The Design and Implementation of a Java Virtual Machine on a Cluster of Workstations

by

Carlos Daniel Cavanna

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
2003

ABSTRACT

The Design and Implementation of a Java Virtual Machine on a Cluster of Workstations

by

Carlos Daniel Cavanna

Master of Applied Science

Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

2003

We present the design, implementation, and evaluation of a Java Virtual Machine (JVM) on a cluster of workstations, which supports shared memory in software. More specifically, we first extend the Jupiter JVM infrastructure to support multithreading on Symmetric Multiprocessors (SMPs). We then extend this multithreaded Jupiter to deal with memory consistency, private memory allocation, and the limitations on the use of some resources on a 16-processor cluster of PCs interconnected by a Myrinet network; which supports Shared Virtual Memory and a pthreads interface. Our experimental evaluation is performed using standard benchmarks. On a 4-processor SMP, it indicates that the performance of the multithreaded version of Jupiter is midway between a naïve and a state-of-the-art implementations of a Java interpreter. Our evaluation on the cluster demonstrates that good speedup is obtained.

For Mariela, my beloved wife; and my dear parents.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, Professor Tarek Abdelrahman, for his constant guidance and encouragement during the development of this work. His high standards, attention to detail and confidence placed this work among the most interesting and challenging experiences I have ever had.

I would also like to thank Patrick Doyle, for the long and productive discussions about Jupiter, his support during the initial stages of this project and for providing such a good system to work with. And Peter Jamieson, for the long hours we spent solving development issues in the SVM system when enabling Jupiter to run on the cluster, and debugging the newly released Linux version of CableS. I must extend my appreciation to Professor Angelos Bilas for providing the resources to make this project advance, and Rosalia Christodoulopoulou and Reza Azimi for continuously improving VMMC.

I am also thankful to all my friends, who were always present when I needed them. In particular to Diego Novillo, who constantly provided accurate suggestions throughout this work, and to Professors Gerardo Castillo and Andrea Lanaro, who I have always considered very good friends, for motivating me so strongly to follow this path.

Last, but not least, I want to express my deepest gratitude to my wife for her understanding, patience and love; and my family, without whose support, endless love, and lifelong encouragement I would not have accomplished this achievement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In recent years Java has steadily gained wide acceptance as a programming language of choice for application development. The main reason for this success is that Java provides a set of tools that give a platform-independent context for developing and deploying these applications. Furthermore, it includes a wide set of classes and libraries that simplify application development. In particular, these features benefit projects that target heterogenous platforms.

However, Java programs often lack performance compared to programs written in languages like C or C++. This is mainly due to the Java execution model, which consists of a software interpreter, called a *Java Virtual Machine* (JVM) that executes a Java program, compiled to an abstract set of instructions called *Java bytecodes*. This model can give rise to poor performance because the functionality required for the execution of individual bytecodes is delivered entirely in software.

Therefore, there has been considerable research into approaches for improving the performance of Java programs. The most significant of these approaches is *just-in-time* (JIT) compilation [AAB$^+$00, IKY$^+$99], which dynamically translates Java bytecodes into native instructions that can be directly executed in the target hardware, thus reducing the performance penalty incurred by interpretation. Other approaches include improved array and complex number support [AGMM99, WMMG99], efficient garbage collection [DKL$^+$00, BAL$^+$01], and efficient support for threads and synchronization [AAB$^+$00].

Parallel processing is another important and orthogonal approach for improving the performance of Java programs. The availability of multiple processors improves the execution time of multithreaded applications. Since their introduction, JVMs have allowed for the execution of multiple threads in a Java program in shared memory on *Symmetric Multiprocessors* (SMPs). Unfortunately, SMPs are limited in their scalability, and can be very expensive at large scales.

A cost-effective alternative to SMPs is *clusters of workstations*, which offer large and potentially scalable environments; clusters can be expanded by easily incorporating additional off-the-shelve hardware and software components. For this reason, clusters provide an interesting platform for the execution of Java applications. However, clusters do not support the traditional shared memory programming model. This creates many challenges in enabling JVMs to run on clusters, including memory consistency and non-shared memory allocation issues. This thesis explores these challenges, and addresses the issues involved in the design and implementation of a JVM on a cluster of workstations.

1.1 Objectives and Contribution

The main goal of this work is to explore the design and implementation of a JVM that delivers good performance on a cluster of workstations. More specifically, the objective is to explore the issues involved in enabling a JVM to execute on a 16-processor cluster with Shared Virtual Memory (SVM). The cluster nodes are interconnected through Myrinet network interfaces [BCF$^+$95, Myr03], which offer low latency and high bandwidth communications, and serve as support media for SVM.

For this purpose, it was necessary to utilize a JVM infrastructure that could be extended to work on the cluster. The Jupiter JVM [Doy02, DA02] is an extensible single-threaded JVM research framework. It was designed with strong emphasis on modularity, flexibility and portability, and thus it possesses the properties required for this work.

Our work focuses on the design and implementation of a multithreaded and cluster-enabled versions of the Jupiter JVM, while maintaining its extensibility and flexibility properties. The first part of this work presents the extension of the original Jupiter JVM to support multithreaded Java applications on SMPs. The second part shows the extension of the multithreaded Jupiter JVM to run on the SVM cluster. These two extensions of Jupiter are evaluated using standard benchmark suites (SPECjvm98

[SPE03] and Java Grande [EPC03, Jav03]). Our experimental results demonstrate that the multithreaded Jupiter speed is midway between Kaffe [Wil02], a naïve implementation of a Java interpreter, and Sun's JDK [SUN03], a highly optimized state-of-the-art interpreter. The performance of the multithreaded Jupiter is also comparable to that of the single-threaded Jupiter [Doy02]. This indicates that our multithreading extensions do not introduce significant overhead. Furthermore, the cluster evaluation indicates that good speedup is achievable for the applications.

The main contribution of this work is the development of a cluster-enabled JVM, that delivers good performance and speedup. In this respect, the extensions to Jupiter deliver superior speedup to existing JVMs that run on clusters. More specifically, this work involves the addition of some needed functionality that would allow Jupiter to run multithreaded programs on an SMP, and ultimately on the cluster. This further involved:

- The creation of the required infrastructure to support all the operations involved with thread handling, which had to correspond to the interfaces of the Java `Thread` class.
- The development of the necessary support for the thread behavior stated in the JVM Specification [LY99].
- The synchronization of those Jupiter structures and objects that become shared in a multithreading configuration.
- The extension of the Java opcode optimizations (quick opcodes) to work safely in the presence of threads.
- Dealing with a lazy release memory consistency model [KCZ92], provided by the SVM system, which adheres to the Java Memory Model.
- The exploitation of non-shared memory allocation, which led to an efficient implementation of a distributed JVM on the current realization of the cluster.
- A careful utilization of the cluster resources that have limitation constraints.

In addition, this work constitutes a step towards providing a single system image (SSI) based on a JVM on a cluster, and provides an infrastructure for further investigation of issues related to the design of JVMs on SMPs and clusters. However, it does not currently address the issues of supporting a single-image I/O infrastructure for Java programs or distributed garbage collection on the cluster.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces the background required for the understanding of this work and presents some related work. Chapter 3 describes the steps needed to extend the Jupiter JVM to support multithreading. Chapter 4 details the design issues and techniques involved in enabling Jupiter to work on the cluster. Chapter 5 shows the results of the experimental evaluation. Chapter 6 draws conclusions. Appendix A gives details on the benchmarks used for the experimental evaluation. The thesis concludes with Appendix B where an alternative implementation of the Java stack is explored and evaluated.

CHAPTER 2

Background and Related Work

This chapter introduces necessary background required for understanding the implementation of the multithreaded and cluster-enabled versions of the Jupiter JVM, and provides a summary of related work that aims at running JVMs on clusters of workstations. Section 2.1 briefly describes the Java model. Section 2.2 gives a short overview of the Jupiter JVM. In Section 2.3, the concept of memory consistency and the consistency models relevant to this work are discussed. Clusters are introduced in Section 2.4 and Shared Virtual Memory clusters, including the platform used for this work, are addressed in Section 2.5. Finally, Section 2.6 provides a summary of the related work in this area.

2.1 The Java Model

A Java Virtual Machine [LY99] is an abstract specification for a computer implemented in software that interprets and executes a program, in a similar way in which a microprocessor executes machine code. This program is referred to as the *Java program*. Java programs are compiled to a standard set of codes, called *Java bytecodes*, which are an abstract, portable, machine-independent representation of the executable code [LY99]. The Java environment includes the JVM, Java compilers (to generate Java bytecodes from Java program source files) and Java system classes. The latter is a set of standard classes that perform common tasks, and act as a uniform and limited API to the underlying operating system.

Portability is one of the most important features of the JVM environment. For that reason, Java has gained much popularity both in industry and academia. Today, there are several JVMs publicly available, including: Sun's JDK [SUN03], Kaffe [Wil02], ORP [ORP02], Kissme [KIS03], SableVM [SAB03], OpenJIT [OPE01, MOS$^+$98, Mar01] and JikesRVM (Jikes Research virtual machine, formerly known as Jalapeño) [AAB$^+$00, JAL01, JIK03]. Some of these JVMs were designed for special purposes or are not flexible or modular enough to be suitable for general research environments.

2.2 The Jupiter JVM

The Jupiter JVM [Doy02, DA02] was conceived as a modular and extensible JVM research framework. It was designed to be a JVM infrastructure that will support experimentation with a wide variety of techniques for improving JVM performance and scalability. Thus, it was developed with a set of carefully designed abstract interfaces. This allows the different Jupiter components to be fully isolated from one another, thus becoming interchangeable. Ultimately, these features made Jupiter portable and easy to modify [Doy02, Jog03].

Jupiter uses the ClassPath [GNU03] project as its core Java runtime library. ClassPath is distributed under a General Public Licence (GPL), and aims at creating a free software replacement for Sun's proprietary Java core class libraries. It is intended to include all native methods and internal classes necessary to support a completely functional JVM. At the time of this writing, ClassPath is still under development.

Jupiter suited the needs of our project well, therefore it was elected as the base research platform. Its inherent modularity properties gave the necessary flexibility for the ultimate goal, enabling the project to run on top of a cluster. Furthermore, this was a good opportunity to fully test its extensibility in a real and challenging project.

2.3 Memory Consistency

A memory consistency model for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e., to become visible to other processors) with respect to one another. This includes operations to the same and different memory locations and by the same or different processes[1] [CSG99].

There are many memory consistency models. An overview of the two models that are relevant to this work is presented in the remainder of this section.

A memory model is *sequentially consistent* if the result of any execution is the same as if operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program [Lam79]. Every process in the system appears to issue and complete memory operations one at a time and atomically in program order [CSG99]. A memory operation does not appear to be issued until the previous one issued by the same process has completed. The shared memory appears to service these requests one at a time in an arbitrarily interleaved manner. Memory operations appear atomic in this interleaved order. Synchronization is still required to preserve atomicity (mutual exclusion) across multiple memory operations from a process or to enforce constraints on the interleaving across processes.

The *lazy release consistency* memory model [KCZ92] relaxes all program orders for memory operations that take place outside synchronization operations by default. It only guarantees that orderings will be made consistent at synchronization operations that can be identified by the system. When synchronization operations are infrequent, this model provides considerable reordering freedom to the hardware and compiler. This model is based on the observation that most parallel programs use synchronization operations to coordinate accesses to data when it is necessary. Between synchronization operations, they do not rely on the order of accesses being preserved. The intuitive semantics of these programs are not violated by any program reordering that happens between synchronization operations or accesses as long as synchronization operations are not reordered with respect to data accesses or one another. This memory model makes a distinction among types of synchronization operations, *acquires* and *releases.* An acquire operation is defined as a read (or read-modify-write) operation performed to gain access to a set of operations or variables; for example, `lock(mutex)`[2]. A release is

[1] In this sense, memory consistency subsumes coherence.

[2] A mutex is a variation of a binary semaphore or lock. The terms mutex and lock are used indistinctively throughout this work.

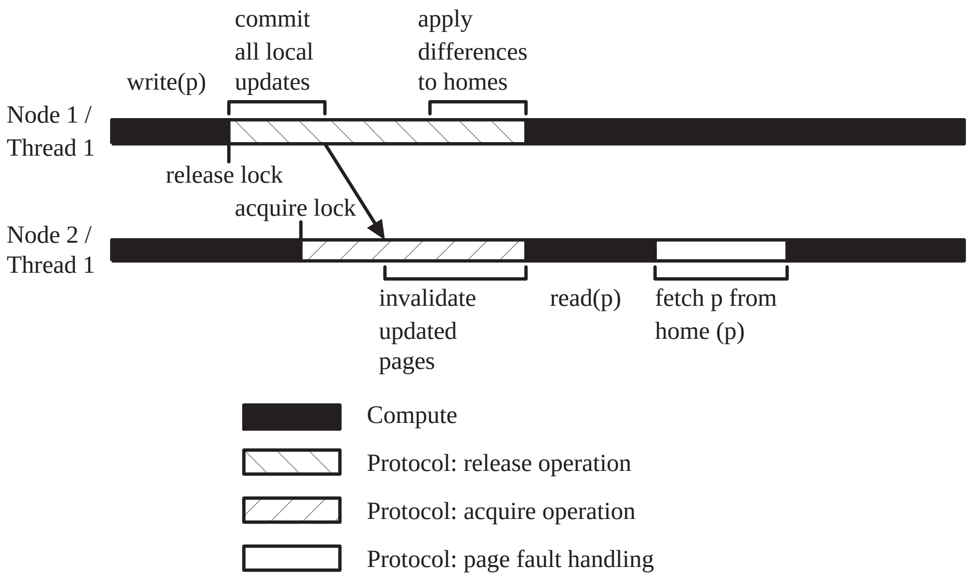


Figure 2.1: Lazy Release Memory Consistency Model

a write (or a read-modify-write) operation that grants permission to another processor to gain access to some operations or variables; for example, `unlock(mutex)`.

In the lazy release model, each page is designated with a processor, or a node, which is called its home node. Processors cycle through acquire-compute-release cycles. As seen in Figure 2.1, when a thread requests a lock release, it ends the current time interval by committing all pages updated by any local thread during the past interval. Then, the thread releases the corresponding lock to the next requesting node, and computes and sends the differences of the updated pages to their home nodes. During an acquire operation, the processor fetches from each remote node the list of updates which are needed for this synchronization step and invalidates the corresponding pages. This is determined by comparison of *timestamp* values. A subsequent access to an invalidated page triggers a page fault that results in remote fetching the latest version of the page from its home node.

In SMP systems, which are usually sequentially consistent, it is the hardware that guarantees the memory properties. In SVM systems (to be discussed in Section 2.5), which for the purpose of this work are assumed to adhere to the lazy release memory consistency model, it is the application that is responsible for using the tools provided by the memory system and signal the appropriate execution points (synchronization points) where these should take place.

2.4 Clusters

Clustering is the use of multiple computers (typically workstations), multiple storage devices and (sometimes) redundant interconnections, to form what appears to users as a single (and sometimes highly available) system [CSG99].

A cluster can be used for load balancing, fault tolerance, high availability and high-performance computing. It is a relatively low-cost form of parallel processing for scientific and other applications that lend themselves to parallel operations. Other benefits include the ability to use commodity hardware. This allows clusters to expand at the sole expense of adding (and interconnecting) new off-the-shelf hardware.

SMPs provide a single address space programming model, where shared memory is equally accessible by all processors. Cluster systems lack a single address space, which is provided by additional support (either in hardware or software). Also, shared memory is distributed among the cluster nodes, and access time is not uniform to all CPUs. Since private memory resides in the local node, its access time is equivalent to that of an SMP. Thus, the use of private memory becomes an important issue to consider, as an efficient alternative to shared memory, whenever the semantics of the program being executed allow it.

2.5 Shared Virtual Memory Clusters

Software Distributed Shared Memory (SDSM) clusters provide users with a shared memory programming abstraction. The shared address space of these systems is usually page-based[3], and it is implemented either by run-time methods or by compile-time methods [LH89, ACD$^+$96, SGT96]. SDSM systems can be cost-effective compared to hardware DSM systems [LC96]. However, there is a large coherence granularity imposed

[3]It is also possible to construct object-based SDSM systems.

by the page size of the underlying virtual memory system that tends to induce false sharing[4] for most applications.

SVM [SHT98] clusters belong to the family of Software Distributed Shared Memory (SDSM) clusters. SVM is an extension of the virtual memory manager of the underlying operating system, where the paging system is modified to implement the mapping between local memory and shared memory on each node. The distinctive feature of SVM is that the operating system page fault exceptions are used to trap memory read and write operations. Shared memory is accessed in the same way as local memory, with remote page faults taking longer time than local ones. Thus, the running application memory reads and writes use the virtual protection mechanisms provided by the operating system.

2.5.1 The Myrinet Cluster, VMMC, GeNIMA and CableS

The cluster system used for this work is comprised of a number of software and hardware layers, as shown in Figure 2.2. These layers allow the cluster memory to be seen as a single address space. The current configuration consists of 8 interconnected dual-processor PC workstations, referred to as nodes, providing a total of 16 processors. For the remainder of this thesis, the above system will be referred to as "the cluster".

The system area network (SAN) layer in this project is Myrinet [BCF$^+$95, Myr03], a point-to-point interconnect, which provides a fast, high bandwidth physical communication layer to the system. The cluster nodes are interconnected by PCI Myrinet network interface cards (NICs) through a Myrinet (Myricom) switch, and by Ethernet NICs to an Ethernet switch.

VMMC [BLS99, Azi02] is the communication layer, which supports internode communication. It consists of a custom NIC control program installed in the Myrinet card, an operating system driver and a user library. It allows access to the NIC in a protected manner, avoiding the need to trap (interrupt) in the kernel for data exchange. When user programs access any memory area, they can only make a reference to a

[4]False sharing occurs when two or more processors access different variables within a page, with at least one of the accesses being a write. In the case of SVM clusters, this may cause coherence messages to be sent to other processors when shared pages are modified.

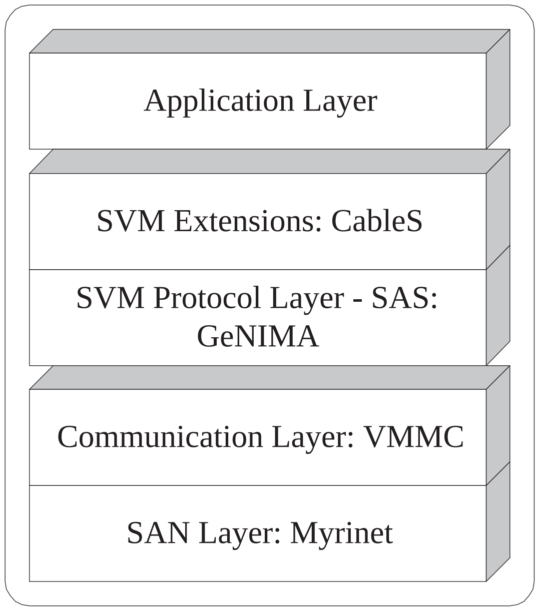


Figure 2.2: Cluster Layers

virtual memory address. VMMC translates those addresses into physical references, using a dynamic mapping scheme [CBD+98].

In addition to VMMC, which provides internode communication, GeNIMA [BJS01, JOY+99] implements the protocol that provides the shared address space abstraction.

CableS [Jam02b, JB02] is a set of libraries that provide thread and memory extensions to the SVM functionality in GeNIMA. It aims at supplying a single system image with respect to thread and memory management on top of the cluster. Its services are accessed through a standard POSIX [IEE90] pthread interface[5]. CableS provides a set of mechanisms that allow the application to use the system resources safely among the different cluster nodes. These mechanisms are:

- *Shared Memory Allocation.* CableS includes routines that allow the application program to allocate memory in a shared area. This memory will be visible to other cluster nodes, subject to the restrictions imposed by the lazy release consistency memory model [KCZ92].

[5]POSIX, which stands for Portable Operating System Interface, is a set of standard operating system interfaces based on the Unix operating system.

- *Locks.* Locks are usually employed as a mechanism for enforcing a policy for serializing access to shared data. However, on the cluster they are also used to guarantee memory consistency.

- *Barriers.* A barrier is a synchronization mechanism for coordinating threads operating in parallel. It requires threads reaching a barrier at different times to wait until all threads reach the barrier.

The application layer in Figure 2.2 is the beneficiary of all the previously mentioned layers. It is a shared memory application that uses POSIX pthreads. However, the application requires some modifications in order to work properly with CableS and the SVM system. These modification are further explored in Section 2.5.2. For the purpose of this work this layer is the Jupiter JVM.

2.5.2 Programming using CableS

CableS provides a standard POSIX interface. This may give the impression that CableS is a set of libraries that can make a POSIX-compliant application suitable to execute on the cluster. This is not the case, because CableS exposes the underlying lazy release memory consistency model of the SVM system to the application. Thus, all shared memory accesses must be protected with synchronization operations, even when synchronization is not necessary.

In the SVM system, locking has a double role. In addition to mutual exclusion, the SVM system locks invoke the routines that keep track of the changes made to the shared memory regions. Memory consistency operations are performed when the acquire and release lock primitives are invoked. This implies that every access to shared memory has to be protected with locks when that memory is read or written. At that point, the SVM system takes over and performs all the necessary memory update operations.

The locking operations guarantee that the latest values are always available to the other threads in the system. It must be noted that this does not immediately refresh values in the remote nodes. When the memory is written, it is marked as modified. When that memory is read after an acquire operation, perhaps from a remote node, the

```
procedure ChangeFlagValue(NewValue)
  Lock(SharedMutex)
  Flag ← NewValue
  Unlock(SharedMutex)
procedure WaitForFlag()
  repeat
    Lock(SharedMutex)
    TempFlag ← Flag
    Unlock(SharedMutex)
  until TempFlag ≠ OldValue
```

Figure 2.3: Use of Locks for Shared Values

system checks for the latest value for the accessed addresses in the whole system, updates it in the local node (if necessary) and returns it to the application. For example, when `pthread_mutex_lock` is called, the standard call for locking a mutex, the SVM system executes the memory consistency routines that search for the most up to date copy of the memory areas the program accesses. After the mutex is unlocked, the memory consistency routines are no longer invoked.

The following examples illustrate situations where the acquisition and release of a mutex signal the SVM system when the memory segments have to be updated from one node to another. Figure 2.3 shows a shared variable which is used as a flag. Its value can only be safely probed if the lock is acquired before reading and writing it. A call to `ChangeFlagValue` made from one thread will signal another thread, which is waiting on `WaitForFlag`, that a certain event has occurred. If access to the flag is not protected in `WaitForFlag`, there is no guarantee that the new value set in `ChangeFlagValue` will be visible if these calls occur in different cluster nodes. Analogously, `ChangeFlagValue` must also protect its access to the flag. Notice that `ChangeFlagValue` and `WaitForFlag` have to access the same lock, `SharedMutex`.

Figure 2.4 shows how a *dynamically allocated* array must be protected before each or all of its values are set. Then, when these values are accessed, perhaps by another thread, the reading operation also has to be protected with locks. This is true *even if the array is defined as read-only*, because the appropriate use of locks guarantees

```
procedure CreateAndSetArray()
  Array ← CreateArray()
  Lock(ArrayMutex)
  for each element in the array do
    Array[element] ← Value
  end for
  Unlock(ArrayMutex)
procedure ReadArray(Array,element)
  Lock(ArrayMutex)
  ReturnValue ← Array[element]
  Unlock(ArrayMutex)
```

Figure 2.4: Use of Locks for Read-Only Arrays

that the SVM system updates the memory values when or if it is required. Notice that the memory allocation routines, which are called from `CreateArray`, are not protected. Section 4.2 provides further details on the usage of locks in the SVM system.

2.6 Related Work

This section presents various projects that aim at running a JVM on a cluster.

2.6.1 JavaParty

JavaParty [PZ97] is a cooperating set of JVMs that run on nodes of a distributed system. It is implemented as a preprocessor to the Java compiler and a runtime system. JavaParty extends the Java language with a new class modifier, called `remote`. This modifier is used to explicitly indicate to the preprocessor when classes can be spread among remote nodes. The precompiler and the runtime system ensure that access to remote class fields or methods are handled by Remote Method Invocations (RMI). JavaParty aims to hide the increased program size and complexity of explicitly using RMI or sockets for parallelization of Java programs on distributed systems. Object migration in JavaParty is supported through the use of proxies, with no replication, which slows down access to those objects that are shared between threads. The performance of JavaParty is reported to be comparable to that of RMI.

Unlike Jupiter, JavaParty requires that Java programs be modified. Jupiter supports the standard Java thread model, while JavaParty requires that Java programs be expressed using the special `remote` class modifier, which is not part of the Java standard. As a consequence, Java programs lose portability. Furthermore, the runtime component of JavaParty is implemented on top of RMI, which introduces considerable overhead. In contrast, Jupiter uses a shared address space memory model.

2.6.2 JESSICA/JESSICA2

JESSICA [MWL00] is a middleware system which creates a single system image of a JVM on a cluster. It is based on the Kaffe JVM [Wil02]. It uses Treadmarks [ACD$^+$96] for DSM support, as well as the socket interfaces provided by Linux. JESSICA is similar to Jupiter in many respects. Both support the execution of Java programs on a cluster using DSM. However, JESSICA is not focused on obtaining a scalable JVM on the cluster. The main purpose of the system is abstraction from the cluster and appropriate load balancing. Thus, unlike Jupiter and other projects, JESSICA successfully implements I/O support, though it is centralized.

