Distributed Java Virtual Machine with Thread Migration

Wenzhang Zhu

朱文章

A thesis submitted
in partial fulfillment of the requirement for
the degree of Doctor of Philosophy
at the University of Hong Kong

August, 2004

This page intentionally left blank

Abstract of thesis entitled

"Distributed Java Virtual Machine with Thread Migration"

submitted by **Wenzhang Zhu**

for the degree of **Doctor of Philosophy**

at the University of Hong Kong in **August 2004**

Recent advances in better Java class libraries and Just-in-Time (JIT) compilation techniques have greatly improved the performance of Java to match that of C/C++. To fully exploit Java's multithreading feature on clusters, an attractive goal is to extend the Java Virtual Machine (JVM) to be "cluster-aware" so that a group of JVMs running on distributed nodes can work together as a single, more powerful JVM to support true parallel execution of a multithreaded Java application. In a cluster-aware JVM, the Java threads created within one program can run on different cluster nodes to achieve a higher degree of execution parallelism. The distributed system would provide all the virtual machine services to Java programs, and should be fully compliant with the Java language specification. We refer to such a distributed system as a Distributed JVM (DJVM).

In our study, we solve the problem of transparent Java thread migration in a JIT-enabled DJVM. The problem of thread migration is to suspend a running thread in a multithreaded application on one node, and resume its execution on a target node, such that the thread continues to execute and communicate with other threads of the same application on different nodes. In this act, the challenges lie in dealing with different types of pointers associated with the context of the migratory thread. The JIT compilation in JVM has made the native code of Java methods be dynamically placed, thus the method pointer relocation becomes difficult. Shared Java objects must continue to be valid after thread migration, which calls for proper and efficient memory consistency protocol in DJVM. The Java thread stack of the migratory thread also imposes difficulty when it is relocated on the target node.

Our study provides a new thread migration methodology by using JIT compilers to tackle the above pointer problem. The native code instrumentation approach and the JIT recompilation approach are proposed to transform the native thread context into a portable thread context that can be used to efficiently relocate different types of pointers during thread migration. To support the validity of object pointers after migration, we propose a global object space (GOS) embedded in DJVM that can provide a single system image (SSI) illusion to Java threads. This study presents the design, implementation, and analysis of our DJVM named JESSICA2. We provide in-depth study on thread migration, as well as the optimization on distributed object sharing in DJVM based on Java memory model. Performance study has confirmed the efficiency of thread migration in JESSICA2.

Declarations

I hereby declare that the thesis entitled “Distributed Java Virtual Machine with Thread Migration” represents my own work and has not been previously submitted to this or any other institution for a degree, diploma and other qualifications.

Wenzhang Zhu
2004

Acknowledgements

The Ph.D. study at HKU has been an unforgettable and pleasant time in my life. There are so many good friends and colleagues around who have offered helps and advices to me. I will take the opportunity to express my sincere thanks to them.

I would first thank my supervisors Dr. Cho-Li Wang and Dr. Francis Chi Moon Lau for their patient guidance, encouragement, and advices in both research and daily life. I would like to thank Dr. Wang for teaching me how to do research and guiding me in the research of distributed Java Virtual Machine. He has been so careful and energetic in correcting every mistake in my papers. I really learned a lot from his comments. Dr. Wang is also a very generous and kind teacher and friend in daily life. He often invited us to his party and we always had a good time together with his lovely family. I also want to thank Dr. Lau for showing us how to do research that is serious and of great impact. His advices on research is invaluable. I also appreciate Dr. Lau's revision on papers. The papers always looked much better after his refinement. Finally I do appreciate their support in finance for the research. I would also like to thank Prof. Zhi-wei Xu from the Institute of Computing Technology, Chinese Academy of Science, who recommended me for the Ph.D. study in HKU for conducting this research.

