx

      // Get the field value
10    movl 8(%ecx,%esi),%eax

      // Make %edi=0 (see line 3)
11    subl %ebx,%edi

      // Get the operand depth (which is already in %ebx)
12    movl -96(%ebp),%ebx
13    movl (%ebx),%ecx

      // No-op, because %edi is zero
14    subl %edi,%ecx

      // Store the field value on the operand stack
15    movl %eax,(%ebx,%ecx,4)

      // Jump to execute the next instruction
16    addl $3,-108(%ebp)
17    movl -108(%ebp),%esi
18    movzbl (%esi),%eax
19    jmp *g_labels.2986(,%eax,4)
```

Figure 6.6: Poorly-optimized assembly code originally generated by gcc to implement the qgetfield opcode. Instructions executed in the common case are numbered. The comments have been added manually afterward.

```
      /* Improved qgetfield */
      /*
          edi = current frame
          ebx = operand depth
          esi = "this" pointer
          ecx = field offset
          edx = code pointer
      */

      // Get operand depth
1     movl -96(%ebp),%edi
2     movl (%edi),%ebx

      // Get the "this" pointer
3     movl (%edi,%ebx,4),%eax
4     movl %eax,%esi

      // Report error if null
5     testl %eax,%eax
6     jne .L82022
      movl 20(%ebp),%edi
      movl $.LC318,(%edi)
      movl $0,4(%edi)
      jmp .L21167

  .L82022:
      // Get the field value
7     movl -108(%ebp),%edx
8     movzbl 1(%edx),%ecx
9     movl 8(%ecx,%esi),%eax

      // Push it on the operand stack
10    movl %eax,(%edi,%ebx,4)