There are a number of other differences between JESSICA and Jupiter. In Treadmarks, the memory pages allocated in different nodes may not reside at the same addresses. This affects references to objects stored in the heap, which have to be translated when passed between nodes. Jupiter does not suffer from such translation problems. Also, JESSICA does not take full advantage of the relaxed properties of the Java Memory Model. The implementation performs a DSM-lock acquire and release operation for every object access. JESSICA also uses a double locking mechanism for the update of shared objects. First, an object lock is requested, then, a DSM-lock, resulting in some extra traffic and delays. Another difference is that some system requests, such as thread signaling mechanisms (`wait()` and `notify()`) and mutex operations (`lock()` and `unlock()`), are centralized in the main node. While this simplifies implementation, it is a potential bottleneck. Jupiter follows a decentralized schema, provided by the underlying SVM system.

Only three applications were used for the performance evaluation of JESSICA, for a cluster configuration of up to 8 processors. For RayTracer and SOR, the speedup obtained in Jupiter on the cluster is significantly better than that in JESSICA, which suffers from a significant slowdown introduced by the distributed object model and DSM systems. The slowdown ranges from 3% to 131% for a single worker thread.

JESSICA2 [ZWL02] is the extension of the JESSICA project. This prototype follows a different approach, using a JIT compiler, while maintaining much of the properties of the previous architecture. JESSICA2 differs from JESSICA in three respects. JESSICA2 uses a JIT compiler; instead of a page-based DSM system, JESSICA2 uses object-based DSM (which adheres to the Java Memory Model); and the master and worker nodes communicate through TCP connections. The experimental evaluation on the cluster is based on three Java applications, with a configuration of up to 8 nodes. The performance of JESSICA2 is considerably better than that of JESSICA, because of the use of the JIT compiler. However, the speedup is worse.

2.6.3 Java/DSM

Java/DSM [YC97] is an implementation of a JVM based on Sun's JDK 1.0.2 [SUN03] on top of the Treadmarks [ACD$^+$96] distributed shared memory system. It runs on a cluster of heterogeneous computers, providing a single system image and hiding hardware differences. Java/DSM aims at integrating heterogeneous environments, using a translation mechanism to convert data of the same type between different hardware platforms. Java/DSM requires the threads in the Java program to be modified to specify the location to run. It is unclear if it was ever developed to completion.

Java/DSM bases its memory management on Treadmarks, and thus, it suffers from the same address translation drawbacks described earlier for JESSICA [MWL00]. Java/DSM differs from Jupiter in that its focus is to integrate heterogeneous systems. It also requires that threads specify the location where they run, thus making Java programs not portable. The synchronization primitives `wait` and `notify`, which are important for developing multithreaded Java applications, are reported not to work between threads running on different nodes. Jupiter supports these primitives transparently.

The focus of the experimental evaluation of Java/DSM is to compare the difficulty of programming the application using Java/DSM and RMI. Thus, the experiments are based on a distributed spreadsheet that supports collaboration between users.

2.6.4 cJVM

cJVM [AFT99, CJV00, AFT$^+$00] is a JVM based on Sun's JDK 1.2 [SUN03] which provides a single system image of a cluster, where the nodes are interconnected with a Myrinet [Myr03] network. This project focuses mostly on exploiting optimization strategies based on Java semantic information obtainable at the JVM level, which is used to determine the most efficient way to handle remote object access. cJVM does not support all the core classes that have native methods.

In contrast to Jupiter, cJVM uses message passing (MPI) for communication, which requires extensive modifications to the JVM, and introduces efficiency and complexity problems. Jupiter follows a shared memory approach. The object model is also different from Jupiter's. In Jupiter memory pages that store objects are shared between the nodes. However, cJVM supports distributed access to objects using a master-proxy model. This causes the thread stack to be distributed across multiple nodes, making the load distribution across the cluster dependent on the placement of the master objects.

The implementation was only tested to up to 4 cluster nodes, running one thread per node. Unfortunately, the applications used for the experimental evaluation are different from those used in Jupiter.

2.6.5 Kaffemik

Kaffemik [AWC$^+$01] is a cluster-enabled JVM based on the Kaffe JVM [Wil02]. It provides a single machine abstraction of a cluster, with hardware support for DSM offered by a Scalable Coherent Interface (SCI) [IEE93]. Kaffemik is in its early stages of development.

Unlike our cluster-enabled Jupiter, the current implementation of Kaffemik does not support a release consistency memory model, object replication or caching; rather, it relies on cache coherence provided by SCI. Therefore, frequent access to remote objects as well as array operations suffer from performance loss. This also creates some problems at thread creation, since part of the Java Thread object remains on the original node instead of being sent to the remote node. Jupiter fully supports the release memory consistency model, with data replication handled at the SVM level. The optimization of the execution is done at the Java application level, modifying it for enhancing memory locality. Jupiter is able to run unmodified Java applications efficiently.

The performance of Kaffemik is only reported for up to 3 nodes, running one thread per node. For this purpose, non-optimized and optimized versions of RayTracer were used. The speedup is comparable to that of Jupiter only for the optimized version.

2.6.6 Hyperion

Hyperion [ABH^{+}01] provides a single system image of a JVM on a cluster of workstations. The cluster supports two interconnection networks: SCI [IEE93], which uses the SISCI protocol, and Myrinet [Myr03], which uses message passing (MPI). It executes on PM2, a distributed multithreaded run-time system that provides a generic DSM layer. Hyperion comprises a Java-bytecode-to-C translator and a run-time library for the distributed execution of Java threads.

A distinctive feature of Hyperion, which contrasts with Jupiter, is that it transforms Java bytecodes into C. The C code is then compiled and linked with Hyperion's run-time library before it can be executed. Also, Hyperion uses an object-based DSM system. Hyperion keeps a single consistent master copy of an object. The nodes use their locally cached copy of objects, which are explicitly written back to the master copy at Java synchronization points using RPC. During the transfer, the thread blocks and performance is likely to suffer if objects are modified frequently. In contrast, Jupiter uses a page-based DSM system, and pages are flushed following a lazy release memory consistency protocol. Hyperion lacks support for classes containing native methods, which must be manually converted in order to work.

The evaluation of Hyperion was performed to up to 8 processors using a single specially-developed application.

CHAPTER 3

Multithreading Extensions

This chapter discusses the design and implementation of multithreading in Jupiter. Section 3.1 gives an overview of the functionality that a multithreaded JVM must provide in order to correctly execute multithreaded Java programs in accordance with the JVM Specification [LY99]. Section 3.2 presents the approach taken in Jupiter to implement these requirements. Section 3.3 explains the changes in the initialization routines of Jupiter, which involve the integration between Jupiter and the ClassPath libraries. Section 3.4 describes the use of synchronization operations to avoid race conditions in the various Jupiter shared data structures. Finally, Section 3.5 details the support for multithreaded quick opcodes.

3.1 Multithreading in JVMs

The JVM Specification [LY99] and the `Thread` class interface require that a JVM provide three general types of functionality, relevant to multithreading: *thread handling*, *synchronization*, and *wait-sets and notification*.

3.1.1 Thread Handling

Thread handling refers to all the operations that can be performed on threads. A JVM provides threading functionality to Java programs through the interface of the `Thread` class, which is the only mechanism available for this purpose. These operations fall into the following categories:

- *Definition of thread creation properties.* Java threads can be set to behave as non-daemon threads or as daemon threads. Non-daemon threads can be joined on termination, while daemon threads cannot.

- *Definition of thread execution priorities.* Priorities are attributes that determine the relative importance given to threads for CPU allocation, thus allowing some threads to have precedence over others. Java allows for this attribute to be modified.

- *Thread startup.* This is the operation by which a thread begins its execution. In Java, threads are created in two steps. First, a `Thread` object (a subclass of `Thread` or a class that implements the `Runnable` interface) must be instantiated and configured (priority and daemon properties). After that, the thread must be launched calling the `start` method.

- *Thread joining.* A Java thread has the mechanisms to wait for another thread to complete its execution. This is called thread joining, and can only be performed on non-daemon threads. It accepts an optional time-out parameter.

- *Thread execution state verification.* This operation probes a thread to check if it has completed its execution or not.

- *Thread sleep.* The currently executing thread can be forced to temporarily cease execution for a period of time, which may make it also relinquish the CPU.

- *Thread interruption.* Some thread operations, such as thread sleep and join, can be interrupted. This allows the thread to resume execution.

- *Processor yield.* The currently executing thread can be forced to relinquish the CPU. In this case, other threads in the system will compete for the CPU, which allocation will be determined by the scheduler.

- *Thread identification.* A `Thread` object can be uniquely identified, thus providing the means to distinguish it from others. At this level, the interface only requires that a `Thread` object have a name.

Notice that there are no operations for *thread termination*. The reason is that Java threads cannot be killed. The `stop` method, initially intended for this purpose, has been deprecated [SUN03].

3.1.2 Synchronization

Synchronization is the process by which two or more threads coordinate their execution or the use of shared resources or common data structures. For this purpose, the Java language allows method definitions as well as blocks of code to be marked with a special `synchronized` clause. This functionality requires support for two opcodes, as specified in Chapter 6, Section 7.14 and Section 8.13 of the JVM Specification [LY99]. The `monitorenter` and `montitorexit` opcodes are invoked when a `synchronized` section is entered or exited respectively. They must guarantee mutual exclusion using the locks associated with Java object instances [LY99]. If the `synchronized` code is marked as `static`, then the lock associated with the `Class` object that represents the class in which the method is defined is used instead of object instance locks.

3.1.3 Wait-Sets and Notification

Each Java `Object` must have an associated *wait-set and notification* mechanism [LY99]. This mechanism is used in the `Object` class for the implementation of the methods: `wait`, `notify` and `notifyAll`, for which a JVM must provide the following support:

- `wait`. This method causes the calling thread to enter a *waiting-for-notification* state and be placed in the *wait-set* of the corresponding object. The thread's activity is suspended, and it remains so until it receives a *notification*. This interacts with the scheduling mechanism for threads. When a thread is in a waiting-for-notification state, it can be forced to relinquish the CPU and, therefore, is not scheduled for execution. The `wait` method has to be called from within `synchronized` code. The lock held on the object, regardless of the level of recursion (i.e., successive calls to `monitorenter`), is released, but locks on other objects, if any, are retained [NBF96].

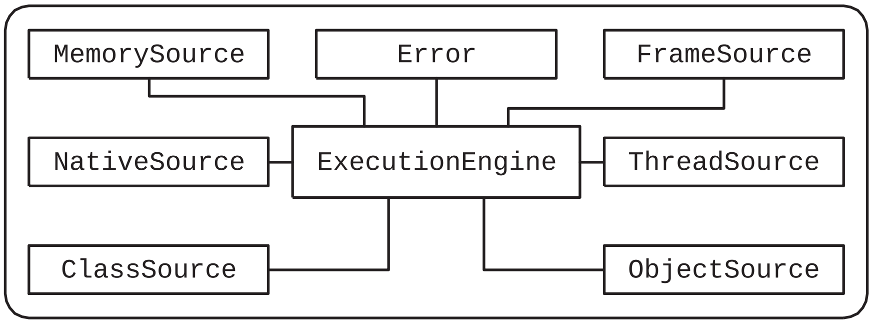


Figure 3.1: Single-Threaded Design

- `notify`. This method sends a notification to one thread in the wait-set of the object on which the method containing `notify` is applied. This causes the thread to exit from the wait-set so that it can be rescheduled for execution. Since the thread had executed the `wait` from inside a `synchronized` method, it will be competing again to obtain the object lock, which it had previously released. When it finally reacquires the lock, it does so with the same recursion level it had before the execution of the `wait` method. The `notifyAll` method is a special case, where the notification is sent to all the threads in the wait-set. The `notify` and `notifyAll` methods have to be called from within `synchronized` code.

3.2 Multithreading in Jupiter

The overall structure of Jupiter is shown in Figure 3.1. The `ExecutionEngine` is the heart of the opcode interpreter. It integrates and coordinates the calls to the different components of the system. `MemorySource` provides the interface for memory allocation. `Error` handles the error conditions that may appear. `FrameSource` allocates frames in the thread stack. `NativeSource` manages the calls to native methods. `ThreadSource` is used for spawning child threads. `ClassSource` controls the creation of Java classes. Finally, `ObjectSource` is responsible for creating Java objects.

We elected to implement multithreading in Jupiter by having a completely functional JVM in every thread, where each thread shares some of its components with other threads, but has other components that remain private. In this case there is an addi-

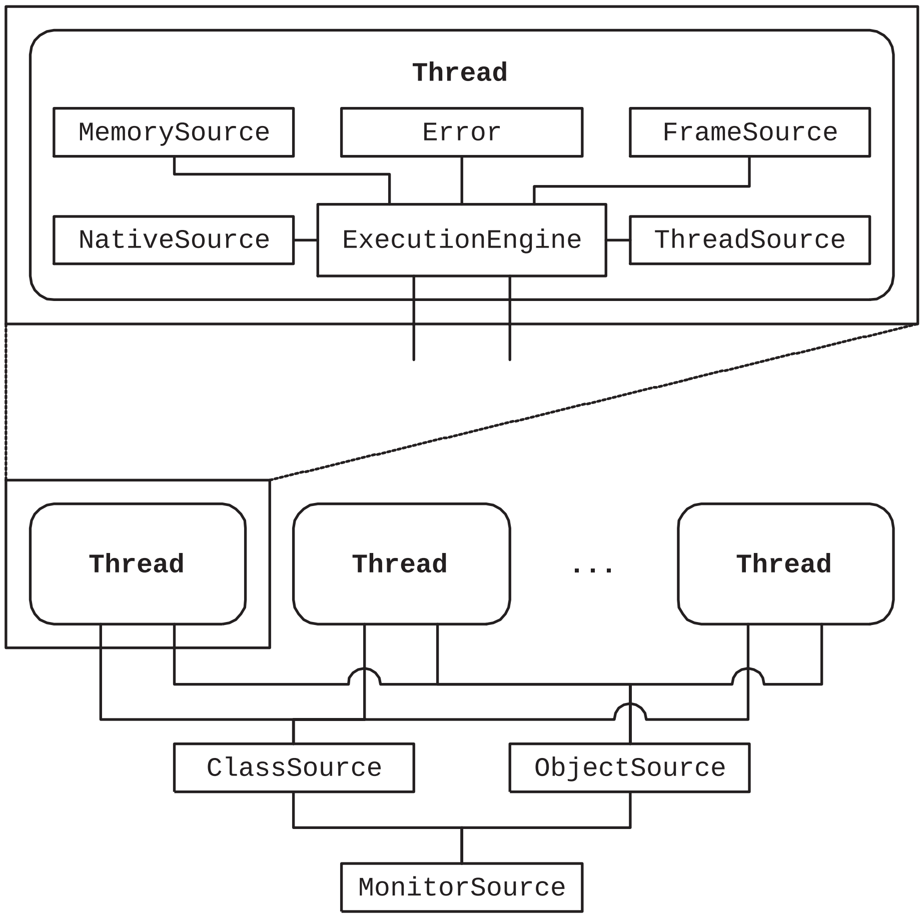


Figure 3.2: Multithreading Design

tional module, `MonitorSource`, which creates `Monitor` structures. This architecture is shown in Figure 3.2. As the figure shows, `ExecutionEngine`, `MemorySource`, `Error`, `FrameSource`, `NativeSource` and `ThreadSource` remain private to the thread, while `ClassSource`, `ObjectSource` and `MonitorSource` are shared. The rationale for these design decisions are listed below:

- `ExecutionEngine`. Having independent JVMs in each thread allows this module to remain private, since this comprises the opcode interpreter.

- `MemorySource`. Since memory allocation is performed through the standard POSIX call `malloc`, it can be independently invoked by all threads, thus making this module private. In an SMP, this same call is used to allocate both memory that will be private to a thread or shared among threads. The allocated shared memory will be equally accessible by all threads. A similar situation occurs for the cluster case, which is described in Section 4.3.1.

- `Error`. This module is made private because each thread can manage its error conditions independently of the other threads.

- `FrameSource`. Since the stack always remains private to the thread, this module is not required to be shared.

- `NativeSource`. In Jupiter, the management routines for native calls can be called by more than one thread simultaneously. Each thread can safely construct its own references to native calls, since the internal Jupiter structures that are used do not require to be shared. Thus, this module is made private to each thread.

- `ThreadSource`. This module is used for thread creation and keeping track of non-daemon child threads, which must be joined before a JVM finishes its execution. These operations can be managed independently between threads, since each thread waits for its own child threads before finishing, thus allowing this module to remain private. The joining of threads is explained in further detail in Section 3.2.6.

- `ClassSource`. The JVM Specification [LY99] requires that loaded Java classes be accessible to all threads. Therefore, Jupiter creates classes in the global heap. The `MonitorSource` module and the internal data structures that store references to such classes (which are used to access them) must be shared, thus making `ClassSource` a shared module.

- `ObjectSource`. Even though it is not required by the standard, Jupiter conservatively creates Java objects in the global heap. This allows them to be shared among threads. Similar to the case of `ClassSource`, the `MonitorSource` module, which is used for the creation of these objects, must be shared, making `ObjectSource` a shared module.

- `MonitorSource`. This module relies on the use of a single special instance of `ThreadSource`, which is created in the main thread. Therefore, a single instance of `MonitorSource` is created for the system, which is shared among threads.

This high level design anticipates the need to enable the multithreaded Jupiter to run on the cluster of workstations, which will be described in Chapter 4. Having independent, decentralized components reduces the use of shared memory and the need for synchronization operations on the cluster, and thus, it is likely to improve performance.

The remainder of this section discusses the development of the multithreading modules, the design decisions made in the process, the problems faced and how they were solved.

3.2.1 Threading Modules

The design of the threading modules was planned to accommodate the needs of two other modules, the Java class libraries and the native thread libraries. On the one hand, there are functionality requirements imposed by the Java `Thread` class, the `Object` class (with support from the ClassPath class called `VMObject`) and the implementation of some opcodes. On the other hand, there is the functionality provided by the native thread libraries on the target system. Their requirements and services do not match one to one, although they aim at providing equivalent functionality.

Therefore, the main responsibility of the threading modules is to act as a simple interface between these modules, avoiding conflicts and constraints between the requirements of the upper layer and the services provided by the lower layer.

Furthermore, it was necessary to provide an interface that would abstract away the details of the native thread libraries. This requires that the interface do not have complex functionality that some native libraries may not be able to supply, and it must only provide the smallest set of functions necessary to implement multithreading in Jupiter.

The threading modules implement two abstraction layers, `Thread` and `Monitor`, and `ThinThread` that link the Java class libraries and the native thread libraries, as shown in Figure 3.3.

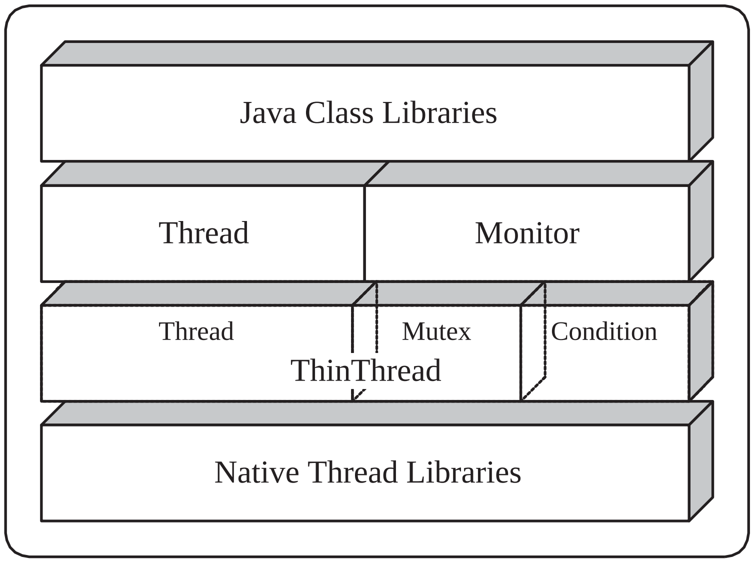


Figure 3.3: Multithreading Modules

`Thread` and `Monitor` provide services to the Java class libraries, whose interface was specified by ClassPath and adheres to the JVM Threading Specification, which was described in Section 3.1. `Thread` and `Monitor` rely on `ThinThread`, which in turn abstracts the native thread libraries.

The problem and solutions of over-constraint interfaces were introduced in the discussion on *design for flexibility* in the original work on Jupiter [Doy02]. In the design of the `Thread`, `Monitor` and `ThinThread` modules, it was attempted not to change the interfaces defined in the original Jupiter JVM. These interfaces were designed with the purpose of making Jupiter flexible and modular.

An important element in designing the interface was to isolate different functionality in several layers, gradually accommodating the abstract interface of the native thread libraries into the needs of Jupiter. While a single interface could have been used, a layered approach is a design technique that simplifies the implementation and improves maintainability. In this case, instead of having a single and complex module, the functionality is divided into smaller tasks and placed in separate modules. The functionality on each layer is supported on simple and well defined functions provided by the layer below.

`Thread` and `Monitor` are designed to enhance portability. They provide functionality that may not always be available in the lower layers or was restricted at `ThinThread`, such as recursion, protection against spurious wake-ups, thread joining and priority setting. In those cases where it can rely on the lower layers, it supplies a uniform access interface.

3.2.2 `ThinThread`

`ThinThread` encompasses the minimal set of functionality that Jupiter requires for thread handling. It defines a standard and simple interface upon which `Thread` and `Monitor` can be built. As its name indicates, this is a thin layer, thus it does not add functionality to the native thread libraries. Instead, it is intended to simplify the replacement of native libraries without the need of any major redesign in the upper layers of the threading modules, as was described above.

`ThinThread`'s interface was inspired in the POSIX thread libraries, because they are well known and standard, in particular for Unix systems. `ThinThread` is divided in three sections:

- *Thread operations.* These include thread startup, execution priority setting (limited to priority values defined internally), processor yield, sleep, interrupt and self identification.
- *Mutex operations.* These comprise acquire, release and try. It is possible to implement custom locks at this level using `testandset`.
- *Condition Variable operations.* These include wait and waitwithtimeout on conditions, and signal and broadcast of completed operations. It must be noted that this functionality does not necessarily have to rely on condition variables. It can be implemented using shared variables or signals.

This functionality, and the rationale for including such functionality in this layer are described below:

- *Definition of thread creation properties.* The option of setting the daemon property of a thread was not included in the `ThinThread` module. The reason for this decision is to avoid relying on this feature provided by POSIX, which makes `ThinThread` as simple as possible. Therefore, only detached (daemon) threads can be created, and thread joining is controlled explicitly by Jupiter (see Section 3.2.6 for further details). Notice that this gives freedom of implementation because the interfaces do not rely on any specific property of the native thread libraries.

- *Definition of thread execution priorities.* The priority setting at the JVM level has to be isolated from the setting at the native library level. The reason is that both use different priority scales and `ThinThread` must remain as abstract as it is possible in this respect. Thus, `ThinThread` has an independent range of thread priorities that is converted to a valid range within the values that the native library can handle.

- *Thread identification.* The need for thread identification at a lower level extends the requirements of the `Thread` class. At this point, it involves the internal identification used, at system level, by the native thread libraries. This is necessary for operations such as priority setting, thread interruption and self thread identification. The latter is used to give support to the implementation of thread startup and `Monitor`.

- *Thread execution state verification.* The native thread libraries do not provide primitives for checking the execution state of threads. It is the operating system that keeps track of this information. In this case, an internal Java field that will store this state information is kept in the Java `Thread` class. When a thread starts and terminates, its value is updated using calls to `ThreadSource` and `Thread` that ultimately use the thread identification mechanisms in `ThinThread`.

It must be noted that some of the functionality required by the `Thread` class is programmed in the Java class libraries, and there is only specific functionality that Jupiter must provide this class through the JNI (Java Native Interface).

`ThinThread` limits locks to the *fast* (i.e., non-recursive) type [IEE90]. This design decision avoids relying on the recursive feature supplied by POSIX, thus making `ThinThread` simple and the system libraries interchangeable. This restriction proved useful when enabling Jupiter to run on the cluster since CableS does not provide recursive locks.

`ThinThread` also allows condition variables to have spurious wake-ups[1], which are also permitted by the underlying system libraries. The reason for the system libraries to experience such behavior is that a completely reliable once-and-only-once wake-up protocol at that level can be expensive. Furthermore, it is considered that spurious wake-ups promote good programming practices [HPO01]. This behavior is allowed at the `ThinThread` interface because it simplifies the calls to the underlaying thread library.

3.2.3 `Thread` and `Monitor` Modules

`Monitor` provides the synchronization and the wait-sets and notification functionality required by the JVM Specification [LY99]. It is a structure associated with Java objects and classes, which is used for the implementation of `synchronized` code. The design allowed for both synchronization and wait-sets to be included in the same module because they base their mutual exclusion requirement on the same mutex. `Monitor` provides the following interfaces: Enter, Exit, Try (monitor operations), Mine (monitor ownership), Wait/WaitWithTimeout and Notify/NotifyAll (wait-sets and notification). In Jupiter, `Monitor` is implemented using mutexes and condition variables. However, there are two aspects in `Monitor` design that had to be considered:

- *Recursion.* The JVM Specification [LY99] allows for monitors to be called recursively. However, the `ThinThread` interface does not allow mutexes to provide this property. Only fast (non-recursive) mutexes are provided. This restriction requires that `Monitor` be responsible to handle this case. This will be described in Section 3.2.4.