I would like to express my appreciation in working with Fang Weijian during the Ph.D. study. We had good cooperations in the research project. We also had a good time in our leisure time. We organized hiking often, and discussed research problems at the same time. Through hiking we developed good bodies for research. Here I would like to list other partners and thank them for sharing the joy of hiking. They are Gaoyan Luo, Tingting He, Lin Chen, Hui Zhang, Xin Zhang, Tianqi Wang, Weisong Chen, Donglai Zhu, Wei Li, Zhenlong Bai, Huimin Deng, Tianchi Ma, Kangkai Yin, Jingdong Zhang, etc.

I would like to thank my colleagues who have worked in the same office with me during the past years. They are Chungen Ouyang, Huidong Li, Jian Wu and others. We have had a good time in sharing the office and learned a lot from each other.

I would like to thank the members in our research lab who offered many helps to me or had a good time with me. Anthony Tam and Roy Ho have helped a lot in organizing the research environment and gave a lot of useful advices. I also had a good time in working with Ge Chen , Jiang Zhu , David C. M. Lee, Benny W. L. Cheung, Sit Yu Fai, Chow Yuk, Kwan Vivien, Matchy C.M. Ma, Zoe Ching Han Yu, Ling Zhuo, Ye Wang, Zhang Xiaolei, Nalini Belaramani, Ken Cheuk Kan Tsang, Mark Cheuk Man Tsang and many others.

I would like to thank the professors in CS department of HKU for offering such excellent courses in my postgraduate study. The lecturers who course I took include my supervisor Dr. Cho-Li Wang, Dr. H. F. Ting, Dr. David Cheung, Dr. Lucas Hui and Dr. T. W. Lam. I would also like to thank the staffs in CS department of HKU. They have created an excellent research environment.

I would like to thank other friends who I know well during my study in HKU. They are Xiaobin Zhang, Jinbo Xu, Yijun Pan, Yudong Sun, Jie Zhang, Lian Wang, Huiping Cao, Yutao Shou, Lin Cheung, Yongjian Wu. I also want to thank the external reviewers and examiners for their valuable comments on the project and my thesis.

Last but not least, I thank my parents and the family for their love and support!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A computer cluster is a group of connected commodity computers working together to appear as a single system [13]. The cluster is becoming the dominant high-performance parallel architecture. There are 294 clusters among the 500 most powerful computers in the world, according to the latest (Nov. 2004) top 500 supercomputer list [71]. The popularity comes from the fact that clusters are cost-effective as all the hardware components are inexpensive commodities.

Programming cluster is hard. The existing parallel languages or runtime supports, such as Parallel Virtual Machine (PVM) [36], Message Passing Interface (MPI) [49], and software Distributed Shared Memory (DSM) [4, 18, 47, 64, 66], are not easy to use. To use these tools, the end users must write in special Application Programming Interfaces (APIs), explicitly partition the workload and data on different nodes, and manually coordinate different processes on cluster nodes. The lack of programmability has harmed the productivity of applications to take advantage of the cluster hardware.

The Java programming language [37] has potential to be a better environment for high-performance cluster computing. Java is well-known for its clean and simple object semantics, cross-platform portability. It is ubiquitous on nearly all computer architectures today. Recent results indicate that

Java can deliver 65–90% of the best Fortran performance for a variety of benchmarks [9] and can compete with the performance of C++ [50]. Multithreading is a built-in feature of Java to support concurrency programming, which could make parallel programming in Java more nature. Due to the large population of Java programmers, the productivity of cluster software could be boosted by using Java.

Java programs are running on top of a *Java Virtual Machine* (JVM). The current JVM technology is limited in supporting high-performance execution of multithreaded Java applications. Standard JVMs, such as Sun JVM [61], IBM's JVM [67], etc., mainly emphasize on high-speed native execution on single node. However, these systems cannot scale to large settings such as clusters.

In our research, we study *distributed Java Virtual Machine* (DJVM) which can provide a high-performance execution platform for multithreaded Java applications on clusters. The research will encourage parallel programming in Java to improve the productivity of cluster software.