      // Jump to execute the next instruction
11    addl $3,%edx
12    movl %edx,-108(%ebp)
13    movzbl (%edx),%eax
14    jmp *g_labels.3187(,%eax,4)
```

Figure 6.7: Hand-written code to implement the qgetfield opcode. Instructions executed in the common case are numbered.

would apply widely across Jupiter's code. Although the impact of each optimization is small, the individual improvements accumulate, so the performance enhancement offered by another compiler might be substantial. However, we have left this as future work, since, as mentioned above, the interpreter's present performance is sufficient that it will not be a factor once a JIT compiler is added.

6.2.3 Conclusion

Jupiter's flexibility has allowed us to implement, with relatively little effort, an interpreter with performance comparable to existing commercial and open-source JVMs. The present level of performance is sufficient to render the interpreter's contribution to overall execution time insignificant, once a JIT compiler is added.

Opportunities can still be found for further optimization, by profiling and tuning the code, and by using more advanced optimizing compilers to build the Jupiter executable. However, because Jupiter's performance is already sufficient, this optimization work has reached the point of diminishing returns, and additional optimizations work on the interpreter is left as future work.

Chapter 7

Conclusions

In this thesis, we have presented the design of a modular, flexible framework intended to facilitate research into JVM scalability. We have described the building-block architecture employed, as well as the design and implementation of the key modules. We have presented design techniques that allowed us to achieve the desired flexibility and performance, which we have shown to be successful in creating a useful platform for JVM research.

Experimentation with our framework demonstrates that Jupiter's flexibility has facilitated a number of sophisticated modifications, some of which are difficult to accomplish using Kaffe. Jupiter's structure served to demarcate, localize, and isolate the code that needed to be changes, allowing the modifications to be implemented and combined with relatively little effort.

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. This level of performance is sufficient that the interpreter's performance will have a negligible impact on the performance of the JVM as a whole, once JIT compilation is implemented.

By providing a flexible JVM framework, we hope to encourage future research into JVM scalability. A highly scalable JVM would be a tremendous asset for high-performance computing, combining the portability benefits of Java with the performance benefit of scalable multiprocessor computing.

7.1 Future Work

The primary focus for future work will be on JVM scalability issues. Jupiter has been carefully designed so it can be made scalable with relatively little effort, but the work of achieving scalability is ongoing [Cav02]. We anticipate that the object-oriented nature of Jupiter's design, combined with its building-block architecture, will provide the same scalability benefits found in other scalable systems that use the same techniques [Gam99, Kri94, K4201].

In addition, we anticipate the need for additional design work on Jupiter, since the current design does not facilitate certain modifications. There are two important modifications in particular that are not facilitated: precise garbage collection, and JIT compilation.

Jupiter currently has no facility to describe the location of pointers in memory, thus necessitating the use of a conservative collector. Adding a precise collector will require a new memory allocation interface that keeps track of pointers. We believe that such an interface, written in the same style and spirit as the rest of Jupiter's interfaces, could be added without disturbing most of the system.

The second modification that Jupiter does not facilitate is switching from an interpreted execution engine to a JIT compiler. An interesting approach to this modification is to redesign all the `base` interfaces so that, rather than perform the desired action, they explicitly generate code in some intermediate representation (or IR). The interpreter would then interpret the IR instead of the bytecode, while the JIT compiler would, as usual, translate the IR into native code. This would make the interpreter and the JIT compiler trivially interchangeable, while providing the additional performance benefit that optimizations developed for the JIT compiler would also improve the performance of the interpreter.

One final design consideration for future work is in the handling of metadata. Currently, Jupiter's object model spreads the responsibility for storing metadata throughout the system, requiring each object to be rather heavyweight. If metadata were stored separately, the objects could be very lightweight, which would enable substantial optimizations to be performed, in a fashion similar to the `MemorySource` optimizations

described in Section 2.3.4.

We expect that these avenues of design research will be the first steps toward an infrastructure that provides flexibility for the design and implementation of scalable, high-performance Java virtual machines.

Bibliography

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[ABH+01] 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.

[AFT99] 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*, pages 21–24, 1999.

[AWC+01] J. Andersson, S. Weber, E. Ceccet, C. Jensen, and V. Cahill. 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.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[Boe02] H. Boehm. A garbage collector for C and C++, http://www.hpl.hp.com/personal/Hans_Boehm/gc, 2002.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[Cav02] C. Cavanna. Scalability of java virtual machines. Master's thesis, University of Toronto, in progress, 2002.

[CHS72] H. Curry, J. R. Hindley, and J. Seldin. *Combinatory Logic, Volume II.* North-Holland, Amsterdam, 1972.

[CNI02] The Cygnus Native Interface for C++/Java integration, http://gcc.gnu.org/java/papers/cni/t1.html, 2002.

[CP02] GNU Classpath, http://www.gnu.org/software/classpath/classpath.html, 2002.

[DA01] Patrick Doyle and Tarek Abdelrahman. Jupiter: A modular and extensible JVM. In *Proceedings of the Third Annual Workshop on Java for*

High-Performance Computing, ACM International Conference on Supercomputing, pages 37–48. ACM, June 2001.

[DAK00] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable jvms. *IBM Systems Journal*, 39(1), 2000.

[Doy02] P. Doyle. Codegen: A code generator for abstract syntax trees, http://www.eecg.toronto.edu/ doylep/codegen, 2002.

[FSUZ88] G. Feil, M. Stumm, R. Unrau, and S. Zhou. Hurricane operating system — a preliminary design. Technical report, University of Toronto, September 1988.

[Gam99] B. Gamsa. *Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System.* PhD thesis, University of Toronto, 1999.

[GCJ02] The GNU compiler for the Java programming language, http://gcc.gnu.org/java, 2002.

[GH01] 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*, pages 27–39, 2001.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, October 1994.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 2000.

[IKY+99] 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*, pages 119–128, 1999.

[Jal02] Jalapeño project, http://www.research.ibm.com/jalapeno, 2002.

[Jik02] Jikes Research Virtual Machine, http://www-124.ibm.com/developerworks/oss/jikesrvm, 2002.

[JNI02] Java Native Interface, http://java.sun.com/j2se/1.3/docs/guide/jni/index.html, 2002.

[JT98] M. Johnson and E. Troan. *Linux Application Development.* Addison Wesley, 1998.

[Jup02] The jupiter project, http://www.eecg.toronto.edu/jupiter, 2002.

[K4201] The K42 Team. *K42 Overview*, 2001.

[K4202] The K42 Team. The K42 project, http://www.research.ibm.com/K42/index.html, 2002.

[KHBB01] 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.

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[Kri94] O. Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, 1994.

[KS97] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, 1997.

[Lak96] J. Lakos. *Large-Scale C++ Software Design.* Addison-Wesley, 1996.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.

[Mar01] F. Maruyama. OpenJIT 2: The design and implementation of application framework for JIT compilers. In *Proceedings of USENIX JVM'01*, 2001.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[Mey97] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[MOS+98] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT: A reflective Java JIT compiler. In *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, 1998.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[MWLX99] 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*, pages 2781–2787, 1999.

[Ott02] T. Ottinger. Ottinger's rules for variable and class naming, http://www.objectmentor.com/publications/naming.htm, 2002.

[Par72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 14(1):221–227, 1972.

[Pow98] A. Powell. A literature review on the quantification of software change. Technical report, University of York, 1998.

[PZ97] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[SPE02] SPECjvm98, http://www.specbench.org/osg/jvm98, 2002.

[Sun02] Sun Microsystems, http://www.java.sun.com, 2002.

[SW01] F. Siebert and A. Walter. Deterministic execution of Java's primitive bytecode operations. In *Proceedings of the USENIX JVM Research and Technology Symposium*, pages 141–152, 2001.

[Wha02] J. Whaley. joeq virtual machine, http://joeq.sourceforge.net/index.htm, 2002.

[Wil02] T. Wilkinson. Kaffe — a virtual machine to run Java code, http://www.kaffe.org, 2002.

[YC97] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.