- *Spurious wake-ups.* The native thread libraries, as well as the `ThinThread` layer, do not enforce that condition variables, used for `Monitor` waiting, return only after a successful notification. This means that there is no guarantee that the return from a `wait` call is due to a `signal` or `broadcast` call [OPE97]. The spurious

[1]A spurious wake-up is a return from a waiting call on a condition variable predicate, even if the predicate remains false.

wake-up demands that `Monitor` give extra support to eliminate the problem. This will be explained in Section 3.2.5.

`Thread` presents the necessary interfaces for thread handling. These are as follows: Start (thread startup), Set/Get Priority (priority setting), Interrupt (operation interruption), Self Identification (thread identification), Yield (processor yield), Sleep and Join/JoinWithTimeout (thread joining). This module also presents two distinct aspects that required further consideration and analysis:

- *Thread startup.* In contrast to the two-step thread creation in Java, once the thread libraries create a thread, it immediately begins its execution. This is done through the call to a generic function which receives the thread function as a parameter. Thread creation in ClassPath works differently. It calls a native function for creating the system thread when the `Thread` object constructor is invoked. This is designed for the creation of the system thread without triggering its execution. Then, a different native function is called when the `start` method is invoked, which is intended to start the execution of the previously created system thread. This accommodates thread libraries that may work in two steps. This difference requires a careful design, which maintains Java semantics, when deciding where to execute the thread creation function[2] from the thread library. One possibility was to create the thread during the creation of the `Thread` object, and force the thread to spin until it is signaled to continue from within the `start` method. However, the alternative elected was to invoke the thread creation function in the `start` method, which creates and starts the system thread in a single step.

- *Thread joining.* Since only daemon threads are supported by the `ThinThread` layer, Jupiter must provide its own thread joining mechanisms. The solution to this problem is detailed in Section 3.2.6.

As the requirements of the Java class libraries are unlikely to change drastically in the future, the interfaces of `Thread` and `Monitor` were designed to match them. This way, the lowest level of the Java class libraries remains simple.

[2]System thread creation is implemented as calls to the Jupiter modules `ThreadSource` and `Thread`.

3.2.4 `Monitor` Support for Recursive Calls

Monitors can be called recursively. This means that a thread that is already holding a monitor can request to use it again. This functionality can be implemented using recursive locks [LY99]. However, since the `ThinThread` interface restricts the use of locks to the fast (non-recursive) type, this had to be tackled in `Monitor`. This was achieved by direct control of recursion and ownership of the monitor, using functions for thread identification, provided by the `Thread` module.

An outline of the algorithm needed by `Monitor` to support recursive calls is shown in Figure 3.4. There are two important aspects to consider. In the method `enter`, if the current thread already holds `Monitor` (and, therefore, the lock), the only required action is to increment the locking count `MonitorCount`. Otherwise, the thread requiring exclusive access to `Monitor` must contend to acquire the lock. In the method `try` similar steps are taken. It must be ensured that the internal count `MonitorCount` shows a complete unlock before `Monitor` can be given to another thread. In the method `exit`, the count is decreased until it reaches 0. At that point, no thread owns the monitor, and therefore the lock can be released. Normally, the Java compiler guarantees that there are no more unlocking than locking operations [LY99].

3.2.5 `Monitor` and Spurious Wake-ups

The `Monitor` module implements the wait-set and notification mechanism for Java objects and classes. The synchronization structures used to support this mechanism are condition variables, because they best match the functionality requirements. A condition variable is a synchronization object which allows a thread to suspend execution until some associated predicate becomes true [OPE97].

Consequently, an interface for accessing condition variables was added at the `ThinThread` layer. However, the native thread libraries as well as the `ThinThread` module are allowed to produce spurious wake-ups on condition variables. This behavior should not occur in the implementation of wait-sets. As a result, it was not possible to use condition variables directly for the wait-sets. It was required that their behavior be adapted to the needs of `Monitor`. A solution was implemented at `Monitor` level,

```
procedure enter()
  if MonitorOwner $\neq$ CurrentThread then {If this thread is not the
  owner of the monitor, contend for its exclusive access}
    Lock(MutexMonitor)
    MonitorOwner $\leftarrow$ CurrentThread {Keep monitor ownership
    information}
  end if
  MonitorCount $\leftarrow$ MonitorCount + 1
procedure exit()
  MonitorCount $\leftarrow$ MonitorCount - 1
  if MonitorCount = 0 then {The thread exited the monitor
  completely}
    MonitorOwner $\leftarrow$ NoThread {No thread owns the monitor}
    Unlock(MutexMonitor)
  end if
function boolean try()
  MutexLocked $\leftarrow$ FALSE
  if MonitorOwner $\neq$ CurrentThread then
    MutexLocked $\leftarrow$ Try(MutexMonitor)
    if MutexLocked then
      MonitorOwner $\leftarrow$ CurrentThread
      MonitorCount $\leftarrow$ MonitorCount + 1
    end if
  else
    MonitorCount $\leftarrow$ MonitorCount + 1
  end if
  return MutexLocked
```

Figure 3.4: Monitor Support for Recursive Calls

which has the added advantage of allowing a simple integration of the functionality of the wait-set with that of the lock associated with each Java `Object` and `Class`. Thus, all the Java synchronization structures could be centralized in one module.

An outline of the wait-sets and notification mechanism that handles spurious wake-ups is shown in Figure 3.5. The number of waiting threads and the notifications received are counted. The call to `WaitOnCondition` in the method `wait` is protected from spurious wake-ups by the count on the notifications. It must be noted that, since the `wait`, `notify` and `notifyAll` methods have to be called from within `synchronized` code, which guarantees mutual exclusion among them, it is safe to increment and decrement the shared variables `WaitCount` and `NotifiedCount`.

```
procedure wait()
  MonitorDepth ← GetMonitorDepth(Monitor)-1
  for MonitorDepth do {Release the Monitor}
    Exit(Monitor)
  end for
  WaitCount ← WaitCount + 1 {One more thread enters the wait state}
  repeat
    WaitOnCondition(Condition, Monitor) {Wait on Condition and
    release the lock in Monitor.  When it returns, the lock in
    Monitor was reacquired}
  until NotifiedCount ≠ 0 {There must be a notification present to
  continue.  This protects from spurious wake-ups}
  WaitCount ← WaitCount - 1 {The notification was received}
  NotifiedCount ← NotifiedCount - 1 {The notification was processed}
  for MonitorDepth do {Reacquire the Monitor}
    Enter(Monitor)
  end for
procedure notify()
  if WaitCount > NotifiedCount then {Protect NotifiedCount from
  notifications sent when no threads are waiting}
    NotifiedCount ← NotifiedCount + 1
  end if
  SignalCondition(Condition) {Signals can be sent even if no threads
  are waiting}
procedure notifyAll()
  NotifiedCount ← WaitCount {Notify all waiting threads}
  BroadcastCondition(Condition) {Broadcasts can be sent even if no
  threads are waiting}
```

Figure 3.5: Wait-Sets and Notification that Handle
Spurious Wake-ups

```
procedure ThreadFunction(Arguments)
  MutexDone ← ReadArguments(Arguments.MutexDone)
  DoneFlag ← ReadArguments(Arguments.DoneFlag)
  ConditionDone ← ReadArguments(Arguments.ConditionDone)
  RunThread(Arguments) {Execute the thread using Arguments}
  Lock(MutexDone)
  DoneFlag ← TRUE
  BroadcastCondition(ConditionDone)
  Unlock(MutexDone)
procedure join()
  Lock(MutexDone)
  while DoneFlag ≠ TRUE do
    WaitOnCondition(ConditionDone, MutexDone) {Wait on ConditionDone
    and release MutexDone.  When it returns, MutexDone was
    reacquired}
  end while
  Unlock(MutexDone)
```

Figure 3.6: Thread Joining

3.2.6 Thread Joining

`ThinThread` only supports daemon threads. Thus, a joining mechanism was implemented in `Thread`, as outlined in Figure 3.6, which involves the use of a lock, a notification flag, and a condition variable to ensure that threads are joined properly.

The use of a condition variable avoids busy waiting. This makes the call to `join` block until the termination event is signaled. A call to `BroadcastCondition` is used to signal the termination event. This is because, at a given point in the execution, there may be more than one thread waiting to join on the same thread. The use of a flag (`DoneFlag`) protects the condition variable `ConditionDone` from spurious wake-ups.

3.2.7 Waiting on Non-Daemon Threads

The JVM Specification [LY99] states in Section 2.17.9 the conditions under which the JVM can terminate its execution:

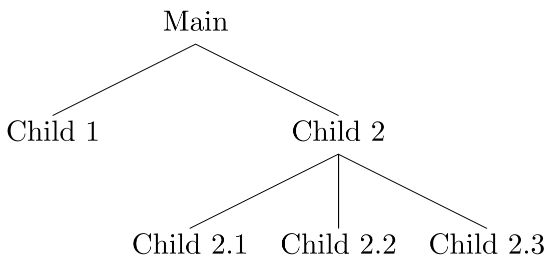


Figure 3.7: Thread Spawn Relationship

> *"The Java virtual machine terminates all its activity and exits when one of two things happens:*
>
> - *All the threads that are not daemon threads terminate.*
> - *Some thread invokes the* `exit` *method of class Runtime or class System, and the exit operation is permitted by the security manager."*

Thus, it is necessary to guarantee that the multithreaded version of Jupiter treats non-daemon threads according to the specification. This implies that Jupiter needs to maintain a repository where references to running child threads are stored.

It was elected to use a decentralized structure, with each of its fragments kept private to each thread. Each thread keeps track of all the non-daemon child threads it spawns. When the thread completes its execution, it performs a last call that verifies the status of all its child threads. This call will not return until all of them are reported to have completed.

Therefore, the spawn relationship among threads may be viewed in the form of a tree, as illustrated in Figure 3.7. This tree is only used for joining the threads, and does not impose any kind of hierarchy among them. As indicated by the figure, Child 2 waits for Child 2.1, Child 2.2 and Child 2.3 to exit before finishing. The Main thread, does not terminate, and therefore finalize the execution of the program, until Child 1 and Child 2 complete their execution. This solution has the advantage of avoiding a centralized repository, which would require expensive lock operations to be accessed by multiple threads.

3.3 Initialization Routines

Jupiter's main thread is responsible for the initialization of the whole JVM. This entails:

- Creation of the first instances of the key objects that comprise the JVM, such as: memory management (`MemorySource`), class management (`ClassSource`), object management (`ObjectSource`), stack management (`FrameSource`, `Context`), opcode execution (`ExecutionEngine`) and opcode profiler, if enabled.
- Initialization of the Java class libraries, ClassPath.

Extending Jupiter to support multithreading involves the addition of two more tasks:

- Creation of the `Monitor` manger (`MonitorSource`) and `Thread` manager (`ThreadSource`).
- Explicit initialization of the Java objects used for multithreading, the `Thread` and `ThreadGroup` Java classes, which are part of the Java core classes, in the *java.lang* package [CWH00].

Every thread executing in a JVM has to be associated with a Java `Thread` object. Threads in the Java program have to be created using `Thread`, which handles all the threading operations. When Jupiter obtains control on the native thread creation routines, the connection between the native thread and `Thread` is made. This association is used from within `Thread` in the native method `currentThread`, which is part of the implementation of the construction of the `Thread` object, priority settings, property settings, sleep, thread identification (used for groups and subgroups) and interruption [GNU03].

The main thread is an exception, because it is implicitly created at JVM startup, when none of the routines in `Thread` was invoked, or the ClassPath libraries initialized. Therefore, it is necessary to force the creation of an initial Java `Thread` object as part of ClassPath initialization. This is achieved by calling the appropriate Jupiter methods for loading and initializing classes, and creating an instance of the `Thread` object afterwards. Finally, the association with the main thread is explicitly made.

For its construction, the `Thread` class needs the support of the `ThreadGroup` class. `ThreadGroup` represents sets of threads and allows for some operations to take place on all the members as a group. Its initialization and instantiation was done in a similar way as `Thread` during Jupiter startup.

3.4 Mutable Data and Synchronization

The original Jupiter JVM is limited to the execution of single-thread Java applications. Thus, its code did not provide for sharing and protection of its internal data structures among multiple threads. Jupiter internal structures represent elements in the Java program, such as classes and objects. Some of these structures can be simultaneously accessed by more than one thread. Thus, it is necessary to protect access to these structures with locks to avoid race conditions.

It must be noted that only those Jupiter modules that could be affected by such race conditions are protected. Jupiter provides the appropriate synchronization support for Java programs to use, which are responsible for maintaining consistency in their own data. The Jupiter modules affected by race conditions are:

- *Quick Opcodes.* Java methods can be optimized at runtime by replacing the implementation of some of its opcodes, using a technique called quick opcodes [LY96, Doy02]. Since a Java opcode may be invoked simultaneously by more than one thread, the opcode and its parameters become shared data. If the opcode is changed, it must be protected with appropriate synchronization calls to prevent the interpreter from reading invalid parameter values while the change is taking place. The handling of quick opcodes is exposed in more detail in Section 3.5.1.

- *Class Reflection.* Reflection allows an executing Java program to examine itself and manipulate some of its internal properties. It does so through the creation of Java objects that depict different aspects of the program such as classes, fields, constructors and methods [SUN03]. Since reflection classes can be created and manipulated simultaneously by more than one thread, they become shared and therefore must avoid race conditions. Synchronization on access and manipulation is guaranteed by Java semantics, but the consistency of the internal data structures that represent reflection classes must be ensured by Jupiter.

- *Class Creation.* Class initialization has to follow a strict protocol, detailed in Section 2.17.5 of the JVM Specification [LY99]. This operation can be requested by multiple threads simultaneously. Furthermore, when a class is being initialized, it can trigger the initialization of other classes (thus the class creation modules can be called recursively). This initialization protocol contemplates all the possible cases that may appear in the presence of multiple threads by storing internal initialization status values and requiring that the JVM acquire and release the `Monitor` class at some particular points during class creation. These synchronization points are necessary for properly storing references of those classes that were already initialized in the internal data structures. The use of `Monitor` is advantageous in the case of recursive class initialization, since `Monitor` supports recursive calls. For example, a variable initializer in a Java `Class` A might invoke a method of an unrelated `Class` B, which might in turn invoke a method of `Class` A.

- *Class Jump Tables.* Class jump tables are structures used for finding the appropriate class method (a process called method lookup) to execute in a Java object. They are necessary because of method overloading and class inheritance. Jump tables are constructed during class creation, and at that time they require exclusive access to internal hash tables, that are used for storing references to methods and interfaces accessible from a Java object.

- *Constant Pool Creation.* The constant pool table is a structure that provides executing JVM instructions references to classes, fields, methods, strings and constant values. The JVM Specification [LY99] allows the instructions not to rely on the runtime layout of classes, interfaces, class instances or arrays. The constant pool is created during the class parsing phase, when it may be simultaneously accessed by multiple threads. Locks ensure the consistency of the information stored there.

3.5 Multithreaded Quick Opcodes

The use of quick opcodes is a technique employed in the implementation of a JVM to improve the performance in interpreting some Java opcodes [LY96]. It consists of the replacement, at runtime and under certain conditions, of the standard opcodes with faster, more efficient, implementations.

Jupiter originally employed this technique to tackle single-threaded Java applications [Doy02], where it can be implemented by simply replacing the opcode and its parameters. This is safe because no other section of Jupiter or the Java program will try to simultaneously access an opcode that is being replaced. However, this procedure can lead to race conditions under multithreading, as described in Section 3.5.2. Thus, it became necessary to extend the quick opcode technique to support multithreading. A lock and new temporary opcodes are used to protect the opcode replacement procedures from these race conditions, without introducing significant overhead.

3.5.1 Quick Opcodes Overview

Some Java opcodes must perform several checks in their implementation, since they can be invoked under many different conditions. These conditions are tested every time. However, after the opcodes are executed once, some of these checks become redundant because the conditions they examine are guaranteed to remain unchanged. Thus, the checks can be avoided and the opcodes can be replaced with more efficient implementations, which are referred to as quick opcodes.

For example, when `putfield` is invoked to operate on a field for the first time, the field is checked to verify if it exists and is accessible. This is referred to as *field resolution*. Thus, if a reference to the field was not yet resolved, the resolution process is executed and the reference is made available for `putfield` to use. This requires the support from internal data structures. However, the check and resolution steps are necessary only the first time the opcode is encountered. Performing such operations on successive invocations of the same `putfield` opcode is redundant, since the resolution process would return the same result every time. Furthermore, traversing the data structures is time consuming, and can be avoided using a cache. Thus, after its first invocation, a specific `putfield` opcode is replaced with a quick opcode, `qputfield`, that does not require any resolution or checks and caches the results from the internal Jupiter data structures, thus saving execution time.

Bytecodes are maintained in the Java class. There is a bytecode array that contains a sequence of bytecodes for each method to which they belong in the class. Each bytecode consists of an opcode and its parameters. The opcode interpreter reads bytecodes, and for each opcode in a bytecode, it obtains its parameters. With this information, it can call the appropriate procedure that will service it. The JVM Specification [LY99] defines the number and size of the parameters that each opcode takes.

For example, the `putfield` opcode has the following format [LY99]:

putfield	parameter1 (index1)	parameter2 (index2)

It takes two parameters, *index1* and *index2*, which are byte values used to construct an index into the runtime constant pool of the current class. In turn, the item stored at that index location contains a symbolic reference to a field in the object where `putfield` was invoked. From there, an *offset* value must be obtained to access the field value.

Figure 3.8 depicts the mechanism of the `putfield` opcode execution. When the `putfield` opcode is read, a resolution table is used to point to the appropriate procedure that must be executed, in this case `putfield_impl`. The resolution table contains an entry for each standard Java opcode in addition to the quick opcodes defined in Jupiter. The `putfield_impl` procedure reads the opcode parameters, *index1* and *index2*, which are used to resolve the references into the Jupiter structures and obtain the *offset* value. Then, `putfield_impl` can rewrite `putfield` as `qputfield` and *index1* as the *offset* value (*index2* is not used). `qputfield` has the following format:

qputfield	parameter1 (offset)	parameter2 (index2)

The only action taken was the replacement of the opcode and its parameters, which was done in the `putfield_impl` function. The `putfield` implementation was

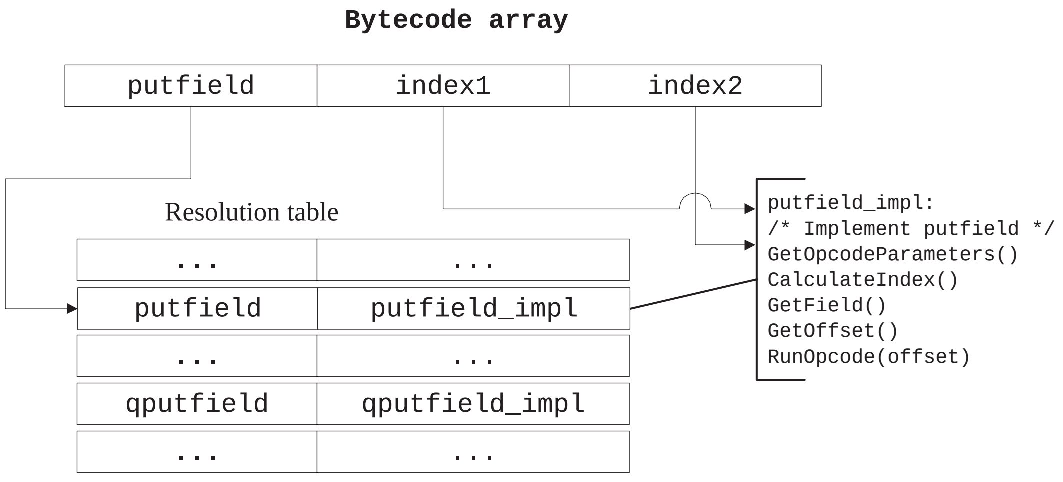


Figure 3.8: `putfield` Opcode Resolution

not invoked. Therefore, `qputfield`, which took the place of `putfield` and has all the information required to service the request, is forced to execute.

During quick opcode replacement no other bytecode is changed, only the occurrence containing `putfield` which was invoked in the bytecode array. When the method containing this bytecode array is executed the following time, the opcode interpreter will find a `qputfield` opcode, which will be serviced by the procedure `qputfield_impl`. This procedure will use the already computed *offset* value stored in the first parameter (ignoring the second parameter), as shown in Figure 3.9. Thus, after the replacement is complete, `qputfield` implements the semantics of `putfield`.

The most important aspect of quick opcodes relevant to the multithreading implementation is how the Java opcodes are actually rewritten. The opcode itself and its parameters must be replaced with the new quick opcode and its new parameters.

It must be noted that the quick opcode technique is not standard. Quick opcodes must be internally defined by each JVM; they are not part of the JVM specification or instruction set and are invisible outside of a JVM implementation. This implies that they do not use any support from external components, such as Java compilers. Nevertheless, they have been proven to be an effective optimization [LY99, Doy02] technique. And the lack restrictions from the specification gives JVM designers freedom to choose any

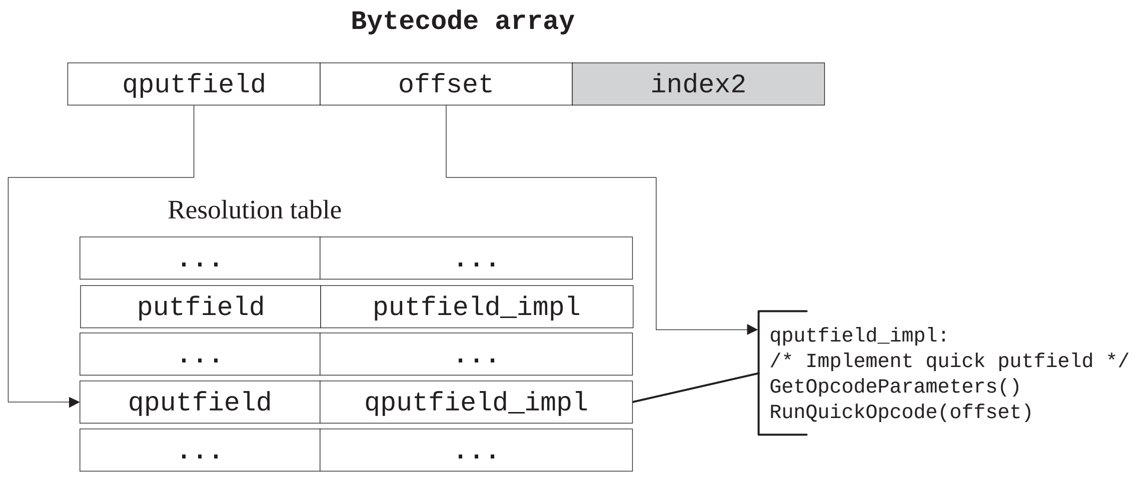


Figure 3.9: `qputfield` Opcode Resolution

particular implementation method that suit their needs.

3.5.2 Multithreaded Quick Opcode Replacement

Implementing quick opcode replacement in the presence of multiple threads involves the risk of race conditions, from which Jupiter must be protected. Immediately after the opcode is replaced with a quick opcode, the values of the parameters still remain as valid values for the original opcode. However, these are inconsistent with the implementation of the quick opcode. Another thread reading through the bytecode array may encounter this invalid data, thus causing an error. Therefore, the bytecodes (i.e., opcode and parameters) must be replaced atomically.

A naïve approach to avoid this race condition is to protect every bytecode, or the entire bytecode stream, with a lock. However, this requires that the lock be acquired every time a bytecode is interpreted, which is unacceptable.