The DJVM is a distributed system that consists of a group of "cluster-aware" JVMs running on clusters working together as a single, more powerful JVM. With a DJVM, the Java threads created within one program can run on different cluster nodes in parallel. The DJVM system provides all the virtual machine services to Java programs like a standard JVM, and is fully compliant with the Java language specification. The DJVM approach allows the existing multithreaded Java programs fully utilize the available computing resources in all participating nodes, unlike the stand-alone JVM which is limited by all the available processors in a single server.

We believe that in order to exploit the enhanced computation powers of clusters, and to make the system scalable, a DJVM needs the following functions.

- High-speed native execution. To improve Java program's performance, a Just-in-Time (JIT) compiler should be incorporated in a DJVM. A

JIT compiler dynamically translates Java bytecode into native code during execution and runs many times faster than an interpreter [23, 61]. JIT compilation removes the time-consuming interpretation layer between the application and the hardware. The hardware machine registers can be directly manipulated by the compiled native code. Moreover, larger scope of program code can be grouped together for code optimization in JIT compilation mode.

- High parallelism. The cluster provides a large-scale parallel hardware platform. To support the high-performance execution of multithreaded Java applications on such a platform, cluster-wide multithreading should be a built-in feature of the DJVM. With the feature, Java threads can be efficiently mapped to different cluster nodes to support large-scale parallel computation.

- Resource utilization and load balancing. Efficient resource utilization and load balancing are essential for enabling large-scale multithreaded Java applications to achieve scalable performance on clusters. The system should support preemptive thread migration which allows a thread to move between machines during its execution. In this way, more advanced thread scheduling policy can be based on the migration support to balance the system workload, and new idle machines can dynamically join a program's execution to improve the performance on the fly.

- Efficient distributed object access. In traditional Java Virtual Machine, all threads share a single heap for data accesses. To provide the same support in a DJVM, a distributed shared heap that is accessible by all distributed Java threads should be provided. Optimization techniques need to be invested in reducing object sharing overheads.

- Programmability. No new APIs or language modifications should be

> introduced in order to use DJVM. The programmer can write the multithreaded Java programs in usual manner. The existing program that can run on a single-node JVM can run on a DJVM without any modification. Single-system image illusion should be provided in the DJVM to ensure the programmability.

There are several research projects targeting at multithreading support on clusters using the DJVM concept [7, 8, 52, 74, 82]. Many of the useful functions are missing in these prototypes. Firstly, the JIT compilation, which is a key component for accelerating the Java's execution speed, is not enabled to support high-speed execution on clusters in most of these projects. Some projects [8, 52, 82] run Java programs in interpretive mode. As a result, they suffer from much slower execution speed. Secondly, load balancing mechanisms are supported in a limited way. Most of the multithreading projects on clusters, map each Java thread to a fixed node during its life time, i.e., once a thread is placed on a node, it is fixed until it terminates. There is no support for re-mapping threads according to the system workload at runtime. For a multithreaded Java application with imbalanced workloads among threads, the overloaded nodes will slow down the execution time of the whole application. Thirdly, the feature of dynamically loading Java classes is disabled in some DJVM projects adopting static compilers. A few DJVMs [7, 74] statically compile a Java program into a stand-alone parallel application. The whole application package needs to be ready before execution. Such an approach prohibits the dynamic loading of Java bytecode from a remote machine. Thus they cannot be fully compliant with the Java language specification. Fourthly, efficient object access is lacking in most of the projects. Some DJVMs [52, 82] use page-based DSM to provide the object sharing among cluster nodes. The page-based DSM technologies will suffer false sharing problem in supporting the distributed shared object for Java, since the access unit of the page-based DSM is a page, which is different from a Java object. Other DJVMs [7, 74] link existing OO-based DSM libraries in the compiled binary to support distributed object sharing. Though the OO-based DSM's granularity matches a Java object, the objects stored in the DSM are usually regarded as independent ones. The information of object types and accessing thread is not easy to be passed to the underlying DSM for optimizations on reducing object sharing overheads.