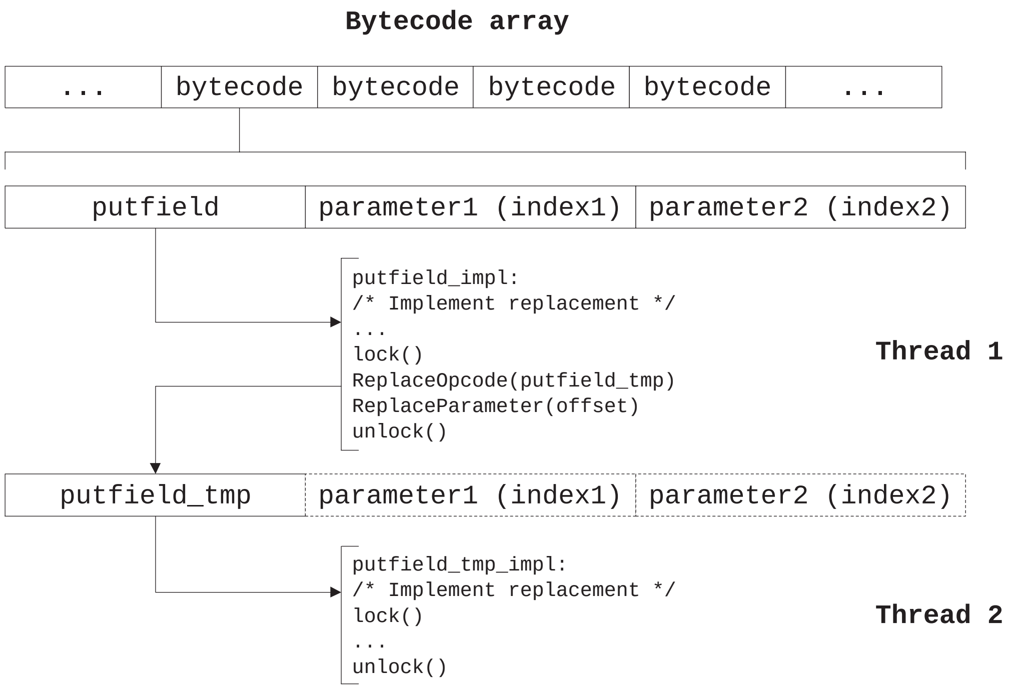
Instead, a more efficient solution is presented, which involves the use of a lock and a temporary opcode. In this case, the quick opcode change is made in two steps. When the original opcode is executed, it is replaced with a temporary opcode. Then, the temporary opcode is replaced with the quick opcode, which will provide the functionality required from the original opcode. Both steps are protected with locks. As in the single-threaded case, after each opcode is modified, its replacement is forced to execute in order to continue with the replacement process and, ultimately, to service the opcode that was originally intended to run. Subsequent calls will utilize the quick opcode without going through all these steps.

Each Java method contains a lock that is used to protect its entire bytecode array. This lock is not acquired when the bytecodes are read. Instead, it is only acquired when the bytecodes are replaced with their intermediate and quick versions. This allows threads to proceed to read the bytecodes and parameters and use the resolution table to determine the implementation function to execute. In the first invocation of the bytecode, the implementation function will perform the replacement.

3.5.2.1 Opcode Replacement

Threads are allowed to read the bytecode stream without locking. However, it is still necessary to prevent another thread from reading the opcode until the parameters have been updated. For this, a special lock and a temporary opcode are used. The special lock is acquired in the implementation function, before the opcode replacement takes place, and is released afterwards. This function then proceeds to replace the original opcode and parameters with the temporary opcode and the new parameters. The execution of this temporary opcode by other threads, while the first is still updating the opcode parameters, will cause its implementation to also attempt to acquire the same lock. However, these other threads will not be able to proceed until the replacement of both the opcode and its parameters is completed. When the temporary opocde implementation acquires the lock, it replaces itself with the quick opcode, thus ensuring that the parameters are valid for the quick opcode.

Figure 3.10 illustrates this process, which shows a scenario that exemplifies the way locks are used. While the `putfield` opcode is replaced by *Thread 1*, the lock is being held by the `putfield_impl` procedure. However, since the opcode interpreter must remain lock-free, it may cause a race condition because the interpreter in *Thread 2* can encounter inconsistent opcode and parameter values while the replacement takes place. In this case, the `putfield_tmp` opcode is found by the interpreter while the changes



```
putfield_impl:
/* Implement replacement */
...
lock()
ReplaceOpcode(putfield_tmp)
ReplaceParameter(offset)
unlock()
```

```
putfield_tmp_impl:
/* Implement replacement */
lock()
...
unlock()
```

Figure 3.10: Use of Locks for Multithreaded Opcode Replacement

still underway (the parameters were still not modified). For this opcode to be serviced correctly, the `putfield_tmp_impl` procedure must also acquire the lock, thus ensuring that if the opcode is accessible, so are its parameters. If the lock is not acquired in both `putfield_impl` and `putfield_tmp_impl` race conditions may occur.

It must be noted that if `putfield_tmp` remained as the quick opcode, its implementation would acquire the lock every time it executes. This would be detrimental to performance, which would contradict the purpose of using quick opcodes. Therefore, the second replacement stage is introduced. Once `putfield_tmp_impl` can acquire the lock, it is ensured to have valid parameter values. Therefore, it can safely replace the `putfield_tmp` opcode with the final quick opcode, `qputfield`, that will use these parameters but will not require the use of locks.

The complete mechanisms for opcode replacement are depicted in the examples shown in Figures 3.11 and 3.12. These examples also account for the case where the opcode cannot be replaced. The original opcode (`putfield`) is implemented by an assessment function (`putfield_impl`). This will determine if the replacement with a quick opcode can be made or not[3]. If it cannot, then the original opcode (`putfield`) is replaced with a conventional, slow opcode (`putfield_slow`). If it can be replaced, the opcode is changed for a temporary opcode (`putfield_tmp`), which in turn changes the opcode to the quick version (`qputfield`). These replacements are protected with locks, as shown in their respective functions (i.e., `putfield_impl` and `putfield_tmp_impl`). When the original opcode is replaced with the temporary opcode, the parameters are also changed accordingly.

Not that it is not possible to overcome this problem by the use of atomic memory reads and writes on SMPs. Java opcodes and their parameters are logically placed one after another as a bytestream. Thus, the opcode-parameter pair could be seen as a two-byte number that could be replaced atomically, taking advantage of the memory system of a sequentially consistent SMP, which guarantees word-size memory operations to be atomic. However, not all the opcodes take one or two parameters. Even if this was the case, it is possible that the opcode and its parameter do not reside in the same word. This may happen when the opcode is placed as the last byte in the word, so its parameter would be aligned as the first byte of the following word. No hardware architecture can guarantee the atomicity of the memory operations under these conditions.

This solution may give the impression that it is inefficient to use a lock for quick opcode replacement. However, it must be noted that subsequent executions of the code will not run the assessment or replacement functions. The interpreter will encounter either the quick opcode or the slow opcode. In the worst case, locks are used only twice by a thread during the replacement of an individual opcode (the assessment function and the temporary opcode implementation). Neither the quick opcode nor the slow opcode implementations use locks.

Allowing read access to the bytecode stream without locking has the advantage that avoids the use of a synchronization mechanism that will likely generate contention, thus saving execution time. Associating the lock with the bytecode array of each method allows opcode replacement to be protected at method granularity. This makes the quick

[3]It should be noted that if an opcode is to be replaced with a quick opcode, that such replacement will be made by the first thread that encounters the opcode.

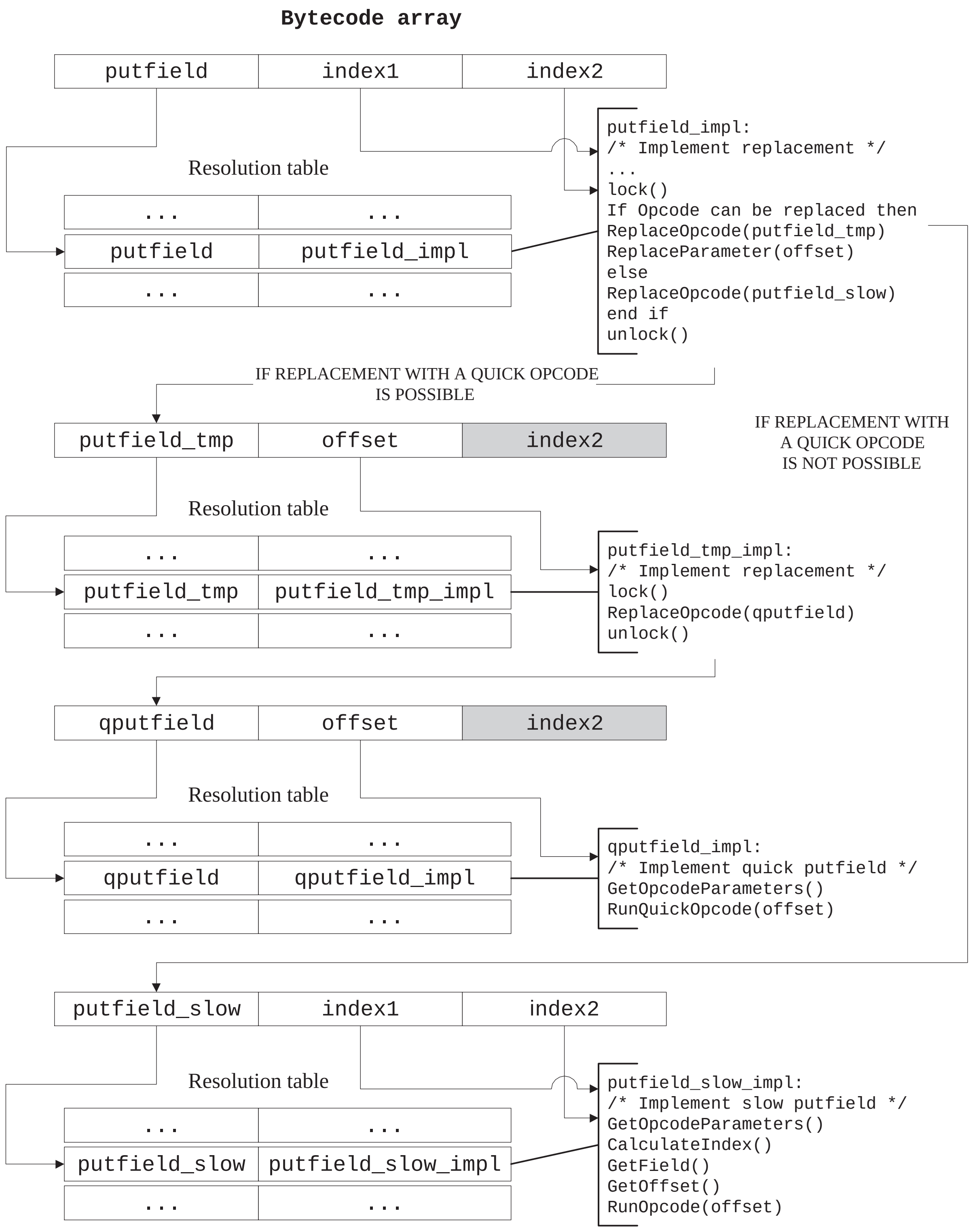


Figure 3.11: Multithreaded Opcode Replacement

```
procedure putfield_impl()
  Lock(QuickOpcodeMutex)
  Parameter ← GetOpcodeParameters()
  Index ← CalculateIndex(Parameter)
  Field ← GetField(Index)
  Offset ← GetOffset(Field)
  if Opcode has not changed then {Protect from threads waiting on
  QuickOpcodeMutex}
    if Opcode can be replaced then {Based on the value of offset}
      ReplaceOpcode(putfield_tmp) {Temporary Opcode}
      ReplaceParameter(Offset) {Parameters must be changed}
      NextOpcode ← putfield_tmp {Continue replacement}
    else
      ReplaceOpcode(putfield_slow) {Slow Opcode}
      NextOpcode ← putfield_slow {Continue execution}
    end if
  else
    NextOpcode ← GetOpcode() {Opcode has changed.  Execute it}
  end if
  Unlock(QuickOpcodeMutex)
  Execute(NextOpcode) {Continue execution}
procedure putfield_tmp_impl()
  Lock(QuickOpcodeMutex)
  ReplaceOpcode(qputfield) {Quick Opcode}
  Unlock(QuickOpcodeMutex)
  Execute(qputfield) {Continue execution}
procedure qputfield_impl()
  Offset ← GetOpcodeParameters()
  RunQuickOpcode(Offset)
procedure putfield_slow_impl()
  Parameter ← GetOpcodeParameters()
  Index ← CalculateIndex(Parameter)
  Field ← GetField(Index)
  Offset ← GetOffset(Field)
  RunOpcode(Offset)
```

Figure 3.12: Multithreaded Quick Opcodes

opcode technique flexible enough to be applied simultaneously at different methods, which also avoids unnecessary contention.

There is no significant degradation in performance for this procedure, which is confirmed by the comparative experimental results on SMPs presented in Chapter 5.

CHAPTER 4

Shared Virtual Memory Extensions

This chapter describes the extensions and architectural changes required for enabling Jupiter to run on the Myrinet cluster. It details the design challenges that arose in this enabling process and the techniques required to tackle them. Section 4.1 introduces the general design issues that had to be addressed. Section 4.2 explains the problems with memory consistency on the cluster and the solutions employed to solve them. Section 4.3 describes how Jupiter exploits private memory allocation. The limitations in the creation and use of some resources in CableS are exposed in Section 4.4. Finally, this chapter concludes with some details on the efficient use of barriers in Section 4.5 and the problems that arise with the use of a garbage collector on the cluster in Section 4.6.

4.1 Design Issues

There are some design issues that impact the enabling of Jupiter to run on the cluster. These issues can be classified in three categories:

- *Memory consistency.* This issue arises due to the use of the lazy release memory consistency model to provide a shared address space on the cluster, as opposed to the sequential consistency model used on SMPs[1]. This implies that, on the cluster, the contents of shared memory are guaranteed to be consistent only when

[1]It must be noted that since the Myrinet cluster nodes are SMPs, intra node memory management remains sequentially consistent. However, internode memory management follows the lazy release consistency model.

synchronization mechanisms (locks) are used to protect read and write memory operations. Consistency issues are dealt with in Section 4.2.

- *Private memory allocation.* Jupiter modules that can benefit from the use of private memory must be carefully identified, allowing for private memory to be exploited. Analogously, those structures that could potentially be used from more than one thread must be conservatively placed in shared memory. This issue is explored in detail in Section 4.3.

- *Limitation of resources.* The cluster system sets some limitations in the number of resources available to the processes executing on the cluster, such as the total shared memory, the number and type of locks that can be created and the locking operations that can be performed on them. Thus, it is critical to design the cluster-enabled Jupiter in such a way to minimize the use of such resources. These issues are described in detail in Section 4.4.

4.2 Memory Consistency

In this section, the process of cluster-enabling those Jupiter components for which the lazy release memory consistency model is an issue is described. This was the most important of the actions needed for enabling Jupiter to run on the cluster, because it is required to guarantee the correct execution of Java programs.

Unprotected access to shared memory in the SVM system is inherently unsafe, because there are no implicit memory consistency mechanisms between the cluster nodes. This requires that some of the multithreaded Jupiter components that relied on sequential consistency be adapted to the lazy release memory consistency model. Thus, it becomes necessary to use additional synchronization operations (locks) to guarantee that the latest values stored in shared memory are correctly updated between the cluster nodes.

The Jupiter components that are affected by memory consistency provide the following functionality: Java class creation (`ClassSource`), Java object creation and access (`Object` through `ObjectSource`), use of volatile fields (`Object` and `Field`), system thread creation (`Thread` and native implementation of the Java `Thread start` method) and Java thread creation and termination (native implementation of the Java `Thread start` method). A more detailed description of the affected components and how the memory consistency issue was addressed in each case is given in the following sections.

4.2.1 Locking on Java Class Creation

From the JVM point of view, the structures in a Java program that represent Java classes (for simplicity, these will be referred to merely as classes) need to be globally accessible. Classes contain the opcode array of the Java methods in the program, that must be accessible to all threads. However, it does not justify by itself the use of shared memory (since this is read-only data, it could be replicated locally on the nodes). Classes can contain `static` fields and methods that need to be visible from all its instances (objects), if there are any, and from other parts of the Java program [GJSB00]. Furthermore, the use of reflection allows for the Java class to be modified from the Java program at run time [SUN03]. These changes must be made visible to other threads. Therefore, classes are conservatively placed in shared memory, which implies they have to be protected with locks when they are created and accessed.

It must be noted that it was not possible to allocate classes in private memory and promote them to shared memory when requested by another thread. The need to share the lock used for class access requires that the thread that initially stores the class in private memory receive a notification from the thread that wants to access the class. The home thread must then acquire the lock that protects the class, copy the class to shared memory, and release the lock. Then, it must notify the user thread that it can safely access the class from shared memory, using the same lock. Unfortunately, the current implementation of CableS does not include Inter Process Communication (IPC) between threads in different nodes.

There are several Jupiter modules that are involved in the creation of Java classes. From a design perspective, Jupiter uses the notion of `Source` classes. These are resource managers with simple and uniform interfaces [Doy02]. In this design, `ClassSource` is the class responsible for orchestrating all the individual calls that constitute the components of a Java class. It returns a valid internal representation of a Java class. As this is a central point where the Java class is constructed, this was the key location where the accesses to shared memory could be protected during class creation.

Some locking operations were previously added to this module when extending Jupiter to support multithreading, as seen in Chapter 3. However, such synchronization points had to be placed in different locations, since for the cluster they have to protect more memory operations. Indeed, most of the code in this module deals with the initialization of structures. In the case of multithreaded Jupiter, described in Chapter 3, the class creation routines were protected from race conditions with a `Monitor`. It was safe and correct to keep this design in the case of the cluster, since the use of `Monitor` guarantees that the memory consistency routines will be invoked[2].

4.2.2 Locking on Java Object Access

The Java Memory Model states that locks act as the points in the Java program where the values in an object that were modified by a thread are read from or flushed to memory. The standard allows for locks not to be used when accessing fields. In this case, any change that threads make is not necessarily made visible (and thus, the modified memory contents are not kept consistent) to other threads. If a Java object requires that a value be properly shared among threads, it has to use these locks. This is achieved only through the use of the `synchronized` statement or wait-sets and notification routines, described in Sections 3.1.2 and 3.1.3 respectively. The order of read and store operations outside the locking statements are not guaranteed either. This behavior is compatible with the semantics of locks in CableS [Jam02a].

The Java Memory Model requires that only the contents of volatile fields be immediately updated in memory, as explained in Section 4.2.4.

Correct initial access by other threads is guaranteed by the mechanisms for memory consistency at Java thread creation, described in Section 4.2.6.

[2]`Monitor` maintains memory consistency in the same way locks do because the implementation of `Monitor` uses a lock for mutual exclusion. The cost of using a `Monitor` on the cluster is comparable to that of using a lock.

4.2.3 Locking on Java Object Creation

All Java objects created in Jupiter are, conservatively, allocated in shared memory. This is because any Java object is potentially shared among threads. This implies that, during creation, these objects must be protected with locks in order to make them shared. Although data updates or regular access to the fields or methods in these objects do not require explicit action to maintain memory consistency, locks are required for object creation in order to avoid memory corruption, as explained earlier. As in the case of Java class creation, described in Section 4.2.1, it was not possible to allocate objects in private memory and promote them to shared memory if required.

At the moment of this writing, Jupiter does not provide the analysis tools required to discern between objects that are private or shared among threads, such as escape analysis [CGS+99]. These tools can be used to reduce the calls to synchronization operations and the number of objects that must be allocated in shared memory.

4.2.4 Volatile Fields

The Java Language Specification [GJSB00] states that *volatile* fields must reconcile their contents in memory every time they are accessed. Furthermore, the operations on volatile fields must be performed in exactly the same order the threads request them.

In Jupiter, both `Field` and `Object` internal classes have an interface for accessing field values. Such accesses were modified to use locks. When a volatile field is accessed, a lock associated with it is acquired. After all the necessary operations are performed, the same lock is released. Figure 4.1 illustrates this process. In CableS this process is also the correct procedure to guarantee that every thread will always have access to the most recent value in memory. The use of locks results in a slowdown in the accesses to volatile fields, both for reading and writing values, but this is required for memory consistency and cannot be avoided.

It must be noted that in the SMP implementation it was not required to take any direct action on the use of single-word volatile fields, because these accesses are guaranteed to be atomic and need not be protected with locks. In this case, sequential consistency guarantees that the latest value stored in memory will eventually become

```
procedure FieldSetValue(Value)
  if field is volatile then
    Lock(FieldMutex)
    SetValue(Value)
    Unlock(FieldMutex)
  else
    SetValue(Value)
  end if
procedure FieldGetValue()
  if field is volatile then
    Lock(FieldMutex)
    Value ← GetValue()
    Unlock(FieldMutex)
  else
    Value ← GetValue()
  end if
```

Figure 4.1: Implementation of Volatiles

available for all threads to use.

4.2.5 Argument Passing at System Thread Creation

There are some Jupiter components that, instead of only using their own local lock[3] to maintain memory consistency, require the aid of a global lock to protect some of the data they handle. One of these components is the system thread creation routine.

When a system thread is created, it is common for the parent thread to pass some argument values to the child thread. This is used as a simple and efficient way of sharing important information, such as data structures and synchronization mechanisms. The POSIX pthread interface, as many others, allows for a single value, a pointer, to be passed to the child thread. This is enough information because a structure, large enough to hold all the necessary values, can be created in memory and passed through this reference.

As it was described earlier, there is no way for two threads to safely share a region of memory if there is not a lock that guards it when the values to that memory

[3]In this work locks are categorized as *local* or *global*, using the same taxonomy for variables in a program. Local locks are accessible from the scope of a function or class. The scope of global locks is the whole program.

```
procedure StartThread()
  SharedMutex ← CreateSharedMutex()
  Lock(SharedMutex)
  Arguments.Values ← CreateArguments()
  Arguments.SharedMutex ← SharedMutex
  Unlock(SharedMutex)
  CreateThread(ThreadFunction,Arguments)
procedure ThreadFunction(Arguments)
  SharedMutex ← ReadArguments(Arguments.SharedMutex)
  Lock(SharedMutex)
  Arguments ← ReadArguments(Arguments.Values)
  Unlock(SharedMutex)
  RunThread(Arguments) {Execute the thread using Arguments}
```

Figure 4.2: Problems with Sharing a Lock via Parameters

are being read and written. Therefore, the child thread needs access to the same lock as the parent, which will allow it to safely access all the parameters its parent is sending. A problem then arises in how to make the child and the parent to agree on a lock.

It is tempting to solve this problem by having the parent send a lock to its child as one of its arguments. This way, the parent can directly share it with the child, and both can use it to protect memory access. However, the point where the child thread needs to acquire this lock precedes the moment in which the arguments can be read safely. Figure 4.2 illustrates this problem. `StartThread`, which is invoked by the parent thread to start a child thread, uses the lock called `SharedMutex` to protect all the values that are set to the `Arguments` variable, which includes the lock itself. Then, `ThreadFunction`, which is the function executed by the child thread, reads the arguments to obtain the lock. However, the arguments cannot be safely read until the lock is acquired. In other words, the shared lock cannot be accessed before the arguments can be read, and the arguments cannot be safely read before the shared lock is accessed.

Thus, an alternative locking mechanism is required for a parent thread to send parameter values to its child thread. The mechanism employed makes both parent and child threads agree beforehand on the lock they are going to use to protect the memory holding the arguments. This makes the lock a system global lock, which both parties

```
procedure StartThread()
  Lock(GlobalMutex)
  Arguments.Values ← CreateArguments()
  Unlock(GlobalMutex)
  CreateThread(ThreadFunction,Arguments)
procedure ThreadFunction(Arguments)
  Lock(GlobalMutex)
  Arguments ← ReadArguments(Arguments.Values)
  Unlock(GlobalMutex)
  RunThread(Arguments) {Execute the thread using Arguments}
```

Figure 4.3: Need for Global Locks

can share and safely use at all times. The parent thread can protect the memory when writing the arguments, and the child thread can do so when it reads them. Figure 4.3 illustrates this mechanism. `GlobalMutex` is the global lock that both parent and child share. It is locked by the parent to write the arguments and released before the child is created. Then, it is locked by the child to safely read these arguments as the first step during its execution.

The implication of using a system global lock is that only one thread at a time in the system will be able to prepare and read the arguments sent to a thread. For example, if several threads are preparing to create one child thread each, only one of them will be setting the arguments to pass, and the others will be waiting for it to finish, even if there is no need to synchronize this activity between them.

Thread creation is a time consuming process in the current configuration of CableS, because in most cases it involves remote communication between nodes and node initialization. Node attachment, one of the steps required in node initialization, requires 3690 miliseconds on average to execute, and it is invoked every time a new node is being used to spawn a thread. This time increases as more nodes are dynamically added since more links between the nodes need to be established [Jam02b]. Thus, the time spent to serialize argument passing to the child threads becomes negligible.

Notice that, if the thread creation routines allowed for a second parameter, representing a lock, to be passed, it would be possible to avoid the global lock entirely. In this case, the child thread would receive two parameters: the structure containing all the arguments, and a reference to the shared lock in the form of a word. As no locking mechanism is required to share a single word value between threads (CableS passes these values automatically), the child thread could simply acquire the lock, and read the arguments safely.

4.2.6 Memory Consistency at Java Thread Creation

A Java thread can, potentially, access any class or object that its parent can. When a Java `Thread` object is created, the parent can initialize the fields in the child Java `Thread` using references to any of the fields, objects or classes the parent can reach. As a consequence, the child thread could contain references to elements that were modified or created by its parent. Thus, it must be ensured that all the values stored in its fields are in shared memory areas that can be safely accessed between nodes. If there is no mechanism to protect this memory, the child thread could access a memory segment containing invalid, uninitialized or inconsistent data, and the program execution may be corrupted or it may fail.