1.1 Research challenges

We identify the problem of supporting thread migration running in JIT compilation mode as the most challenging problem to solve in the DJVM. Our research will focus on finding efficient solutions to tackle this problem.

The problem of thread migration is to suspend a running thread in a multithreaded application on one node, copy its context to a target node and resume its execution, such that the thread continues to execute and communicate with all the other threads of the same application among cluster nodes. As JIT compilation has become the de facto execution mode of Java programs in JVM, Java thread migration system is inevitable to be JIT-enabled.

1.1.1 Existing technologies

In process migration systems [55], such as MOSIX [15], and OpenSSI [75], the virtual space of a process in the source node is duplicated on the target node in the same layout. Both virtual spaces on the source node and the target node start from zero. Therefore the virtual memory addresses of all the entities in the process, such as the code, the stack, and the data, remain the same. These entities are accessed by the program through pointers, which remain valid after migration.

Thread migration is different from process migration that it copies a single thread's context to a remote machine, instead of copying the whole virtual address space. As the context is dynamically allocated on the target node, its offset could be different from that of the source node. The entities in the context cannot be referred to by the original pointers. As a result, resolution of pointers in the thread context becomes a critical problem in thread migration system.

In the past, several distributed systems supporting C thread migration have been proposed [6, 29, 44]. Three types of pointers in thread context need to be resolved, namely, the code pointer, the data pointer, and the intra-stack pointer.

- *The resolution of code pointer.* In the C thread migration system, since the program code is statically linked, the code pointer is not a big problem when the whole program code is loaded in the same virtual address on each node. The exceptional case in code pointer is the pointer to the shared libraries. This case can be solved by similar solutions used in loading the program code before execution in the OS [17]. The external references to shared libraries have been included in the execution binary, therefore the resolution module in thread migration systems just needs to call the proper OS functions to resolve the entries to the shared libraries or system calls.

- *The interpretation of data pointer.* When a migratory thread arrives at the destination node, the system should guarantee that the data pointers used by the thread still make sense. The resolution of the data pointer can be solved by bringing the data copy to the destination node, and directing the data pointer to the virtual address of the copy. However, since multithreading implies that the global data are shared among the threads, memory consistency support should be called for to ensure the correct communication of the threads based on the shared variables. Usually, thread migration systems have an underlying DSM support [6, 29, 44]. The data pointer in these systems is usually interpreted as the global id of the data.

- *The handling of intra-stack pointer.* The intra-stack pointer is the most difficult one among the three types of pointers. Such pointers can point to the local variables which are allocated in the thread stack, or they can be the frame pointers generated by the compilers. The intra-stack pointer will point to incorrect location after migration unless the thread stack is allocated at the same virtual address on target node. Unlike the code which is loaded once when the process is created on target node, the thread stack on typical systems usually cannot guarantee to be in the same virtual address as that in the source node because it is allocated dynamically from the OS on the target node.

  In MILLEPEE [44], a predefined number of thread stacks are allocated at initialization time on every node. When a thread migrates to another node, the stack with the same starting address reserved on the target node will be used for the migratory thread. In this way, the intra-pointers remain the same. However, it sacrifices the flexibility of thread creation by fixing the total thread number. Also it consumes much more resources than necessary since all nodes will reserve all thread stacks, even some of which may not be used. Some systems get around the pointer problems by disallowing the use of data pointers and intra-stack pointers in migrated threads, or accepting undefined pointers after migration [6]. In other systems, stack-related problems is solved by preprocessing. For example, Arachne [29] is based on a thread system called Ythreads, and restricts that the pointers in the threads must be known at compilation time.