Figure 4.4 illustrates how these problems can be avoided. In `StartThread`, a lock that is shared between the parent and the child system threads, called `SharedMutex`, is created and acquired. Then, the memory that will be accessible from the child Java thread (which was not started yet) is marked as *modified* when `TouchMemory` is invoked. Such memory is up to date in the node where the parent Java thread executes. In `TouchMemory`, all the fields in the child Java `Thread` object (or the corresponding instance of a Java class that extends the Java `Thread` class) are traversed, and the objects, arrays, interfaces and the superclass that are accessible from the `Thread` object are identified. Any of these can give the child thread access to many other components, through their respective fields. Therefore, it is necessary to recursively inspect the object fields and mark the memory they reference; each object, array, interface and the superclass encountered are also recursively inspected and marked. When the call to `TouchMemory` returns, `SharedMutex` is released, the child system thread that will execute the Java child thread is created, and `SharedMutex` is passed as a parameter to it. In `ThreadFunction`, which is the function executed by the child system thread, `SharedMutex` is acquired. Then, the child thread starts its normal execution. Every time a memory area passed from the parent thread is accessed, the SVM system ensures that it is updated appropriately (because `SharedMutex`, the same lock used for marking memory, is being held). `SharedMutex` is only shared between the parent and child threads, exclusively for the purpose of ensuring memory consistency. Before terminating, the child thread releases `SharedMutex`. It must be noted that the memory mark mechanism is done in conjunction with the global lock technique used for passing parameters from a parent system thread to a child thread, which justifies the use of `GlobalMutex`, as shown in Section 4.2.5.

Notice that memory accesses within a single node are automatically kept updated by the memory system due to the sequential consistency model. The procedure explained above is only necessary when node boundaries are crossed and the thread is created in another node. But, from the application point of view, it is not possible to learn when this happens, because it does not have any information (nor should it) that may give any indication that the child thread is being created in a different node from the parent. Neither is it possible to anticipate this behavior because there are factors like the system load and the system parameters that can determine what cluster nodes are being used and in what order. Therefore, this procedure must be invoked every time a new thread is created, even when it is started in the same node.

The above mechanism for ensuring memory consistency at thread creation has the advantage of making only the memory segments actually accessed from the child thread to be replicated between the nodes. However, its disadvantage is the recursive traversal and marking of all accessible fields. The amount of memory that has to be marked depends on the number of fields the Java `Thread` object contains, and the objects with which they are initialized. However, the time spent marking memory at thread creation does not significantly impact performance because of the high overhead of thread creation in the current configuration of CableS [Jam02b].

```
procedure StartThread()
  SharedMutex ← CreateSharedMutex()
  Lock(GlobalMutex)
  Lock(SharedMutex)
  TouchMemory()
  Unlock(SharedMutex)
  Arguments.Values ← CreateArguments()
  Arguments.SharedMutex ← SharedMutex
  Unlock(GlobalMutex)
  CreateThread(ThreadFunction,Arguments)
procedure TouchMemory()
  MarkMemory() {Mark the piece of memory as modified}
  for each field reference in the constant pool do
    if object instance or array then
      TouchMemory(field)
    end if
  end for
  for each field in the object do
    if object instance or array then
      TouchMemory(field)
    end if
  end for
  for each interface do
    TouchMemory(interface)
  end for
  if it has a superclass then
    TouchMemory(superclass)
  end if
procedure ThreadFunction(Arguments)
  Lock(GlobalMutex)
  SharedMutex ← ReadArguments(Arguments.SharedMutex)
  Arguments ← ReadArguments(Arguments.Values)
  Unlock(GlobalMutex)
  Lock(SharedMutex)
  RunThread(Arguments) {Execute the thread using Arguments}
  Unlock(SharedMutex)
```

Figure 4.4: Memory Consistency at Java Thread Creation

4.2.7 Memory Consistency at Java Thread Termination

The Java Memory Model [LY99, Lea99] states that, when a Java thread terminates, it has to flush all its written variables to main memory. When the JVM runs on an SMP, there is no need to take direct action on this, since all data is updated properly by the hardware. However, on the cluster, where memory is not sequentially consistent, it is necessary to have a mechanism for forcing the memory system to flush the memory contents to the other system nodes.

This mechanism was implemented using a system call, `svm_flush_memory`[4], that allows updating memory on demand. This call is invoked before the system thread terminates and after all Java code has completed its execution in the terminating thread. There was another alternative for tackling this problem. CableS could automatically force a memory update to other nodes whenever a system thread terminates. However, this is not a general solution, from which all applications using CableS could actually benefit, because there may be applications which may not wish this to happen.

4.3 Private Memory Allocation

The current implementation of the SVM libraries makes access to non-shared memory faster than access to shared memory. This is because non-shared data can be allocated in the private memory of the processor that accesses it, and there is no need to invoke the SVM calls that ensure that this data is consistent across the system. Furthermore, it is natural to allocate non-shared data in private memory to reduce demands on shared memory. Thus, it is desirable to have non-shared memory allocated outside the SVM system, i.e., in private memory.

Jupiter's memory allocation infrastructure was extended with new modules that contemplate the possibility of allocating private as well as shared memory. Several Jupiter modules obtain a benefit from the use of private memory. They are: thread stack, JNI (Java Native Interface), thread handling routines, opcode interpretation and class parsing.

[4]This call was developed by the SVM group specifically for the purpose of this project.

The following sections explain how the Jupiter memory allocation modules were extended to support the allocation of private and shared memory, and describe how the different modules benefit from the use of private memory.

4.3.1 Private and Shared Memory Sources

Jupiter's philosophy encourages the use of Source classes to maintain its flexibility through the abstraction layers. Source classes are resource managers with simple and uniform interfaces [Doy02]. In this context, `MemorySource` is the module responsible for allocating memory. The original memory management module was replaced with an extended version that creates memory allocation objects that use POSIX `malloc` and CableS `svm_global_alloc` system calls for allocating private and shared memory respectively. There were two possible design options for this extension that minimize or avoid changes to Jupiter's interfaces:

- Create two distinct memory allocation objects, one for each memory location (private or shared).
- Provide more than one method for allocating memory in the existing memory allocation module.

Both options are equivalent with respect to functionality. It was decided to implement the first one, because it provides a clearer interface for the rest of the system and has some distinct advantages:

- Allocating shared memory does not involve calling a new method, but simply changing the memory allocation object that is passed to the Jupiter modules. This makes memory allocation calls more abstract than calling different methods, and allows for changes in memory allocation policies to take place at higher levels.
- Keeping the original interfaces of the memory allocation objects allowed to reuse most the existing code without modifying it.

- Having more than one memory allocation object required the extension of the interfaces of those modules that need access to private and shared memory simultaneously. This way, the need to obtain individual instances of these objects is explicitly shown in the new interface. Ultimately, this is beneficial to the users of the modules because the interface gives accurate information about the memory allocation requirements. Not having a clear interface would force the users to understand the implementation to find out for themselves these requirements.

However, creating two distinct memory allocation objects has one disadvantage. In part, this violates one of the initial design constraints, trying to avoid modifications to the original interfaces. Nevertheless, the new interface still keeps Jupiter flexible and extensible, but with a minor change.

Conversely, in the alternative solution some of the advantages become disadvantages, and vice versa. The advantage is that it does not require to change the existing interfaces in the system (with the exception of the memory allocation object). However, it has some disadvantages:

- It would have forced the interface to the memory allocation object to be extended, something that should remain as simple and small as possible.

- In order to maintain the modules compatible, it would have been necessary to extend the original memory allocation class to incorporate an interface for shared memory allocation. However, this would not provide real functionality when used on an SMP. But, since the call to this routine would still exist in some Jupiter modules, it would be misleading for trying to understand the source code. Furthermore, this design would imply that every memory allocation object created in the future would need to provide this interface, even when it would be impossible for it to provide real functionality for it.

- It would have been required to modify existing modules throughout Jupiter to add new calls to the memory allocation method. This would impact every place that shared memory was needed.

From a design point of view, the disadvantages of the second option outweigh its advantage. Moreover, such option does not conform with Jupiter's philosophy of well designed interfaces [Doy02]. Conversely, the advantages of the first solution are more important than its disadvantage. Thus, extending some classes's interfaces was the best approach to accommodate the requirements for the memory allocation modules.

It must be noted that it is not possible to use only private memory allocation unless there is no shared data between the application threads. In this respect, a JVM has to anticipate the needs of the programs it is going to execute and it has to conservatively decide where the memory for some components will be allocated. If there is a possibility that some memory area be used by more than one thread, then it has to reside in shared memory.

4.3.2 Thread Stack

Java programs are compiled to a set of opcodes. These are executed in the JVM, which is a stack-based interpreter. In this matter, Jupiter's responsibility is to create and maintain the stack for the Java program to execute. Stacks are, arguably, the most used elements in a JVM. Therefore, they have to be kept as efficient as it is possible [Doy02].

Section 3.5 of the JVM Specification [LY99] defines the use of stacks for Java threads:

> *"Each Java virtual machine thread has a private Java virtual machine stack, created at the same time as the thread. A Java virtual machine stack stores frames. A Java virtual machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return. Because the Java virtual machine stack is never manipulated directly except to push and pop frames, frames may be heap allocated."*

In Jupiter, threads are independent opcode interpreters, as it was described in Chapter 3. Each having its own stack. A thread's Java stack only stores the state of

Java non-native method invocations (Native method invocations are handled differently, as discussed in Section 4.3.3). Since the stack always remains private to a thread, and is never shared, it was possible to allocate it in private memory.

It must be noted that, at a certain moment in the execution of the Java program, some of the values stored in the stack will be references to objects. These objects will be allocated in shared memory, and it is probable that the most updated copy of those objects will reside in a remote node. However, this does not have any effect in keeping the stack in private memory.

4.3.3 JNI (Java Native Interface)

JNI is formally defined as a standard programming interface for writing Java native methods and embedding the JVM into native applications [SUN03]. It is commonly used to take advantage of native-code platform-specific functionality outside the JVM.

When the operating system creates a new thread, it provides it with a stack that the thread will use to store values, parameters, and return addresses during its execution. This stack is created by the operating system for the thread to use it exclusively, and is not shared. Thus, it is inherently private to the thread. Jupiter allows the call to native methods to use this same stack to receive its parameters. Consequently, calls to native methods in Jupiter can be handled private to the thread, with no synchronization operations.

There are other Jupiter modules that provide additional support for the implementation of native methods. All natives that are accessed from a Java thread are resolved and kept in a cache of method bodies, indexed by method name. The cache contains the resolved addresses of the native functions that were accessed by the Java program. When the native call is requested more than once, a cache hit prevents the need to resolve the addresses again. As the calls to native methods are private, the cache also remains private to the thread.

4.3.4 Thread Handling

In Jupiter, threads are managed in the same way as resources, they have a corresponding Source class, `ThreadSource`. In the current design, a single instance of `ThreadSource` is created for each thread in the system, which remains private to each of them. This is possible because the use of the various `ThreadSource` objects is decentralized, as was described in Section 3.2.

There are thread functions that can also exploit private memory allocation. For example, the JVM has to wait for all non-daemon threads to terminate before it can complete its execution [LY99]. In Jupiter, each thread only needs to be able to identify its children and wait for their completion. The supporting data structure (hash table) used for this purpose, is private to each thread. Therefore, it can be stored in private memory.

However, it was not possible to use private memory everywhere within the thread handling routines. In Jupiter, threads are internally represented by a structure that holds the necessary information to identify and manage a thread in the system, such as its identifier, priority and termination flags. This structure is also used as part of the internal implementation of the Java `Thread` object, where it is stored as a reference in a private field. From the Java program any Java thread in the system can potentially obtain references to other Java `Thread` objects. These references can be used to obtain relevant information from them or perform some operations, such as changing their priority or joining. Since `Thread` objects can be shared among threads, their fields must be stored in shared memory. Thus, the structure that represent threads must also be stored in shared memory. This structure exists regardless of the executing status of the thread (it can be created before a child thread has started executing and can even survive its termination).

It must be noted that if it was known for certain that in a particular Java program the `Thread` objects are not shared, these structures could be kept in private memory. However, this cannot be discerned without the use of compilation techniques such as escape analysis [CGS+99].

4.3.5 Opcode Interpretation

In the current configuration, each thread contains a separate opcode interpreter, or `ExecutionEngine`. It invokes the other components that make the opcode execution possible [Doy02]. This object has a single instance for each of the threads in the system, and that instance is completely independent of the others. That made it possible to allocate it in private memory.

4.3.6 Class Parsing

The procedures that are in charge of parsing the class can do their task privately, since they do not need to keep any permanent information to be shared among threads. Thus, they can be allocated in private memory. A Java class can be passed to the JVM for execution in several ways:

- As the *main* class, which initializes the execution of the Java program.
- As part of the class library (in this case ClassPath) initialization process.
- Referenced by the program or the class library during the execution of class or object methods.

In all these cases, the first step is to verify if the class was already resolved and if it was cached in memory. If it was not, then it must be parsed in order to interpret and discriminate its internal structure. Although the internal parsing structures can be made private, Jupiter's internal representation of the class must be stored in shared memory, as it was described in Section 4.2.1.

4.4 Limitation of Resources

CableS imposes some limitations on the use of some of the resources it provides. Basically, there is a strict limit on the number of locks and condition variables that can be created, and the number of times a lock can be acquired and released. Also, the total amount of shared memory is restricted.

Regrettably, these restrictions are more strict than those set by operating systems such as Linux, which do not place any explicit limitation on the creation of locks or condition variables (although there is always an implicit limit given by the amount of available memory and the address space of the structures used for identifying them). Furthermore, no operating system sets a limit on the number of times a lock can be acquired or released, nor should it.

CableS limits the total amount of available shared memory in the system to 256 Mbytes. This is regardless of the number of nodes the cluster is configured to use. This implies that, when the number of cluster nodes increases, the proportional amount of shared memory that each node can use decreases.

In the current configuration of CableS, up to 65536 locks and 50000 condition variables can be created. There are some limitations in the underlying layers of the system that force CableS to set these restrictions [Azi02]. Locks can be acquired and released approximately 100000 times (between successive barrier invocations). This restriction comes from the way CableS records the memory that the application modifies. Every time a page is changed, CableS keeps a record of the modified addresses. These changes are stored in an internal structure that is used to keep memory consistent among the cluster nodes. At approximately 100000 lock operations, it is no longer possible to store more modifications in that structure. Memory must be made consistent, and the structure is reset. This happens only at barrier invocations.

When a barrier is invoked, all the threads in the system have reached a common point in the execution of the program and CableS can safely refresh the memory modifications among all the system nodes. After that, it resets the internal structures, allowing for another 100000 acquire and release operations on locks. Then, the threads can resume their execution. CableS was designed to work with scientific applications, which make frequent use of barriers. Although an alternative solution to this problem exists [Jam02a], it is currently not implemented in CableS.

Regrettably, not all Java programs use barriers, and thus, it was not possible to rely on the SVM's mechanism for resource release. Furthermore, Jupiter has little use for barriers in a general sense. It is impractical to randomly add calls to barriers to overcome problems of this kind, or even use barriers as synchronization mechanisms instead of locks.

The approach that was taken to overcome these restriction was to carefully select the location where these resources were used. The number of locks and condition variables created had to be minimized, as well as the number of operations performed on them. The limit on acquiring and releasing locks proved a challenging problem to overcome.

However, there are some uses of monitors and condition variables that cannot be fully controlled. During startup, ClassPath [GNU03] creates several classes and objects. These serve as the basis for the initialization of the JVM and, ultimately, for the execution of the Java program. Both classes and objects must contain a mandatory monitor, as specified in the JVM Specification [LY99]. Jupiter's implementation of `Monitor` requires the use of a lock and a condition variable. Consequently, many of the locks and conditions variables used are determined by ClassPath initialization routines and the number of classes and objects created in the Java program.

Condition variables are also used to signal the finalization of threads, as described in Section 3.2.6. In this case there is only one for each thread. Locks are used for the synchronization of access to data structures and to keep memory consistent, as detailed in the previous sections of this chapter.

4.5 Barrier Implementation

The Java Grande benchmarks [EPC03, Jav03], used for performance and scalability evaluation in Chapter 5, use barriers as their primary synchronization mechanism. Initial experiments revealed that the implementation in the benchmark caused some problems. The SOR benchmark was tested with three different implementations of barriers:

- *Benchmark barriers.* These are implemented exclusively in Java and are part of the benchmark. The problem with this implementation is that it generates a very large amount of data replication among the nodes, even for a small number of threads running in few nodes. This implementation does not work well with the benchmarks on the cluster and makes the system slow. Furthermore, these barriers are not guaranteed to be correct under the Java Memory Model. See Section 5.1 for further details.

- *Original native barriers.* These are the original barriers provided by CableS. They were designed with the added purpose of resetting the internal SVM structures, which is a time consuming process, allowing for the use of more resources, such as locks [Jam02b, Jam02a]. However, the cluster extensions of Jupiter were designed to alleviate the limitation on the use of such resources. Thus, Jupiter does not need to rely on the extra functionality of barriers and can avoid the extra overhead they introduce. Furthermore, since these barriers are not used directly by Jupiter, but indirectly by the Java program, it is not possible to depend on them to alleviate the limitations in the use of locks, because it cannot be guaranteed that all Java programs indeed use barriers (e.g., not all the benchmark applications use barriers).

- *Lightweight native barriers.* These are a simplified version of the original CableS barriers. They do not perform any internal structure resetting when they are called. They only execute code that is responsible for the barrier functionality[5].

Simplifying the barrier is only part of the solution. Additionally, it is necessary to provide a way for Java programs to invoke it. In this case, it was desirable to also have an implementation that would be compatible and easy to test with other JVMs. This way, it would be possible to compare results without the need of further changes to Java programs. For this reason, a `Barrier` Java class was created as an interface to the native barrier implementation.

4.6 Garbage Collection

Both the single-threaded [Doy02] and SMP versions of Jupiter currently support version 6.1 of Boehm's conservative garbage collector[6] [Boe02b]. Unfortunately, it is not possible to use this garbage collector on the cluster [Boe02a]. Indeed, the use of a

[5]This simplification was developed by the SVM group specifically for the purpose of this project.

[6]The original Jupiter JVM needed a specially modified 6.0 version of Boehm's garbage collector. The features included in that version were incorporated into the 6.1 release.

distributed garbage collector [PS95] that is also compatible with the memory management routines of CableS is required. The design and implementation of such a collector is out of the scope of this project.

4.7 The Java Memory Model

For cluster-enabling Jupiter, supporting the Java Memory Model, it was necessary to rely on the lazy release memory model provided by the SVM system. Unfortunately, the Java Memory Model is not well understood and is even believed to be flawed [Pug99, Pug00]. However, the restrictions and relaxations stated in the JVM Specification [LY99], which are listed below, were addressed in the context of lazy release consistency, making us believe that we implement a more constrained model than what is required by the Java Memory Model.

- *Volatile fields.* These fields guarantee that the latest value stored in them is always visible to all threads. This is described in Section 4.2.4.
- *Thread creation and termination.* This involves several topics, such as thread handling and the memory consistency operations that take place at thread creation and termination. These are described in Sections 3.1.1, 3.2.6, 3.2.7, 4.2.6 and 4.2.7.
- *Object and class sharing.* Classes, and some objects, have to be visible from all threads. The interaction of these requirements with CableS are detailed in Sections 4.2.1, 4.2.2 and 4.2.3.
- *Synchronization.* The support for synchronization operations was already discussed in the context of the extensions for multithreading support, in Sections 3.1.2 and 3.1.3. Its implications with respect to the lazy release memory consistency model are part of the semantics of locks in CableS, as seen in Sections 2.5.2 and 4.2.

CHAPTER 5

Experimental Evaluation

The performance of the multithreaded Jupiter and the cluster-based Jupiter are evaluated using standard benchmarks. The results of this evaluation are presented in this chapter. Section 5.1 introduces the single-threaded and multithreaded benchmarks elected for the evaluation, and details some minor changes required in some benchmarks for their execution on the Myrinet cluster. Section 5.2 describes the experimental setup and methodology, and the platforms that were employed in our evaluation. Section 5.3 presents the results of the evaluation of the multithreaded Jupiter on a 4-processor SMP system. Finally, Section 5.4 presents the results of the evaluation on the Myrinet cluster.

5.1 The Benchmarks

For the purpose of the experimental evaluation of both the multithreaded and the cluster-enabled Jupiter, two different sets of standard benchmarks were used, SPECjvm98 [SPE03] and Java Grande [EPC03, Jav03]. Collectively, they provide a mix of single-threaded and multithreaded applications. Tables 5.1 and 5.2 list these two sets of benchmark applications. Most of the Java Grande benchmarks provide two data sets, small-size (A) and mid-size (B), and some include a large set (C). Appendix A provides a detailed description on the SPECjvm98 and Java Grande applications.

The Java Grande benchmarks could not be run unmodified on the cluster. It was necessary to perform some minor changes to some aspects of the benchmarks:

Application	**Description**
`201_compress`	Lempel-Ziv compression.
`202_jess`	Expert shell system.
`209_db`	Memory-resident database.
`213_javac`	JDK 1.0.2 Java compiler.
`222_mpegaudio`	ISO MPEG Layer-3 audio file decompression.
`205_raytrace`	Scene rendering by ray tracing.
`228_jack`	Commercial Java parser generator.

Table 5.1: SPECjvm98 Benchmark Applications

Application	**Description**
`Barrier`	Performance of barrier synchronization.
`ForkJoin`	Timing of thread creation and joining.
`Sync`	Performance of `synchronized` methods and `synchronized` blocks.
`Series`	Computation of Fourier coefficients of a function.
`LUFact`	Solving of a linear system using LU factorization.
`Crypt`	IDEA encryption and decryption on an array.
`SOR`	Successive over-relaxation on a grid.
`SparseMatmult`	Sparse matrix vector multiplications.
`MolDyn`	Particle interactions modeling in a cubic spatial volume.
`MonteCarlo`	Financial simulation using Monte Carlo techniques.
`RayTracer`	3D scene rendering by ray tracing.

Table 5.2: Java Grande Benchmark Applications

- *Barriers.* The barriers provided by the Java Grande benchmarks used an array for storing the barrier state of every thread, which wait on the barrier using busy wait with a back-off protocol. The array was declared as `volatile`, thus forcing the update of every modified element to memory using a lock, as described in Section 4.2.4. On the cluster, this implies constant memory replication among the nodes and constant use of locks, which is a limited resource, as described in Section 4.4. Furthermore, the Java Grande benchmarks do not guarantee the implementation of this barrier to be correct under the Java Memory Model [EPC03, Jav03]. For these reasons, the Java interface to the lightweight native barriers was used, which provided an efficient native implementation, as described in Section 4.5. The benchmarks affected by this change are: `LUFact`, `SOR`, `MolDyn` and `RayTracer`.

- *Random number generator.* The standard ClassPath version of the random number generator, which is used by the Java Grande benchmarks by default, has some `synchronized` methods for protecting the changes in the seed and caching results, `setSeed`, `next` and `nextGaussian`. This implies that the monitor, and therefore the lock, associated with the object is constantly being acquired and released. The use of locks is a limited resource on the cluster. Fortunately, the Java Grande benchmarks only generate random values before the parallel computation begins. As the use of locks is very expensive and this was a limitation that could not be avoided, the random generation routines were replaced with a compatible version with no `synchronized` methods. This modification did not affect the correctness of the generated results. The benchmarks affected by this change are: `Crypt`, `SOR`, `SparseMatMult` and `MonteCarlo`.

- *I/O operations.* CableS does not support sharing of internal operating system handles [Jam02a]. This includes those used for opening files for I/O operations. Therefore, all the I/O operations in `MonteCarlo` were replaced with a class that already contains the necessary values stored in it.

Processor	Intel® Xeon^{TM}
Clock Rate	1.40 GHz
Number of Processors	4
Cache Size (L2)	512 Kbytes
Physical Memory	2 Gbytes
Swap Memory	2 Gbytes
Compiler	`GNU gcc 2.95.3 20010315 (release)`
Operating System	Linux 7.3 i686 - kernel 2.4.18-4smp
ClassPath	Version `0.03`[1]
Notes	Hyperthreading was disabled from the BIOS. The compilation optimization level was `-O3`, with no debug information, and combining all the Jupiter source code into a single compilation unit [Doy02], thus facilitating function inlining optimization.

Table 5.3: SMP Experiment Platform

The Java Grande benchmarks can run unmodified under the multithreaded Jupiter on the SMP. However, in order to obtain comparable results for future use, these changes were also performed for the experiments on the SMP.

5.2 The Experiment Platforms

In the SMP and cluster experiments different hardware platforms were used. Table 5.3 shows the hardware and software platforms used for the experiments on SMPs. This was the largest Intel-based SMP available for these experiments. At the time of this work, Jupiter contains some assembly instructions (which are platform dependent), which did not allow to test other hardware platforms. Table 5.4 shows the hardware and software platforms used for the experiments on the Myrinet cluster. This is the largest cluster configuration that is tested and functional under the Linux version of CableS.

[1]The multithreaded version of Jupiter uses ClassPath `0.03`. In order to obtain comparable results, the single-threaded version of Jupiter was updated to support the same ClassPath version as the multithreaded version. The original Jupiter results [Doy02] use ClassPath `0.02`. ClassPath `0.03` uses fewer `synchronized` Java blocks and methods, and consequently fewer acquire and release operations on locks.

Processor	Intel® Pentium II^{TM}
Clock Rate	400 MHz
Number of Processors per Node	2
Cache Size (L2)	512 Kbytes
Physical Memory	512 Mbytes
Swap Memory	128 Mbytes
Nodes	8
Memory Model	Shared Virtual Memory (SVM) provided by CableS
Compiler	`GNU gcc 2.95.3 20010315 (release)`
Operating System	Linux 6.0/6.2 i686 - kernel 2.2.16-3smp
ClassPath	Version `0.03`
Notes	The kernel was modified to accommodate the VMMC modules. The compilation optimization level was `-O3`, with no debug information, and combining all the Jupiter source code into a single compilation unit [Doy02], thus facilitating function inlining optimization.

Table 5.4: Cluster Experiment Platform

5.2.1 Measurement Methodology

The execution times were obtained using the Unix `time` utility. Consequently, the values shown include JVM initialization. The value reported is the total (real) execution time. In the cluster experiments, the performance values reported by each specific benchmark are also given, which show the time spent in the parallel computation section, including thread creation time. This is important because thread creation is a slow process on the cluster [Jam02b]. These values are the standard measurements reported by the Java Grande benchmarks, and were not modified. It must be noted that some Java Grande benchmarks do not start their computation until all the threads are created, and thus report time that does not include initialization.

Unless otherwise stated, all times shown are measured in seconds. The average performance of each benchmark is calculated as the arithmetic mean of several independent runs. For the SMP experiments, 10 executions were used, and 5 for the cluster.

The average performance across the applications is presented using the geometric mean, for consistency with previously reported Jupiter performance [Doy02]. The geometric mean also has the advantage that ratios of the averages are more meaningful. For example, Kaffe/JDK ratio can be obtained as the product of Kaffe/Jupiter and Jupiter/JDK.

The overall speedup of all applications is computed as the ratio between the sum of the execution times of all the benchmarks at one thread and the sum of the execution times of all the benchmarks at a given number of threads. This gives more relative importance to those benchmarks that run for longer periods of time. On the cluster, the speedup of each benchmark is computed as the ratio between the execution time of such benchmark at one thread and the execution time at a given number of threads.

5.3 SMP Evaluation

Three sets of results are presented in this section, based on the SPECjvm98 and Java Grande benchmark suites. The first set compares the execution time of the single-threaded Jupiter [Doy02] and the multithreaded Jupiter using the single-threaded benchmarks from the SPECjvm98 set. The purpose of this set is to demonstrate that the inclusion of multithreading capabilities to Jupiter does not significantly degrade performance. Adding synchronization points, such as locks, increases the number of system calls, and also may create contention in the use of shared resources or structures, which can lead to performance degradation. It is not possible to do this comparison using the Java Grande benchmarks, because they are multithreaded and cannot be executed in the single-threaded version of Jupiter.

The second set of results evaluates the performance of the multithreaded Jupiter using the multithreaded Java Grande benchmarks. These results compare Jupiter's performance to that of Kaffe [Wil02] and Sun's JDK (interpreted mode only) [SUN03]. These results show that the multithreaded Jupiter speed is midway between a naïve JVM implementation and a complex, highly optimized JVM implementation. Furthermore, it shows that the execution time ratios between these JVMs are comparable to those obtained in the original work on Jupiter, thus extending that work [Doy02].

The final set of results presents the speedup obtained from the Java Grande benchmarks. This serves as a reference on the scalability of these benchmarks running

JVM	**Version**	**Parameters**
Sun's JDK [SUN03]	1.3.1	`-noverify -Xint -Xmx1000m -native`
Kaffe [Wil02]	1.0.6	`-noverify -mx 1000m`

Table 5.5: JVMs

Benchmark	**STJ**	**MTJ**	**MTJ/STJ**
201_compress	376.04	391.90	1.04:1
202_jess	116.54	135.47	1.16:1
209_db	178.93	191.68	1.07:1
213_javac	143.09	154.40	1.08:1
222_mpegaudio	314.71	301.69	0.96:1
228_jack	83.56	85.86	1.03:1
		Geometric Mean	**1.06:1**

Table 5.6: Execution Times for the Multithreaded Jupiter with Respect to the Single-Threaded Jupiter Using the SPECjvm98 Benchmarks

in Jupiter, Kaffe and Sun's JDK.

Table 5.5 details the version and execution parameters that were chosen for the reference JVMs. This selection of parameters is aimed at making a fair comparison with Jupiter. Verification and JIT compilation, which may have a significant impact it the execution times of a JVM, were disabled, since Jupiter did not possess any of such features at the time the tests were executed. For presentation purposes, Sun's JDK is referred to as JDK, the single-threaded version of Jupiter as STJ and the multithreaded Jupiter as MTJ. It must be noted that, for the following experiments, garbage collection was enabled for Jupiter and the reference JVMs.

5.3.1 Single-Threaded Jupiter vs. Multithreaded Jupiter

Table 5.6 compares the execution times obtained from the single-threaded[2] and the multithreaded versions of Jupiter running the SPECjvm98 single-threaded benchmarks.

[2]The single-threaded version of Jupiter did not implement synchronization [Doy02].

The average slowdown introduced is in the order of 6%, showing that Jupiter does not suffer from considerable performance degradation due to the addition of multithreading support. The reason for the slowdown is that, even when only single-threaded applications are executed, internal locking mechanisms necessary for multithreaded operations must be created. These include initializing, acquiring and releasing the appropriate locks. Also, class creation was upgraded from the single-threaded version of Jupiter to support the multithreaded class initialization protocol, as was detailed in Section 3.4, which adds extra synchronization points to this module.

The `202_jess` benchmark was slowed down by 16%. This is expected since this particular benchmark, despite being single-threaded, contains two `synchronized` methods, `gensym` and `putAtom`, which are called very frequently, resulting in degraded performance. However, it has the expected relative performance when it is compared with JDK and Kaffe, as it will be seen in Section 5.3.2.

At the same time, the speed of `222_mpegaudio` was improved by 4%. Unfortunately, the sources for this benchmark are unavailable, thus preventing further analysis.

5.3.2 Multithreaded Jupiter vs. Reference JVMs

Table 5.7 compares the execution times obtained from the multithreaded version of Jupiter and the reference JVMs running the SPECjvm98 single-threaded benchmarks. The mean execution time of Jupiter is 2.45 times faster than Kaffe and 2.45 times slower than Sun's JDK. Thus, the performance of the multithreaded-enabled Jupiter remains approximately the same relative to Kaffe and JDK as the single-threaded Jupiter [Doy02].

Table 5.8 compares the times obtained for the Java Grande multithreaded benchmarks, for the size *A* data sets. Notice that `205_raytrace` from the SPECjvm98 benchmarks was also included, since that application is also suitable for testing with multithreaded JVMs. In this case, Jupiter is between 1.77 and 3.00 times faster than Kaffe, and 2.01 and 2.18 times slower than Sun's JDK.

Table 5.9 compares the results obtained from the stress benchmarks in Section 1 of the Java Grande benchmark suite, which measure barrier, fork and join, and syn-

Benchmark	**JDK**	**MTJ**	**Kaffe**	**MTJ/JDK**	**Kaffe/MTJ**
201_compress	180.79	391.90	1217.02	2.17:1	3.11:1
202_jess	41.23	135.47	292.83	3.29:1	2.16:1
209_db	96.16	191.68	369.80	1.99:1	1.93:1
213_javac	51.19	154.40	354.90	3.02:1	2.30:1
222_mpegaudio	158.06	301.69	673.80	1.91:1	2.23:1
228_jack	32.59	85.86	276.14	2.63:1	3.22:1
Geometric Mean				**2.45:1**	**2.45:1**

Table 5.7: Execution Times for the Multithreaded Jupiter with Respect to the Reference JVMs Using the SPECjvm98 Benchmarks

chronization operations per second. Notice that, for these tests, obtaining larger values (or higher ratios) implies better performance. Jupiter still stays midway between Kaffe and Sun's JDK. However, Kaffe outperforms both Jupiter and Sun's JDK when running the stress tests on synchronization operations (`Sync`). These results depend on the implementation of the locking mechanisms in the respective JVMs.

Despite the efforts, the execution of `ForkJoin:Simple` under 4 initial threads runs out of memory in the system. Therefore, it is not possible to compute execution ratios that compare Jupiter's performance with that of JDK and Kaffe.

5.3.3 Speedup

The sum of the execution times of the Java Grande benchmarks is shown in Table 5.10, for the three JVMs. The speedup is shown in Table 5.11, which is computed as the ratio between the sum of the execution times at one thread and at a given number of threads, for each JVM.

The results show that Sun's JDK scales slightly better than Jupiter. In this case, the speedup obtained from Jupiter and Sun's JDK is close to linear. However, Kaffe does not scale well with these benchmarks and even shows a degradation in performance for 4 processors. Figure 5.1 plots these results.

1 Thread					
Benchmark	**JDK**	**MTJ**	**Kaffe**	**MTJ/JDK**	**Kaffe/MTJ**
205_raytrace	12.52	32.52	96.21	2.60:1	2.96:1
Series	97.85	72.06	141.61	0.74:1	1.97:1
LUFact	10.38	26.14	29.40	2.52:1	1.12:1
Crypt	11.81	22.25	41.98	1.88:1	1.89:1
SOR	67.38	150.39	126.81	2.23:1	0.84:1
SparseMatmult	30.50	52.37	116.15	1.72:1	2.22:1
MolDyn	119.27	361.90	612.03	3.03:1	1.69:1
MonteCarlo	69.25	134.91	241.78	1.95:1	1.79:1
RayTracer	224.39	574.60	1419.81	2.56:1	2.47:1
			Geometric Mean	**2.01:1**	**1.77:1**
2 Threads					
Benchmark	**JDK**	**MTJ**	**Kaffe**	**MTJ/JDK**	**Kaffe/MTJ**
205_raytrace	9.45	27.62	287.28	2.92:1	10.40:1
Series	48.96	43.04	71.91	0.88:1	1.67:1
LUFact	5.55	13.85	15.75	2.50:1	1.14:1
Crypt	6.51	12.11	23.73	1.86:1	1.96:1
SOR	34.61	77.27	66.50	2.23:1	0.86:1
SparseMatmult	15.06	26.60	60.46	1.77:1	2.27:1
MolDyn	64.28	182.82	372.42	2.84:1	2.04:1
MonteCarlo	36.41	72.67	128.86	2.00:1	1.77:1
RayTracer	116.49	290.00	949.30	2.49:1	3.27:1
			Geometric Mean	**2.06:1**	**2.13:1**
4 Threads					
Benchmark	**JDK**	**MTJ**	**Kaffe**	**MTJ/JDK**	**Kaffe/MTJ**
205_raytrace	8.07	29.35	1496.76	3.64:1	51.00:1
Series	27.33	22.89	36.16	0.84:1	1.58:1
LUFact	3.13	7.75	9.24	2.48:1	1.19:1
Crypt	3.98	7.07	14.86	1.78:1	2.10:1
SOR	18.28	40.71	36.64	2.23:1	0.90:1
SparseMatmult	8.11	14.74	33.98	1.82:1	2.31:1
MolDyn	32.92	94.89	198.91	2.88:1	2.10:1
MonteCarlo	19.96	40.88	96.29	2.05:1	2.36:1
RayTracer	58.49	199.62	1910.27	3.41:1	9.57:1
			Geometric Mean	**2.18:1**	**3.00:1**

Table 5.8: Execution Times for the Multithreaded Jupiter with Respect to the Reference JVMs Using the Java Grande Benchmarks (Size A)

1 Thread					
Benchmark	**JDK**	**MTJ**	**Kaffe**	**MTJ/JDK**	**Kaffe/MTJ**
Barrier:Simple	1157792.88	593708.31	190097.84	0.51:1	0.32:1
Barrier:Tournament	785766.19	463484.41	197210.28	0.59:1	0.43:1
ForkJoin:Simple	9285544.00	4275599.50	1673243.00	0.46:1	0.39:1
Sync:Method	52593.19	38569.08	122156.92	0.73:1	3.17:1
Sync:Object	46140.52	39630.60	135415.11	0.86:1	3.42:1
			Geometric Mean	**0.61:1**	**0.90:1**
2 Threads					
Benchmark	**JDK**	**MTJ**	**Kaffe**	**MTJ/JDK**	**Kaffe/MTJ**
Barrier:Simple	49233.07	59084.70	3062.46	1.20:1	0.05:1
Barrier:Tournament	484642.50	281715.28	93677.67	0.58:1	0.33:1
ForkJoin:Simple	149.59	1064.55	80.18	7.12:1	0.08:1
Sync:Method	24882.99	25168.76	51589.83	1.01:1	2.05:1
Sync:Object	25098.97	25161.12	72954.05	1.00:1	2.90:1
			Geometric Mean	**1.38:1**	**0.38:1**
4 Threads					
Benchmark	**JDK**	**MTJ**	**Kaffe**	**MTJ/JDK**	**Kaffe/MTJ**
Barrier:Simple	12715.25	11524.10	2503.22	0.91:1	0.22:1
Barrier:Tournament	260670.25	151095.59	64985.20	0.58:1	0.43:1
ForkJoin:Simple	733.19	No memory	32.58		
Sync:Method	13517.32	14383.81	14340.21	1.06:1	1.00:1
Sync:Object	13393.96	14575.08	17271.22	1.09:1	1.18:1
			Geometric Mean	**0.88:1**	**0.58:1**

Table 5.9: Stress Tests for the Multithreaded Jupiter with Respect to the Reference JVMs Using the Java Grande Benchmarks

5.4 Cluster Evaluation

This section gives the results obtained from the evaluation of the cluster-enabled Jupiter running the multithreaded Java Grande benchmarks on the Myrinet cluster. Since the SPECjvm98 benchmarks contain single-threaded applications (only one application, `205_raytrace`, is multithreaded) they were not used. The scalability study was performed using different node configurations, varying in the number of nodes in the system and the number of processors enabled in each node.

At this experimental stage, it is not possible to compare results with any other JVM, since only Jupiter is suitable for execution on the cluster.

Processors	**JDK**	**MTJ**	**Kaffe**
1	643.35	1427.14	2825.78
2	337.32	745.98	1976.21
4	180.27	457.90	3833.11

Table 5.10: Added Execution Times for the Multithreaded Jupiter and the Reference JVMs Using the Java Grande Benchmarks (Size A)

Processors	**JDK**	**MTJ**	**Kaffe**
1	1.00	1.00	1.00
2	1.91	1.91	1.43
4	3.57	3.12	0.74

Table 5.11: Average Speedup for the Multithreaded Jupiter and the Reference JVMs Using the Java Grande Benchmarks (Size A)

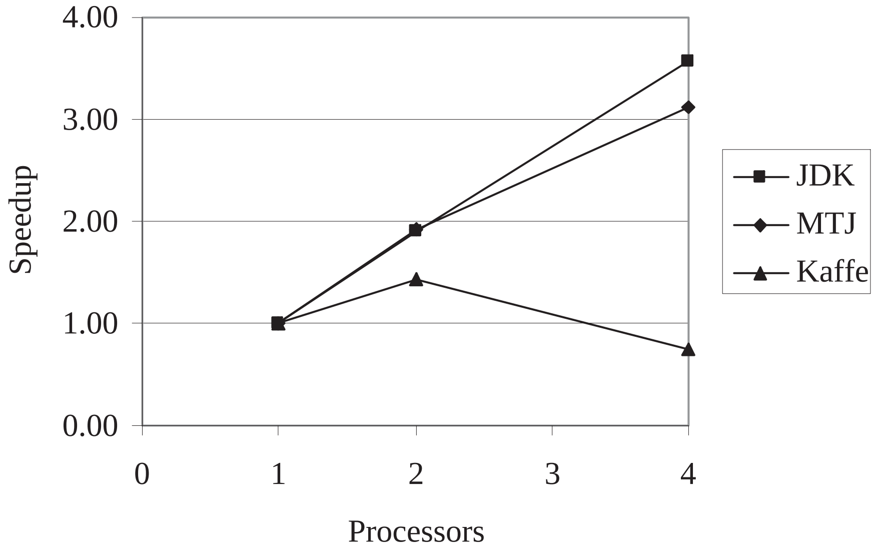


Figure 5.1: Average Speedup for the Multithreaded Jupiter and the Reference JVMs Using the Java Grande Benchmarks (Size A)

Three sets of results are presented. The first set displays the scalability results obtained with a cluster configuration where one processor per node is enabled for thread scheduling. The second set indicates the results with a cluster configuration where both node processors are enabled. These configurations refer only to the number of threads CableS creates in each node, and they do not limit the number of processors available to the operating system for scheduling its applications. The reason for making this distinction is twofold. On the one hand, CableS behaves differently in each case; on the other hand it simplifies the exposition of the results. The third set indicates the slowdown introduced to Jupiter through the use of the SVM system on the cluster.

Unlike the SMP case, described in Section 5.3, size A data sets do not provide enough information for these experiments. It was required that the benchmarks run for longer periods of time. Thus, results for size B data sets are also presented, since larger data sets show more accurately how the benchmarks scale because they amortize the cost of thread creation, node attachment and system initialization.

5.4.1 Evaluation Remarks

For the following experiments, the cluster-enabled Jupiter uses the lightweight barriers, described in Section 4.5. Garbage collection was disabled, as described in Section 4.6.

Even when the experiments are executed on one node, the SVM system is used. Since this system slows down memory access, this provides a fair reference for the speedup comparison with the subsequent cluster configurations. The overhead of the SVM system is explored in Section 5.4.4.

In the results, a distinction between execution time and benchmark time is made. The former shows the total time reported by the Unix `time` utility and it is the time experienced by the user, which includes SVM system initialization and termination. The latter is the time reported by the benchmark, which includes thread creation time. Both values are reported in order to give a more complete speedup reference.

A detailed analysis of the experiments revealed that some of the Java Grande benchmarks (`SOR` and `RayTracer`) violated the Java Memory Model, since they assumed sequential memory consistency [Bul03]. This memory model should not be assumed even for JVMs running on an SMP. In those cases, the necessary corrections were not performed since they would require a deep understanding and modification of the core procedures in the benchmarks.

Results for `Crypt` are not shown. This application has some problems in some configurations on the cluster, since it creates very short-lived threads. Due to CableS thread creation policy, these threads are not distributed evenly among all the cluster machines and they are concentrated on a few nodes. As a consequence, they exceed the current capacity of the system to execute them, since they consume all the available memory. This happens regardless of the data size that is used. It is also important to mention that, when running size A data set, the `Series` and `SparseMatmult` benchmarks do not always create threads in all the nodes. The cause is also due to CableS's thread creation policy. Nevertheless, this is normal system behavior and they still represent a fair test for those configurations. Changing the thread creation policy in CableS is out of scope for this work. Furthermore, the necessary changes would only be useful for those cases where threads are short-lived, which are not the main focus of this study.

When running size B data sets, the `Moldyn` and `MonteCarlo` benchmarks also reach some system limits with respect to the amount of available memory and the number of locking operations that can be performed. The limitation of resources in CableS, described in Section 4.4, does not allow to present results for the stress benchmarks either. Those applications stress CableS more than it can withstand in its current implementation stage.

5.4.2 One Processor per Node

Tables 5.12 and 5.13 compare the execution and benchmark times obtained for the size A and B sets of the Java Grande benchmarks respectively. These benchmarks were executed from 1 to 8 cluster nodes using one processor per node, for a total number of threads in the system ranging from 1 to 8.

The speedup of the benchmarks was computed using the sum of the execution (and benchmark) times of all the benchmarks shown in Tables 5.12 and 5.13. These results are shown in Tables 5.14 and 5.15, and Figures 5.2 and 5.3 depict these values for the respective individual benchmarks and overall results. The reference used for the speedup calculation is the cluster-enabled Jupiter, running one thread on one node.

The benchmarks show close-to-linear speedup in some cases for the size A and all cases in the size B sets. The `MolDyn` and `MonteCarlo` benchmarks do not scale well on the cluster. Overall, the speedup obtained for the size B set is better than that for the size A set.

The gap between the execution and benchmark times shows the impact of system initialization and termination. It must be noted that for the size B set, and in particular for `Series` and `LUFact`, this gap is reduced, and thus the impact of such operations.

5.4.3 Two Processors per Node

Tables 5.16 and 5.17 compare the execution and benchmark times obtained for the size A and B of the Java Grande benchmarks respectively. The benchmarks were executed from 1 to 8 cluster nodes using tow processor per node, for a total number of threads in the system ranging from 2 to 16.

The speedup of the benchmarks was computed using the sum of the execution (and benchmark) times of all the benchmarks shown in Tables 5.16 and 5.17. These results are shown in Tables 5.18 and 5.19, and Figures 5.4 and 5.5 depict these values for the respective individual benchmarks and overall results. The reference used for the speedup calculation is the cluster-enabled Jupiter, running one thread on one node, shown in Tables 5.12 and 5.13.

The benchmarks show a sublinear speedup with all size A and close-to-linear speedup with some size B sets. As in the case with one processor per node, the overall speedup obtained for the size B set is better than that for the size A set.

It is interesting to note that, for the same number of running threads, the execution and benchmark times obtained from the single-processor and dual-processor configurations are similar. This can be seen when comparing the values in Tables 5.12 and 5.16, and Tables 5.13 and 5.17. This confirms that, up to 8 threads, the applications scale in a similar way in both configurations. This could indicate the low impact of communication in the applications, or that the benchmarks follow a communication pattern that is handled very efficiently by the SVM system.

5.4.4 Performance Impact of the SVM System

The third and final set of results measures the impact of the SVM system in the execution of Jupiter. This is because the SVM adds a significant slowdown to the execution of applications. Thus, the SMP and cluster configurations were compared using the same cluster node. The execution times were obtained from running the size *A* data sets of the Java Grande benchmarks, on one and two threads. The results obtained are shown in Tables 5.20 and 5.21 for the execution times and times reported by the benchmark respectively.

As shown, CableS slows down Jupiter by 39% when running one thread, and by 44% when running two threads. The `Series` benchmark experiences some speed improvements. Though it was not possible to confirm the source of the speedup, it is likely that some pthread calls are faster in CableS (when running on a single node) than in the operating system, because their execution may not involve system calls. This, in conjunction with a low use of shared memory (where most of CableS controls take place), may cause the application to run faster. Unfortunately, the work on CableS [Jam02b] did not publish these results.