The supporting of Java thread migration is also a hot topic in mobile agent systems. However, if we follow the traditional process migration techniques, it is not appealing for Java thread migration. This is because we need to migrate the whole JVM process even if we just want to migrate a simple Java program. Copying only the context related to only Java bytecode program without the underlying JVM process context is more compact and portable. In Sumatra [63], JVM interpreter was modified to support the extraction and the restoration of Java thread context. Since the interpreter directly operates on the Java thread context, and the interpreter calls a Java method via its symbolic name, there is no pointer issue in interpreter-based systems. However, the interpreter is slow for Java execution due to the interpretation overhead. Such a solution for thread migration is inefficient. Other approaches [65, 72] use preprocessors to rewrite bytecode programs before execution. The thread migration logic is done at the application level. Since Java bytecode has limitation in accessing the thread context, it usually needs a long sequence of bytecode instructions for fetching one single item such as the *program counter* (PC). The resulted program after rewriting is usually inefficient.

1.1.2 New challenges

The Java thread migration in JIT-enabled DJVM has raised new challenges compared to traditional thread migration systems. In the thesis, the following issues are addressed.

- Java thread stack management in JIT-enabled JVM. Java programs need a JVM process at runtime to support their execution, unlike the stand-alone execution mode of C threads. In the JIT-enabled JVM, the Java thread stack becomes native. The interleaving of frames created by Java methods and the frames created by the JVM internal functions makes it difficult to distinguish the boundaries among them. Also the resolution of the pointers inside the native thread stack raises new complexities compared with the C thread stack. Though no pointers to local variables are used in Java, in each frame of the Java thread stack, there is an operand stack used for storing the temporary computed values. Under a JIT compilation environment, the operands in the operand stack are not structured in a single data structure like the case in interpreters. They can be kept in stack slots or hardware registers. Java threads access to these operands through pointers to memory or registers, which makes the resolution of these new intra-stack pointers difficult.

- Dynamic method resolution. In static languages like C, all user functions are ordered statically in its binary file and are loaded into the code area of main memory as a unit. The offsets of the C functions are fixed on every node that loads it. For Java methods in JIT compilation mode, things are different. During the execution of a Java application on JVM, the Java methods are dynamically loaded and are compiled by the JIT compiler. The memory for each method is allocated on demand like the thread stack. It is now not possible to reserve the same method address on all nodes. Or, we have to treat them like shared library functions. We need to keep track of all the loaded Java methods, and scan the whole code space to resolve all the references to the method entries. Due to the huge memory consumption and search overhead, the approach is also unacceptable. Given the JIT-enabled execution environment, Java migration system needs new solutions for relocating the migrated methods on the target node.

- Java object sharing. Migratory threads in DJVM need to share Java objects with other threads in the distributed environment. Efficient object sharing for distributed threads is essential for high-performance multithreading on clusters. In order to support the single-system image illusion to Java threads, the Java memory model should be followed strictly in DJVM for object sharing. The memory model requires all previous updates of Java objects be accessible to a thread after it enters a critical section. On clusters, this would introduce communication among the nodes in updating the object data. The direct use of existing software DSM technologies to simulate the Java memory model, as used in previous thread migration systems, usually creates many redundant communication messages among the clusters nodes. This is because there is no channel for DJVM to pass the runtime information such as object types and accessing threads to the underlying DSM. As a result, efficient distributed object sharing is hard to achieve without designing new protocols that can make use of JVM's internal information.

1.2 Thesis statement and contributions

The dissertation studies the support for high-performance parallel execution of multithreaded Java applications using DJVM on clusters. We build a single-system image distributed Java Virtual Machine that supports large-scale multithreading, cluster-aware JIT compilation, and load balancing on clusters. The DJVM enables high-performance parallel Java computing using multithreaded model on clusters. The system is able to support the parallel execution of a single multithreaded Java application which scales to a few hundreds of computing nodes. The research advances the current DJVM in the speed, flexibility, and scalability.

Our study solves preemptive and transparent Java thread migration on DJVM running in JIT compilation mode. To address the Java thread context movement in thread migration, we proposed two efficient solutions, namely, dynamic native code instrumentation, and JIT recompilation. Both approaches work at JVM level to extract and restore portable bytecode-oriented thread context even when Java threads are running JIT-compiled methods. In the dynamic native code instrumentation approach, we propose migration points in the compiled native code. Through the method, we can achieve thread context extraction and restoration with a short latency. In the JIT recompilation approach, we eliminate the execution overhead resulted from thread migration. The approach supports the full-speed native execution of Java threads in the thread migration system. The execution overhead will only be charged when there is a thread migration. Both approaches introduce novel uses of JIT compilers to support the strong mobility of threads. The proposed thread migration techniques based on JIT compiler make it possible for portable thread migration without sacrificing native execution performance.

To address the problem of efficient distributed object sharing, we apply adaptive protocols to realize the Java memory model on clusters. We explore the issues of supporting an efficient shared memory illusion for Java applications on clusters inside DJVM. We extend the Java memory model from single physical memory space to physically distributed memory space. The research has shed lights on the performance bottleneck on execution of multithreaded Java applications on clusters.

The following summarizes the main contributions of our research.

- To our best knowledge, our DJVM prototype JESSICA2 is the first DJVM to integrate JIT compilation, thread migration, and distributed object sharing in JVM to achieve large-scale high-performance Java computing on clusters. We advance the current DJVM in execution speed by enabling JIT compilation. We provide high parallelism, better resource utilization and load balancing using thread migration. The system provides an embedded global object space to support efficient distributed object sharing and provide a single-system image illusion for Java threads to access objects and perform I/O operations with location transparency.

- We propose two new efficient approaches for solving the thread context extraction and restoration under JIT compilation mode.

    1. We propose dynamic native code instrumentation approach based on JIT compiler to support native thread context extraction with low migration latency. The approach makes it possible to extract the high-level Java thread context efficiently from the raw thread

execution context when the program is executed under JIT compilation mode, and to relocate the Java methods on the target node. The approach provides fast thread context extraction. We also propose methods to select migration points based on the control flow of the Java program to reduce the migration overhead. Such an approach is valuable for systems that need frequent thread migrations to enforce load balancing policies. The overhead charged to the migration operations is small, and it won't affect the overall system performance.

2. We propose JIT-recompilation approach to support thread context extraction and restoration in native thread stack, without the cost of any prior code instrumentation. The approach enables extraction of Java thread context in the JIT-enabled DJVM. It ensures the high-performance native execution of Java threads at the time of no thread migration. We also propose migration latency hiding in JIT recompilation approach by overlapping recompilation operations on the source node and class loading on the target node. The approach suits the systems that needs to migrate computation intensive threads to idle nodes infrequently. The migration will occur less often in these systems, therefore only minor execution overheads in performing the thread migration operation will be charged.

- We propose adaptive object home migration protocols in supporting the efficient realization of Java memory model on clusters. By extending the Java memory model to the DJVM, we use object caching to represent the working memory defined in Java memory model. Java object type information and the statistic information from the JVM threading system are used in our embedded global object space to reduce object sharing overhead. A number of optimizations, including the object pushing, fast state checking, and socket caching, have been exploited in reducing the cost of accessing objects in the distributed environment.

1.3 Organization of the thesis

The dissertation presents the design, implementation and analysis of a high-performance Distributed Java Virtual Machine named JESSICA2 on computer clusters. The organization is as follows.

Chapter 2 introduces the background of Java Virtual Machine and Distributed Java Virtual Machine. We then analyze the design issues of Distributed Java Virtual Machine and provide our design goal.

Chapter 3 discusses the related work of DJVM in details, and provides classifications of them.

Chapter 4 describes the overview of JESSICA2 Distributed Java Virtual Machine. We discuss the design rational, the architecture of JESSICA2, and the overview of its two main building blocks, the thread system and the distributed shared heap.