1 Thread		
Benchmark	**Execution Time**	**Benchmark Time**
Series	147.75	137.93
LUFact	111.40	97.01
SOR	494.85	469.09
SparseMatmult	182.02	158.87
MolDyn	1583.76	1573.61
MonteCarlo	460.67	448.77
RayTracer	2319.42	2308.33
Total	**5299.87**	**5193.61**
2 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	104.21	73.63
LUFact	92.31	56.46
SOR	294.97	249.54
SparseMatmult	127.02	82.80
MolDyn	880.69	850.47
MonteCarlo	363.11	331.68
RayTracer	1212.91	1176.75
Total	**3075.22**	**2821.33**
4 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	85.48	40.66
LUFact	83.77	31.62
SOR	193.57	129.03
SparseMatmult	101.73	43.45
MolDyn	528.35	485.59
MonteCarlo	265.71	214.09
RayTracer	644.23	600.30
Total	**1902.84**	**1544.74**
8 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	84.77	24.01
LUFact	89.46	22.98
SOR	150.25	72.98
SparseMatmult	105.61	26.78
MolDyn	489.98	423.67
MonteCarlo	291.36	232.86
RayTracer	380.83	320.10
Total	**1592.26**	**1123.38**

Table 5.12: Execution and Benchmark Times for the Cluster-Enabled Jupiter on One Processor per Node Using the Java Grande Benchmarks (Size A)

1 Thread		
Benchmark	**Execution Time**	**Benchmark Time**
Series	1388.30	1378.45
LUFact	795.59	768.16
SOR	1094.15	1050.84
SparseMatmult	360.93	324.65
RayTracer	25705.96	25692.78
Total	**29344.93**	**29214.88**
2 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	749.83	718.82
LUFact	463.54	414.81
SOR	654.78	588.25
SparseMatmult	224.86	165.88
RayTracer	13078.86	13046.50
Total	**15171.87**	**14934.26**
4 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	419.65	379.66
LUFact	292.74	226.41
SOR	384.25	305.80
SparseMatmult	160.17	84.23
RayTracer	6609.27	6563.77
Total	**7866.08**	**7559.87**
8 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	244.03	185.87
LUFact	201.76	130.80
SOR	246.84	156.94
SparseMatmult	139.73	48.13
RayTracer	3367.82	3314.06
Total	**4200.18**	**3835.80**

Table 5.13: Execution and Benchmark Times for the Cluster-Enabled Jupiter on One Processor per Node Using the Java Grande Benchmarks (Size B)

Processors	**Execution Time**	**Benchmark Time**
1	1.00	1.00
2	1.72	1.84
4	2.79	3.36
8	3.33	4.62

Table 5.14: Average Speedup for the Cluster-Enabled Jupiter on One Processor per Node Using the Java Grande Benchmarks (Size A)

Processors	**Execution Time**	**Benchmark Time**
1	1.00	1.00
2	1.93	1.96
4	3.73	3.86
8	6.99	7.62

Table 5.15: Average Speedup for the Cluster-Enabled Jupiter on One Processor per Node Using the Java Grande Benchmarks (Size B)

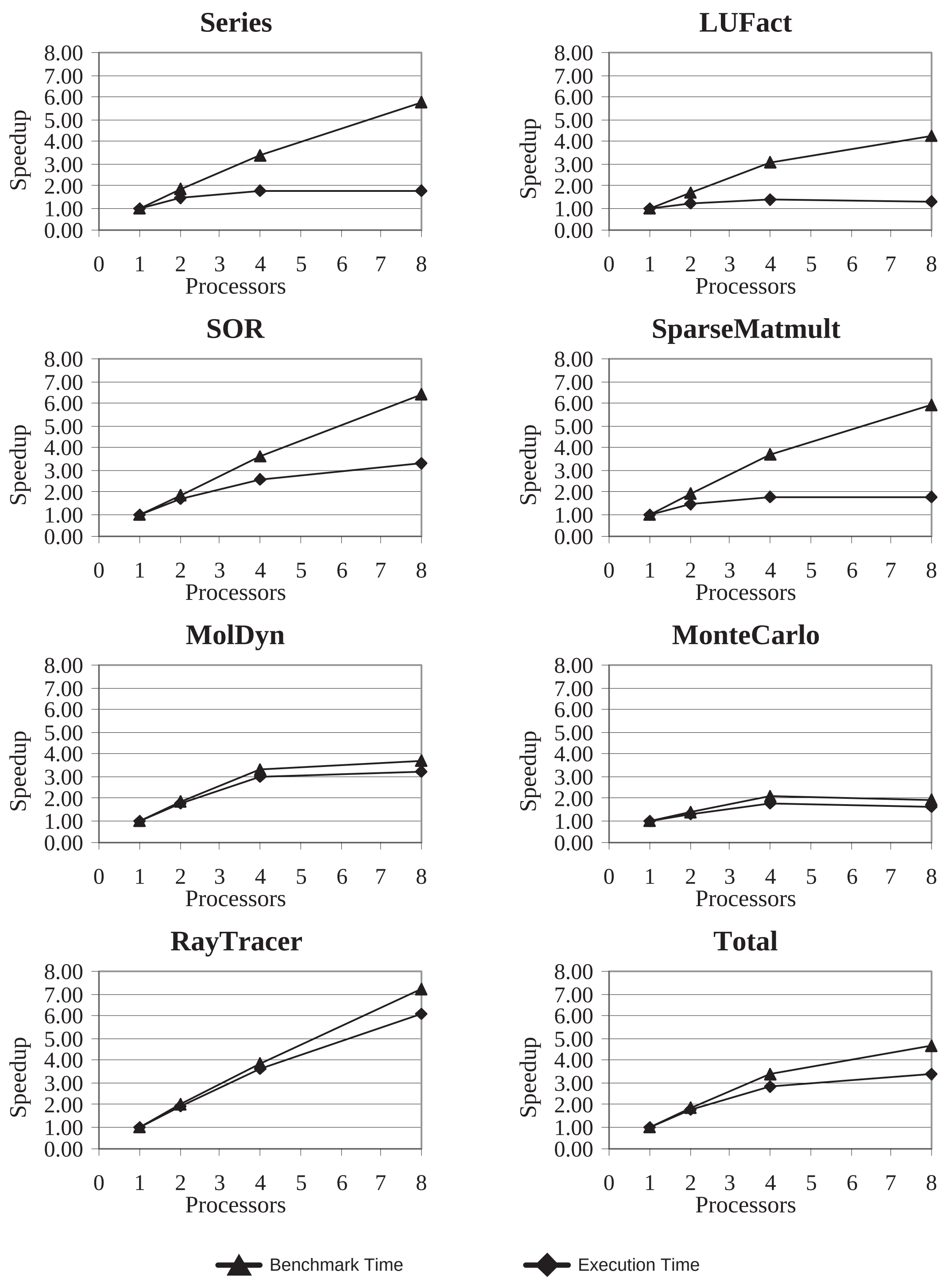


Figure 5.2: Speedup for the Cluster-Enabled Jupiter on One Processor per Node Using the Java Grande Benchmarks (Size A)

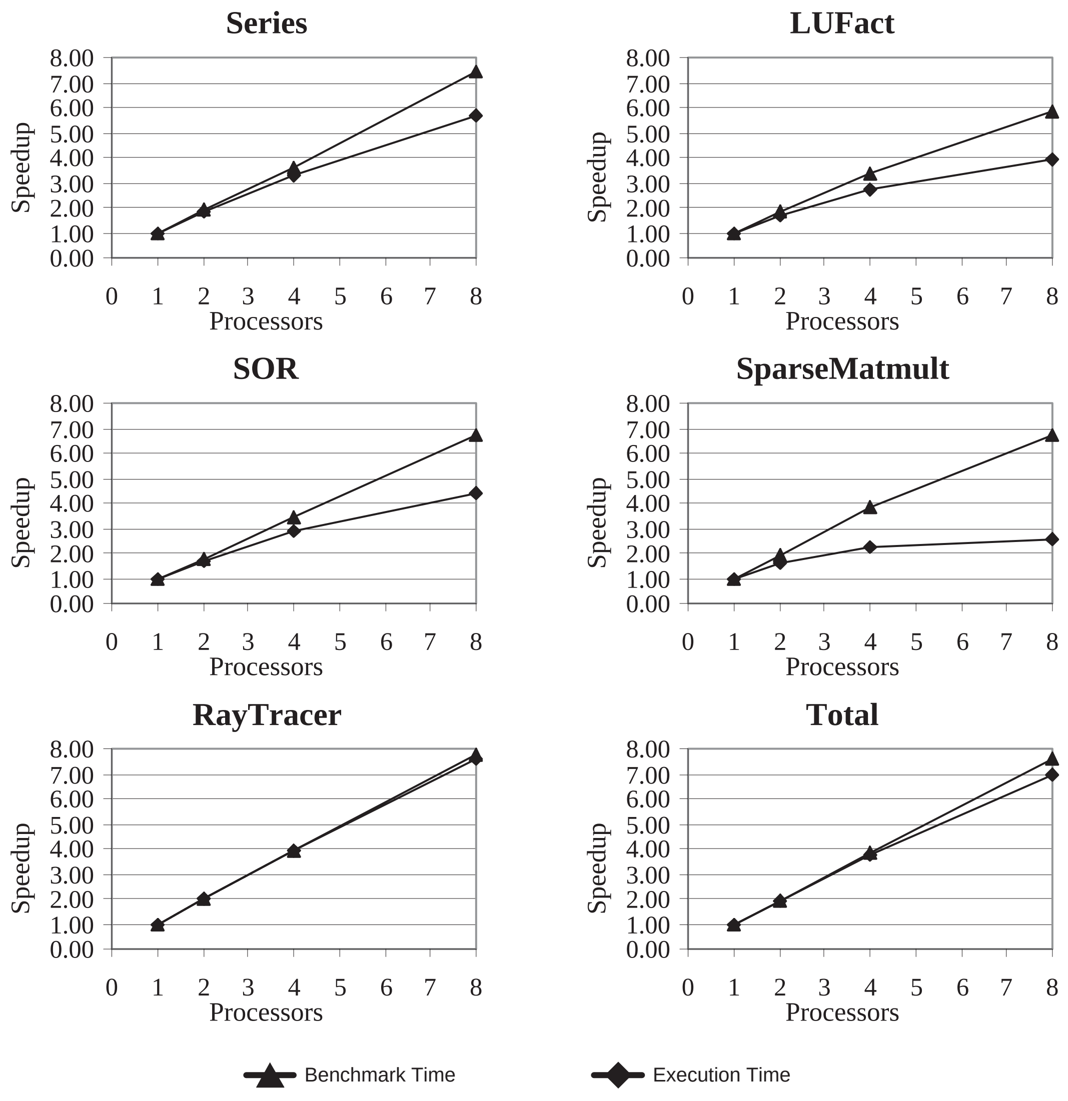


Figure 5.3: Speedup for the Cluster-Enabled Jupiter on One Processor per Node Using the Java Grande Benchmarks (Size B)

2 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	81.67	71.35
LUFact	65.83	50.82
SOR	266.37	239.78
SparseMatmult	105.96	81.08
MolDyn	827.16	815.74
MonteCarlo	263.87	251.10
RayTracer	1186.33	1175.64
Total	**2797.19**	**2685.51**
4 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	70.54	40.16
LUFact	70.73	34.69
SOR	173.93	128.11
SparseMatmult	88.68	43.48
MolDyn	507.57	477.91
MonteCarlo	275.55	244.02
RayTracer	641.13	611.17
Total	**1828.13**	**1579.54**
8 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	66.20	23.50
LUFact	75.18	22.19
SOR	136.27	72.00
SparseMatmult	92.63	26.22
MolDyn	425.66	375.90
MonteCarlo	311.05	259.95
RayTracer	370.94	322.01
Total	**1477.93**	**1101.77**
16 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	74.83	20.19
LUFact	86.91	23.50
SOR	170.37	49.12
SparseMatmult	111.38	25.56
MolDyn	692.41	632.00
MonteCarlo	317.12	253.35
RayTracer	264.33	207.32
Total	**1717.35**	**1211.04**

Table 5.16: Execution and Benchmark Times for the Cluster-Enabled Jupiter on Two Processors per Node Using the Java Grande Benchmarks (Size A)

2 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	715.22	704.39
LUFact	425.52	397.34
SOR	581.18	536.72
SparseMatmult	204.36	165.77
RayTracer	13057.39	13046.42
Total	**14983.67**	**14850.64**
4 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	392.69	362.48
LUFact	265.88	214.73
SOR	360.03	293.87
SparseMatmult	146.61	84.95
RayTracer	6596.79	6566.13
Total	**7762.00**	**7522.16**
8 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	246.79	190.67
LUFact	188.94	124.84
SOR	239.96	159.20
SparseMatmult	127.11	49.32
RayTracer	3357.52	3311.92
Total	**4160.32**	**3835.95**
16 Threads		
Benchmark	**Execution Time**	**Benchmark Time**
Series	153.83	101.29
LUFact	154.99	81.62
SOR	186.11	95.76
SparseMatmult	145.61	36.86
RayTracer	1781.02	1722.70
Total	**2421.56**	**2038.23**

Table 5.17: Execution and Benchmark Times for the Cluster-Enabled Jupiter on Two Processors per Node Using the Java Grande Benchmarks (Size B)

Processors	**Execution Time**	**Benchmark Time**
1	1.00	1.00
2	1.89	1.93
4	2.90	3.29
8	3.59	4.71
16	3.09	4.29

Table 5.18: Average Speedup for the Cluster-Enabled Jupiter on Two Processors per Node Using the Java Grande Benchmarks (Size A)

Processors	**Execution Time**	**Benchmark Time**
1	1.00	1.00
2	1.96	1.97
4	3.78	3.88
8	7.05	7.62
16	12.12	14.33

Table 5.19: Average Speedup for the Cluster-Enabled Jupiter on Two Processors per Node Using the Java Grande Benchmarks (Size B)

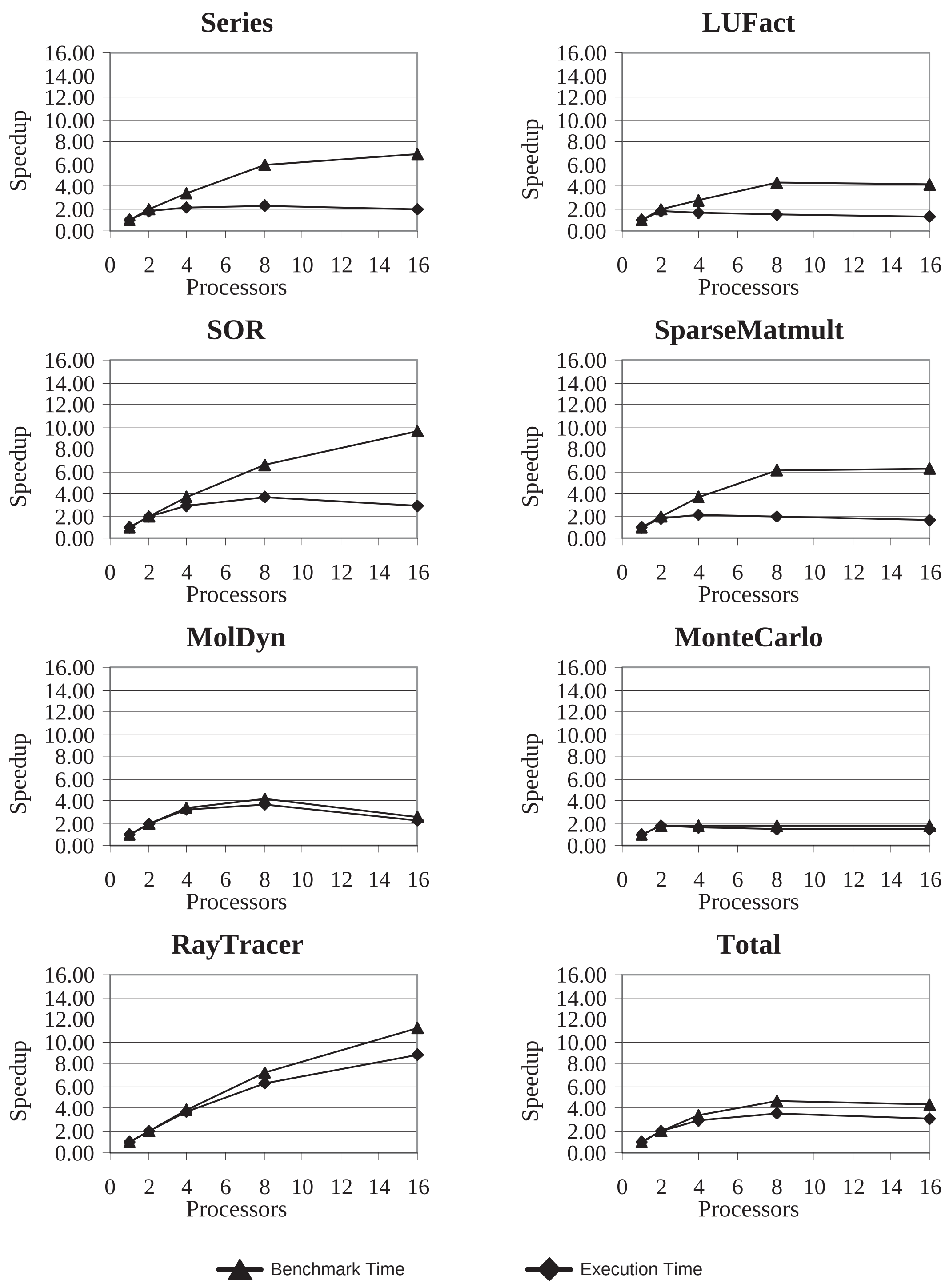


Figure 5.4: Speedup for the Cluster-Enabled Jupiter on Two Processors per Node Using the Java Grande Benchmarks (Size A)

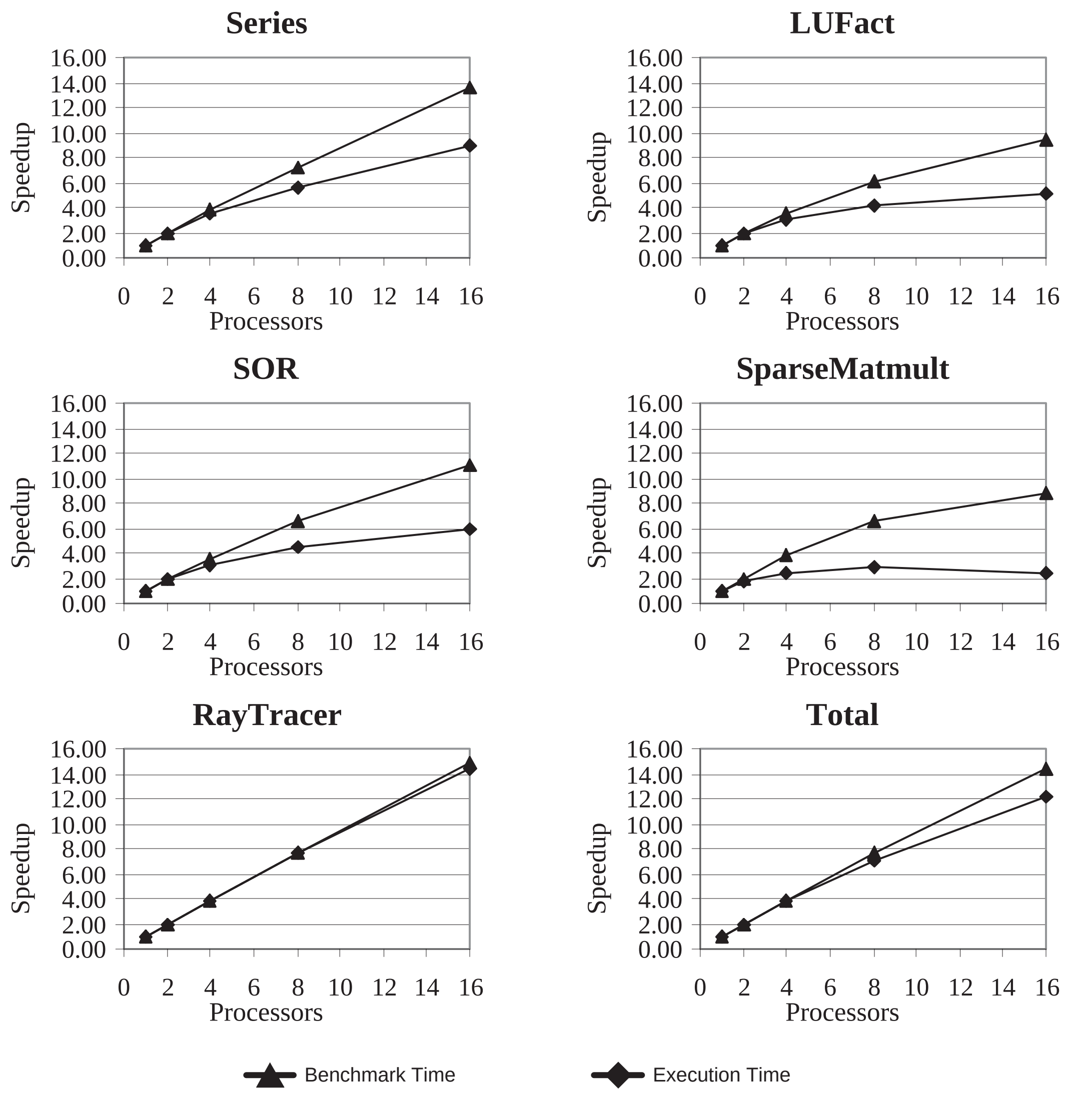


Figure 5.5: Speedup for the Cluster-Enabled Jupiter on Two Processors per Node Using the Java Grande Benchmarks (Size B)

1 Thread			
Benchmark	**MTJ**	**Jupiter-SVM**	**Jupiter-SVM/MTJ**
Series	153.18	147.75	0.96:1
LUFact	69.16	111.40	1.61:1
SOR	342.04	494.85	1.45:1
SparseMatmult	131.20	182.02	1.39:1
MolDyn	969.68	1583.76	1.63:1
MonteCarlo	266.75	460.67	1.73:1
RayTracer	1523.38	2319.42	1.52:1
		Geometric Mean	**1.45:1**
2 Threads			
Benchmark	**MTJ**	**Jupiter-SVM**	**Jupiter-SVM/MTJ**
Series	77.40	81.67	1.06:1
LUFact	36.86	65.83	1.79:1
SOR	177.11	266.37	1.50:1
SparseMatmult	70.07	105.96	1.51:1
MolDyn	490.32	827.16	1.69:1
MonteCarlo	134.38	263.87	1.96:1
RayTracer	763.96	1186.33	1.55:1
		Geometric Mean	**1.56:1**

Table 5.20: Execution Time Overhead Introduced by the SVM System

1 Thread			
Benchmark	**MTJ**	**Jupiter-SVM**	**Jupiter-SVM/MTJ**
Series	152.08	137.93	0.91:1
LUFact	65.79	97.01	1.47:1
SOR	332.82	469.09	1.41:1
SparseMatmult	123.05	158.87	1.29:1
MolDyn	968.57	1573.61	1.62:1
MonteCarlo	265.64	448.77	1.69:1
RayTracer	1522.23	2308.33	1.52:1
		Geometric Mean	**1.39:1**
2 Threads			
Benchmark	**MTJ**	**Jupiter-SVM**	**Jupiter-SVM/MTJ**
Series	76.30	71.35	0.94:1
LUFact	33.50	50.82	1.52:1
SOR	167.94	239.78	1.43:1
SparseMatmult	61.39	81.08	1.32:1
MolDyn	489.22	815.74	1.67:1
MonteCarlo	133.26	251.10	1.88:1
RayTracer	762.82	1175.64	1.54:1
		Geometric Mean	**1.44:1**

Table 5.21: Benchmark Time Overhead Introduced by the SVM System

CHAPTER 6

Conclusions

6.1 Summary and Conclusions

This thesis presented the design and implementation of a JVM on a cluster of workstations. For this purpose, two sets of extensions to the single-threaded Jupiter were implemented. First, Jupiter was enabled to execute multithreaded Java applications. Then, the multithreaded Jupiter was extended to run on a cluster under a lazy release memory consistency model.

For the multithreading extensions, it was necessary to develop the infrastructure to support thread handling, while following the design guidelines of abstract interfaces defined in the original Jupiter project. This involved the development of the support for thread behavior and synchronization specified in the JVM Specification [LY99], synchronization of those Jupiter structures that become shared in a multithreading configuration, and extensions to the quick opcode optimization to support concurrency.

For the cluster, it was necessary to deal with the lazy release memory consistency model. In most cases, this implied maintaining memory consistency using locks. Moreover, private memory allocation was used to store components that are not shared among threads. Finally, it was required to handle the limitations in the use of some resources.

The performance of both the multithreaded and cluster-enabled Jupiter was evaluated using two standard benchmark suites: the SPECjvm98 [SPE03] and the Java Grande [EPC03, Jav03]. The evaluation indicated that the performance of the multithreaded version of Jupiter is comparable to that of the single-threaded version, and that the addition of multithreaded functionality did not cause a significant overhead (6% on average). Furthermore, the speed of the multithreaded Jupiter interpreter is midway between that of Kaffe [Wil02], a naïve implementation of a Java interpreter, and that of Sun's JDK [SUN03], a highly optimized state-of-the-art interpreter.

The cluster evaluation showed the scaling of the cluster-enabled Jupiter. In this case, the system was tested on up to 16 processors. Overall, most of the benchmarks show scaling performance. Moreover, the speedup of the Size A Java Grande benchmarks on the cluster is comparable to that of the SMP for up to 4 processors. For example, the average speedup on the SMP is 3.12, while on the cluster it is 3.36. Furthermore, the impact of the SVM system was measured, which shows that the performance degradation introduced by the use of SVM is between 39% and 44%.

This work showed that it is possible and feasible to construct a scalable JVM to run on a cluster of workstations under an SVM model. There is also a number of lessons learned from the experience of extending Jupiter with both, multithreading and cluster support. We briefly touch on these lessons:

- This work affirmed the extensibility and modularity properties of Jupiter. Although this cannot be quantified, it was inferred from the development of this project. Jupiter's properties played a decisive role when enabling it to run on the cluster. The well designed Jupiter abstraction layers allowed us to limit changes to specific modules and reduced reliance on functionality from the native thread libraries. For example, it was straightforward to create new memory allocation modules for the cluster, and the behavior of the system with respect to memory allocation was modified easily.

- The use of SVM was challenging in unexpected ways. While it was known in advance that the release memory consistency model would require extra care in the placement of locks, it was not known that there would be limitation on the use of some resources provided by the SVM system. The SVM system assumes that applications use barriers as their synchronization mechanism, which is common practice in scientific applications but not in a JVM. Indeed, in the initial process of testing the cluster-enabled Jupiter, the experiments ran out of locks while Jupiter was bootstrapping, this is, even before the Java program could begin to execute!

- Some unexpected problems with the benchmarks were also encountered. In addition to replacing barriers and the random number generator in some applications, some of the benchmarks were found to assume sequential memory consistency [Bul03]. This is probably because most currently available JVMs today run on SMPs, which provide this consistency model in hardware. We were the first to find these problems, and they were reported to and acknowledged by the developers at EPCC [Bul03].
- The heavy load of Jupiter exposed protocol problems on the SVM system, which were tackled and fixed by the SVM group. This made our experiments difficult to carry out when crossing node boundaries (i.e., 1 to 2 nodes, 2 to 4 and 4 to 8).
- Finally, there is a noticeable lack of tools (e.g., debuggers) for developing and debugging applications on the cluster. This added an extra degree of difficulty in enabling Jupiter to run on the cluster, which increased with the addition of cluster nodes.

6.2 Future Work

Although this work has successfully implemented a functional JVM on an SVM cluster, there are still various directions from where this system can be improved.

- JIT compilation is a very effective technique for improving the performance of Java programs [AAB+00, IKY+99]. At present, Jupiter does not possess the necessary infrastructure to support it. Thus, in order to gain further acceptance, Jupiter requires the addition of a framework for JIT compilation, that will also serve as a tool for further research in this area.
- The cluster-enabled Jupiter does not utilize a garbage collector, because Boehm's garbage collector [Boe02b], which was part of the single-threaded and multithreaded versions of Jupiter, is not suitable for execution on the cluster [Boe02a]. Thus, an

interesting venue for future work is to explore the design and implementation of a distributed garbage collector that is also compatible with the memory management routines in the SVM system.