Chapter 5 presents the most distinguished feature of JESSICA2, i.e., its thread migration system. We show how JIT compiler is used to support the lightweight thread migration across cluster nodes. In this chapter, we first present the solution of dynamic native code instrumentation scheme. We step further to propose another advanced solution called JIT recompilation.

Chapter 6 presents the Global Object Space (GOS), an important feature of JESSICA2 that supports the transparent object access in cluster environment. We introduce the Java memory model and its realization in a distributed environment. We show the design architecture of the GOS. The optimization techniques in reducing communication overhead will be discussed.

Chapter 7 presents the performance results. We report the migration overhead under the two thread migration techniques. We evaluate the effects of the optimization techniques in GOS. We give the results of the application benchmarks and dynamic load balancing experiments.

Chapter 8 draws the conclusions and gives future directions for the research on Distributed Java Virtual Machine.

Chapter 2

Distributed Java Virtual Machine

2.1 Background

Distributed Java Virtual Machine (DJVM) is the distributed system consisting of a group of JVMs running cooperatively on different cluster nodes to support the execution of one single multithreaded Java application. In this section, we provide the background to help better understanding of the discussion of DJVM in later chapters. The background includes Java programming language, JVM, JIT compiler, clusters, single-system image, and the parallel programming paradigms of Java.

2.1.1 Java and Java Virtual Machine

Java [37] is a popular general object-oriented programming language. It supports multithreading programming at the language level. A Java program comprises a number of reusable components called classes, which contain both data and methods. Unlike most of other programming languages, a Java source program is usually not directly compiled into native code running

```
public mtApp extends Thread{
    static final N=10;
    static int sum=0;
    int data;
    private synchronized void add(){
        sum+=id;
    }
    public void run(){
        add();
    }
    mtApp(int i){data=i;}
    static public void main(String [] args){
        mtApp [] worker = new mtApp[N];
        for (int i=0; i<N; i++){
            worker[i] = new mtApp(i);
        }
        for (int i=0; i<N; i++){
            worker[i].start();
        }
        try{
            for (int i=0; i<N; i++){
                worker[i].join();
            }
        }catch(Excpetion e){}
        System.out.println("Sum="+sum);
    }
}
```

Figure 2.1: A multithreaded Java program.

on the specific hardware platform. Instead, the Java compiler will by design translate the Java source program into a machine-independent binary code called *bytecode*. The bytecode consists of a collection of class files, each corresponding to a Java class.

A JVM is used to provide the runtime environment for the execution of the

Java bytecode. Once a JVM is designed on a specific computer architecture, the computer can execute any Java program distributed in bytecode format without recompilation.

Java supports multithreading at the language level. A Java thread class is created by inheriting from the standard *java.lang.Thread* class or by implementing a *Runnable* interface. The creation of a Java thread instance from the its class is through the use of *new* operator. The execution logic for the thread is defined in its *run()* method. Figure 2.1 shows an example multithreaded Java application. It creates 10 worker threads. Each thread adds its own data to the shared variable.

Java objects can be regarded as *monitors*. Each object has a header containing a lock. A lock is acquired on entry to a synchronized method or statement block, and is released on exit. Each lock is associated with a wait queue. At Java language level, methods within a class declared with *synchronized* keyword run under control of monitors to ensure that variables in the critical sections are accessed exclusively. In the above example, the method *add* is declared as synchronized method to ensure the atomic operation on the shared variable *sum* .

The class *java.lang.Object* provides three additional methods to control the wait queue within synchronized methods or statement blocks—*wait*, *notify*, and *notifyAll*. The method *wait* causes the current thread to wait in the queue until another thread invokes the *notify* method or the *notifyAll* method; these two latter methods would wake up a single thread or all the threads waiting on this object respectively.

Each Java object consists of data and methods. The object has a pointer to a method table. The table stores the pointers to the object's methods. When a class is loaded into JVM, the class method table will be filled with pointers to the entries of the methods. When an object is created, its method table pointer will point to its class method table.