- The cluster-enabled Jupiter does not currently support single-image I/O. Thus, it is interesting to incorporate this functionality either in Jupiter or CableS.
- In the cluster-enabled Jupiter, objects are conservatively stored in shared memory. This creates some performance degradation. If it could be ascertained that an object is not shared among threads, it would be possible to store it in non-shared memory, resulting in some performance improvements. Such knowledge can be obtained through a compilation technique called escape analysis [CGS$^+$99]. Thus, it is interesting to incorporate support for such technique into Jupiter.
- The SVM system sets some limitations in the use of some of its resources. It is desirable to alleviate these limitations by releasing such resources at non-barrier synchronization points, such as locks. Moreover, the SVM system could provide better tools, or a set of common practices, for sharing parameters between a parent and a child thread. This would benefit all applications using the SVM libraries, since it would avoid the development of complex procedures such as those described in Sections 4.2.5 and 4.2.6.
- Some Java Grande benchmark applications violate the Java Memory Model by assuming sequential memory consistency [Bul03]. Thus, it is required to redesign and implement these applications to make them conform to the Java model. Furthermore, the benchmarks lack an efficient implementation of barriers in Java, which regrettably, are not semantically correct with respect to the Java Memory Model [EPC03, Jav03] either. Thus, these barriers should be revised for their use on SMPs as well as clusters.
- This work uses a page-based DSM system for the implementation of distributed memory. It is interesting to evaluate if Jupiter can benefit from the use of object-based DSM, which would allow more control over smaller memory regions (where

Java objects are stored), thus allowing them to remain consistent as a single unit. This may also reduce the cost of false sharing usually associated with page-based systems.

APPENDICES

APPENDIX A

Benchmark Details

Two sets of benchmarks were selected for the experimental evaluation detailed in Chapter 5, SPECjvm98 [SPE03] and Java Grande [EPC03, Jav03]. This appendix gives some details on what types of applications they run.

A.1 SPECjvm98

SPEC, which stands for Standard Performance Evaluation Corporation [SPE03], is an organization that develops a series of benchmarks using a standardized suite of source code, based upon existing applications. The use of already accepted and ported source code is advantageous because it reduces the problem of making comparisons between incompatible implementations.

The following benchmarks comprise this test platform:

- `201_compress` implements a modified Lempel-Ziv compression algorithm [Wel84]. It compresses real data from files instead of synthetically generated data, making several passes through the same input data.
- `202_jess` is an expert shell system [JES03], which continuously applies a set of `if-then` statements, called rules, to a set of data. The benchmark solves a set of puzzles, searching through progressively larger rule sets as execution proceeds.

- `209_db` performs multiple functions on a memory-resident database. It reads a file which contains records with names, addresses and phone numbers of entities and a file which contains a stream of operations to perform on those records. The commands performed on the file include add, delete, find and sort operations.

- `213_javac` is the commercial version of the Java compiler from the JDK 1.0.2.

- `222_mpegaudio` is the commercial version of an application that decompresses ISO MPEG Layer-3 audio files [MPE03]. The workload consists of about 4 Mb of audio data.

- `205_raytrace` is a raytracer that works on a scene depicting a dinosaur, where the threads render the scene in the input file, which is 340 Kbytes in size. This is the only multithreaded benchmark is this suite. `227_mtrt` is also part of this application.

- `228_jack` is a commercial Java parser generator. The workload consists of a file which contains instructions for the generation of jack itself. The results are then fed to jack, so that the parser generates itself multiple times.

A.2 Java Grande

EPCC stands for Edinburgh Parallel Computing Centre. Located at the University of Edinburgh, its objective is *"the effective exploitation of high performance parallel computing systems"* [EPC03]. EPCC is a participant in the Java Grande Forum[1] [Jav03] activities, developing a benchmark suite to measure different Java execution environments against each other.

EPCC provides several sets of benchmarks, aiming at sequential, multithreaded and message passing execution. For the purpose of this work, the multithreaded set was elected, which was designed for parallel execution on shared memory multiprocessors. The message passing benchmarks were designed for parallel execution on distributed

[1]The Java Grande Forum is an initiative to promote the use of Java for the so-called "Grande" applications. A Grande application is an application which has large requirements for memory, bandwidth or processing power. They include computational science and engineering systems, as well as large scale database applications and business and financial models.

memory multiprocessors. The multithreaded suite is divided in three sections, containing the following benchmarks:

- *Section 1.* This section measures the performance of low level operations such as barriers, thread fork/join and `synchronized` methods/blocks. These benchmarks are designed to run for a fixed period of time, the number of operations executed in that time is recorded and the performance reported as operations/second.

  - `Barrier` measures the performance of barrier synchronization. Two types of barriers are tested: one using `signal` and `notify` calls and a lock-free version using arrays and busy waits. The latter is not formally guaranteed to be correct under the Java Memory Model. Nevertheless, it is used in Sections 2 and 3 of the benchmark suite. This was replaced with an interface to the cluster native barriers, as shown in Section 5.1. Performance is measured in barrier operations per second.
  - `ForkJoin` measures the time spent creating and joining threads. Performance is measured in fork-join operations per second.
  - `Sync` measures the performance of `synchronized` methods and `synchronized` blocks in the presence of multiple threads, where there is guaranteed to be contention for the object locks. Performance is measured in synchronization operations per second.

- *Section 2.* This section contains short codes with the type of computations likely to be found in Grande applications. For each benchmark the execution time and the performance in operations per second (where the units are benchmark-specific) are reported.

  - `Series` computes the first Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval [0,2]. Each iteration of the loop over the Fourier coefficients is independent of every other loop and the work may be distributed between the threads, which is divided evenly between the threads in a block fashion. Each thread is responsible for updating the elements of its own block. Performance units are coefficients per second.

- `LUFact` solves a linear system using LU factorization followed by a triangular solve. This is a Java version of the Linpack benchmark. The factorization is parallelized, but the remainder is computed in serial. Barrier synchronization is required before and after the parallel loop. The work is divided between the threads in a block fashion. It is memory and floating point intensive. Performance units are Mflops per second.
- `Crypt` performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array. It is bit/byte operation intensive. This algorithm involves two loops, whose iterations are independent and are divided between the threads in a block fashion. Performance units are bytes per second.
- `SOR` performs 100 iterations of successive over-relaxation on a grid. This algorithm uses a "red-black" ordering mechanism which allows the loop over array rows to be parallelized. The outer loop over elements is distributed between threads in a block manner. Only nearest neighbor synchronization is required, rather than a full barrier. The performance is reported in iterations per second.
- `SparseMatmult` performs a sparse matrix vector multiplication., using a sparse matrix for 200 iterations. It involves an outer loop over iterations and an inner loop over the size of the principal arrays. The loop is parallelized by dividing the iterations into blocks which are approximately equal, but adjusted to ensure that no row is access by more than one thread. This benchmark stresses indirection addressing and non-regular memory references. The performance is reported in iterations per second.

- *Section 3*. This section runs large scale applications, intended to be representative of Grande applications. They were modified for inclusion in the benchmark suite by removing any I/O and graphical components. For each benchmark the execution time and the performance in operations per second (where the units are benchmark-specific) are reported.

  - `MolDyn` models particle interactions in a cubic spatial volume. The calculation of the forces applied to each particle involves an outer loop over all particles and an inner loop ranging from the current particle number to the total number of particles. The outer loop was parallelized by dividing the range of the iterations between the threads in a cyclic manner. This avoids load imbalances. Performance is reported in interactions per second.
  - `MonteCarlo` is a financial simulation using Monte Carlo techniques. It generates sample time series with the same mean and fluctuation as a series of historical data. The loop over a number of Monte Carlo runs was parallelized by dividing the work in a block fashion. Performance is measured in samples per second.
  - `RayTracer` measures the performance of a 3D raytracer. The outermost loop (over rows of pixels) was parallelized using a cyclic distribution for load balance. The performance is measured in pixels per second.

APPENDIX B

Stack Design

In order to give Jupiter more control over the memory allocated for the Java stack, an alternative stack design was implemented. The goal of this implementation is to reduce the memory requirements of a running thread (or make them more accurate to the program's needs). This goal required a redesign of Jupiter's fixed-size stack segment to a more flexible one. In the following sections, the design options for this are explored and some experimental evaluation results are given.

B.1 Java Stack Overview

The Java stack[1] is composed of stack frames, as defined in Section 3.5.2 of the JVM Specification [LY99]. Each of these frames contains the state of one Java method invocation. When a thread calls a method, the JVM pushes a new frame onto the thread's stack. When the method completes, the JVM pops and discards the frame for that method.

In its original configuration, Jupiter [Doy02] uses a fixed-size stack segment. This is a region of memory allocated as part of Jupiter's initialization process, which is used to store the Java stack during the execution of the Java program. Since a fixed-size segment cannot be resized if needed, this creates some problems with the use of memory. If the stack segment is too small, programs may run out of stack memory; if it is too large, they waste it. The implementation of a different policy for managing the stack, called

[1]In this section, the term "stack" will be used to refer to the Java stack.

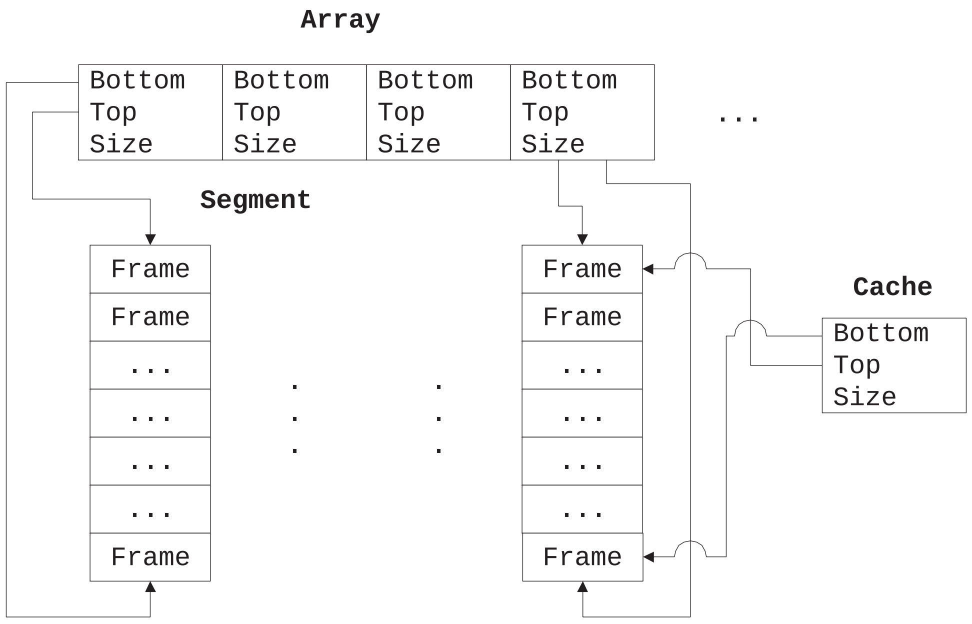


Figure B.1: Variable-Size Stack Design

variable-size stack, aims at giving Jupiter more flexibility. It allows Jupiter to make the stack grow when it is required.

B.2 Variable-Size Stack Segment

Figure B.1 shows the design for the variable-size stack segment. In this design, several memory segments comprise the stack segment. An array is used to point to each of these memory segments, which are allocated and used depending on Jupiter requirements. When one of the segments is full, a new one is allocated and referenced from the following index location of the array. Access to the upper and lower limits of each of the segments must then be controlled. When these points are reached, they are treated as special cases that make the program switch to the previous or the following segment accordingly.

In order to avoid further performance penalties for the use of extra arithmetic, every time a different segment is used, the important values used for accessing it, such as bottom and top memory addresses, and segment size, are stored in a data structure that serves as a cache. This way, these values can be accessed from the cache instead of dereferencing the pointers and performing arithmetic operations on the array. Consequently, the checks performed do not differ much from the original implementation, and only extra checks were added around the segment boundaries.

Each of the stack segments can be handled individually. They can remain allocated for the life of a Jupiter instance, be garbage collected or even manually deallocated. It must be noted that the array used for referencing the memory segments can be either of fixed size or variable size. For simplicity, a fixed-size array is used in the current design.

An alternative design for the stack is the use of a single memory segment, which can be resized using the `realloc` call[2] according to the needs of Jupiter. This has the advantages that it does not require any extra computation derived from the use of additional data structures, and it allows both to grow and shrink the stack as needed. The disadvantage is that the POSIX standard allows for `realloc` to move memory segments, thus changing their memory addresses and invalidating any pointer that references them. This requires a careful implementation of the stack, and the storing of pointers and cached values. It is necessary that there not be a pointer reference to a frame in the stack when there is a possibility that the stack be moved.

However, it must be noted that having such a flexible stack implementation, with no pointer references to it, is the first step toward thread migration.

The advantage of the variable-size stack is that it allows for the creation of policies on how the segments grow, with less burden on the operating system for finding a sufficiently large memory segment to expand or move the stack. In the variable-size design, new stack segments do not need to be contiguous to those previously allocated. The disadvantage is that it requires some extra arithmetic to handle the memory structures that hold the individual stack segments.

[2]Both designs can be used on the cluster without any further modification of Jupiter for using the SVM system. In the cluster, stacks are allocated in private memory and handled using the libc libraries and not the cluster memory interface, which currently does not support `realloc`.

Benchmark	**MTJ**	**VSSJ**	**VSSJ/MTJ**
201_compress	391.90	381.53	0.97:1
202_jess	135.47	132.80	0.98:1
209_db	191.68	195.93	1.02:1
213_javac	154.40	164.81	1.07:1
222_mpegaudio	301.69	299.73	0.99:1
228_jack	85.86	88.95	1.04:1
		Geometric Mean	**1.01:1**

Table B.1: Benchmark Times for the Multithreaded Jupiter with a Fixed-Size Stack with Respect to a Variable-Size Stack (VSSJ) Using the SPECjvm98 Benchmarks

B.3 Experimental Evaluation

A performance comparison was made between the multithreaded Jupiter using the fixed-size stack and the variable-size stack. In these experiments, the following configuration was used for the variable-size stack. The main array contains 16 entries, each single stack segment is the same size as a memory page in Linux (i.e., 4 Kbytes), and the stack grows uniformly (i.e., all the segments are the same size). All unused fragments are left for the garbage collector. This configuration with a small stack guarantees that this module is thoroughly used in its most critical paths in at least one case. Setting a larger-stack configuration would not provide much information because stack expansions will not likely be exercised.

Tables B.1[3], B.2 and B.3 show the performance difference in Jupiter with the fixed-size and variable-size stacks for the SPECjvm98, Java Grande, and Java Grande Stress benchmarks, respectively. Although it belongs to the SPECjvm98 benchmark suite, the results from `205_raytrace` are included in Table B.2, with the evaluation of the multithreaded benchmarks in the Java Grande suite.

This module introduces an average slowdown of about 1% for the singe-threaded and multithreaded benchmarks, with speed improvements in some cases and a significant slowdown in one case. For the stress benchmarks, it can be observed that this brings some improvements, in particular for `ForkJoin`. The variable-size stack version of Jupiter can

[3]The acronyms and units used for presenting these results are the same as those used in Chapter 5.

1 Thread			
Benchmark	**MTJ**	**VSSJ**	**VSSJ/MTJ**
205_raytrace	32.52	34.90	1.07:1
Series	72.06	72.91	1.01:1
LUFact	26.14	26.62	1.02:1
Crypt	22.25	22.59	1.02:1
SOR	150.39	149.42	0.99:1
SparseMatmult	52.37	53.01	1.01:1
MolDyn	361.90	349.88	0.97:1
MonteCarlo	134.91	135.53	1.00:1
RayTracer	574.60	575.71	1.00:1
Geometric Mean			**1.01:1**
2 Threads			
Benchmark	**MTJ**	**VSSJ**	**VSSJ/MTJ**
205_raytrace	27.62	29.02	1.05:1
Series	43.04	43.70	1.02:1
LUFact	13.85	14.00	1.01:1
Crypt	12.11	12.58	1.04:1
SOR	77.27	77.28	1.00:1
SparseMatmult	26.60	26.90	1.01:1
MolDyn	182.82	180.44	0.99:1
MonteCarlo	72.67	73.20	1.01:1
RayTracer	290.00	291.35	1.00:1
Geometric Mean			**1.01:1**
4 Threads			
Benchmark	**MTJ**	**VSSJ**	**VSSJ/MTJ**
205_raytrace	29.35	33.90	1.16:1
Series	22.89	22.74	0.99:1
LUFact	7.75	7.74	1.00:1
Crypt	7.07	7.47	1.06:1
SOR	40.71	40.88	1.00:1
SparseMatmult	14.74	14.99	1.02:1
MolDyn	94.89	93.04	0.98:1
MonteCarlo	40.88	41.33	1.01:1
RayTracer	199.62	185.43	0.93:1
Geometric Mean			**1.01:1**

Table B.2: Benchmark Times for the Multithreaded Jupiter with a Fixed-Size Stack with Respect to a Variable-Size Stack (VSSJ) Using the Java Grande Benchmarks (Size A)

1 Thread			
Benchmark	**MTJ**	**VSSJ**	**VSSJ/MTJ**
Barrier:Simple	593708.31	593923.12	1.00:1
Barrier:Tournament	463484.41	464356.84	1.00:1
ForkJoin:Simple	4275599.50	4149112.00	0.97:1
Sync:Method	38569.08	38291.19	0.99:1
Sync:Object	39630.60	39151.63	0.99:1
		Geometric Mean	**0.99:1**
2 Threads			
Benchmark	**MTJ**	**VSSJ**	**VSSJ/MTJ**
Barrier:Simple	59084.70	56774.39	0.96:1
Barrier:Tournament	281715.28	251773.20	0.89:1
ForkJoin:Simple	1064.55	2956.97	2.78:1
Sync:Method	25168.76	25528.04	1.01:1
Sync:Object	25161.12	25140.38	1.00:1
		Geometric Mean	**1.19:1**
4 Threads			
Benchmark	**MTJ**	**VSSJ**	**VSSJ/MTJ**
Barrier:Simple	11524.10	12479.68	1.08:1
Barrier:Tournament	151095.59	151325.28	1.00:1
ForkJoin:Simple	No memory	No memory	
Sync:Method	14383.81	14376.10	1.00:1
Sync:Object	14575.08	14587.03	1.00:1
		Geometric Mean	**1.02:1**

Table B.3: Stress Tests for the Multithreaded Jupiter with a Fixed-Size Stack with Respect to a Variable-Size Stack (VSSJ) Using the Java Grande Benchmarks

create close to 3 times the number of threads as the fixed-size stack version. On early experimentation, the `ForkJoin` benchmark ran out of memory when using a fixed-size stack configuration in some systems, and the variable-size stack enabled this benchmark to run properly.

The slowdown is introduced when crossing the stack segment boundaries repeatedly, which is a case that requires special handling. Another cause is the constant expansion and shrinking of the stack. The speed improvement is obtained for those cases where a small stack is sufficient for running the Java application and the stack needs remain fairly constant.

BIBLIOGRAPHY

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. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling multithreaded Java bytecode for distributed execution. *Lecture Notes in Computer Science*, 1900:1039–1052, 2001.

[ACD$^+$96] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[AFT99] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[AFT$^+$00] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. Transparently obtaining scalability for Java applications on a cluster. *Journal of Parallel and Distributed Computing*, 60(10):1159–1193, 2000.

[AGMM99] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. High performance numerical computing in Java: Language and compiler issues. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–17, 1999.

[AWC$^+$01] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik - a distributed JVM on a single address space architecture, 2001.

[Azi02] R. Azimi. miNi: Reducing network interface memory requirements with dynamic handle lookup. Master's thesis, Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 2002.

[BAL$^+$01] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 92–103. ACM Press, 2001.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[BJS01] A. Bilas, D. Jiang, and J. P. Singh. Accelerating shared virtual memory via general-purpose network interface support. *ACM Transactions on Computer Systems*, 19(1):1–35, 2001.

[BLS99] A. Bilas, C. Liao, and J. P. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, pages 282–293, 1999.

[Boe02a] H. Boehm. Hewlett-Packard Company, 2002. Personal communication.

[Boe02b] H. Boehm. A garbage collector for C and C++, 2002. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[Bul03] J. M. Bull. Edinburgh Parallel Computing Centre (EPCC), 2003. Personal communication.

[CBD+98] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A mechanism for address translation on network interfaces. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 193–204, 1998.

[CGS+99] J. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, pages 1–19, 1999.

[CJV00] Cluster virtual machine for Java. International Business Machines Corporation, 2000. `http://www.haifa.il.ibm.com/projects/systems/cjvm/`.

[CSG99] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture. A Hardware/Software approach*, pages 23, 513–521. Morgan Kaufmann Publishers Co, 1999.

[CWH00] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial, 3rd edition.* Addison-Wesley Publishers Co, 2000.

[DA02] P. Doyle and T. S. Abdelrahman. Jupiter: A modular and extensible JVM infrastructure. In *Proceedings of the USENIX 2nd Java Virtual Machine Research and Technology Symposium*, pages 65–78, August 2002.

[DKL+00] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanorer. Implementing an on-the-fly garbage collector for Java. In *Proceedings of the 2nd International Symposium on Memory Management*, pages 155–166. ACM Press, 2000.

[Doy02] P. Doyle. Jupiter: A modular and extensible Java virtual machine framework. Master's thesis, Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 2002.

[EPC03] Edinburgh Parallel Computing Centre (EPCC). University of Edinburgh, 2003. `http://www.epcc.ed.ac.uk/`.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification, 2nd edition*. Addison-Wesley Publishers Co, 2000.

[GNU03] GNU Classpath. GNU, 2003. `http://www.gnu.org/software/classpath/classpath.html`.

[HPO01] HP OpenVMS systems documentation. Guide to the POSIX threads library. Hewlett-Packard Company, 2001.

[IEE90] *System Application Program Interface (API) [C Language]*. Information technology - Portable Operating System Interface (POSIX). IEEE Computer Society, 1990.

[IEE93] *IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface*. IEEE, Inc., August 1993.

[IKY$^+$99] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 119–128. ACM Press, 1999.

[JAL01] Jalapeño. International Business Machines Corporation, 2001. `http://researchweb.watson.ibm.com/jalapeno`.

[Jam02a] P. Jamieson. University of Toronto, 2002. Personal communication.

[Jam02b] P. A. Jamieson. Cables: Thread and memory extension to support a single shared virtual memory cluster image. Master's thesis, Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 2002.

[Jav03] Java Grande Forum, 2003. `http://www.javagrande.org/`.

[JB02] P. A. Jamieson and A. Bilas. CableS: Thread control and memory management extensions for shared virtual memory clusters. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, pages 263–274, 2002.

[JES03] Jess. Sandia National Laboratories, 2003. `http://herzberg.ca.sandia.gov/jess/`.

[JIK03] Jikes research virtual machine. International Business Machines Corporation, 2003. `http://www-124.ibm.com/developerworks/oss/jikesrvm`.

[Jog03] Jogether, 2003. `http://sourceforge.net/projects/jogether`.

[JOY+99] D. Jiang, B. O'Kelley, X. Yu, S. Kumar, A. Bilas, and J. P. Singh. Application scaling under shared virtual memory on a cluster of SMPs. In *Proceedings of the 13th international conference on Supercomputing*, pages 165–174. ACM Press, 1999.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[KIS03] Kissme, 2003. `http://kissme.sourceforge.net`.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LC96] T. Lovett and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 308–317, 1996.

[Lea99] D. Lea. *Concurrent Programming in Java. Design Principles and Patterns, 2nd edition*, pages 90–97. Sun Microsystems, 1999.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[LY96] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification, 1st edition.* Addison-Wesley Publishers Co, 1996.

[LY99] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification, 2nd edition.* Addison-Wesley Publishers Co, 1999.

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

[MOS+98] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT a reflective Java JIT compiler. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, 1998.

[MPE03] MPEG layer-3 audio specification. Fraunhofer Institut fuer Integrierte Schaltungen, 2003. `http://www.iis.fraunhofer.de/amm/`.

[MWL00] M. J. M. Ma, C. Wang, and F. C. M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.

[Myr03] Myrinet. Myricom Inc., 2003. `http://www.myri.com`.

[NBF96] B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming. A POSIX Standard for Better Multiprocessing, 1st edition.* O'Reilly & Associates, September 1996.

[OPE97] The single UNIX specification, version 2. The Open Group, 1997. http://www.opengroup.org.

[OPE01] OpenJIT. Tokyo Institute of Technology, 2001. http://www.openjit.org.

[ORP02] Open Runtime Platform. Intel Corporation, 2002. http://orp.sourceforge.net.

[PS95] D. Plainfosse and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 211–249, 1995.

[Pug99] W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 89–98, 1999.

[Pug00] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

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

[SAB03] SableVM, 2003. http://www.sablevm.org.

[SGT96] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the* 7^{th} *Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, 1996.

[SHT98] W. Shi, W. Hu, and Z. Tang. Shared virtual memory: A survey. Technical Report 980005, Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences, April 1998.

[SPE03] SPEC. Standard Performance Evaluation Corporation, 2003. http://www.specbench.org/.

[SUN03] Java Depelopment Kit (JDK). Sun Microsystems Inc., 2003. http://java.sun.com.

[Wel84] T. A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

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

[WMMG99] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 109–118, 1999.

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

[ZWL02] W. Zhu, C. Wang, and F. C. M. Lau. Jessica2: A distributed Java virtual machine with transparent thread migration support. In *IEEE 4th International Conference on Cluster Computing*, Chicago, USA, September 2002.