The heap is the shared memory space for Java threads to store the created objects. The heap stores all the master copies of objects. Each thread has a local working memory to keep the copies of objects loaded from the heap that it needs to access. When the thread starts execution, it operates on the data in its local working memory.

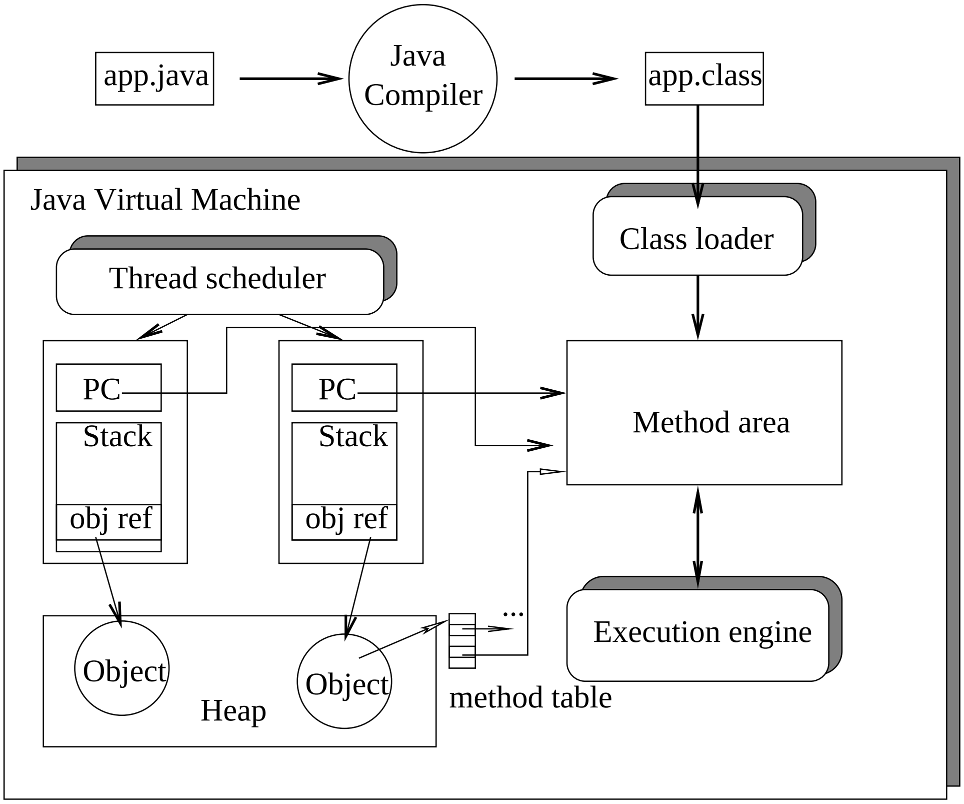


Figure 2.2: The architecture of Java Virtual Machine.

The JVM is a stack-oriented and multithreaded virtual machine. Figure 2.2 illustrates the architecture of the JVM. Inside a JVM, each thread has a runtime data structure called the Java stack that holds the local variables. The threads create Java objects in the heap and refer to the objects using object references in the Java stack. All the created objects are accessible for all the threads having the object references. The Java classes are loaded into the method area. The PC of each thread is pointed to the code inside the method area.

```
while(bc != RET ){ //loops until meets bytecode RET
      switch (bc){
      case GETFIELD: //get the field data of an object
      ...
        break;
      case AALOAD: // load data from the object array
               ...
      }
      bc = getMextBytecode(); //fetch next bytecode instruction
}
```

Figure 2.3: The interpreter loop.

The execution engine is the “processor” of JVM. The earlier JVMs execute Java bytecode by interpretation. The method entries of the object are set to a call to the interpreter with the method identification as the argument. The interpreter will create the data structures for the Java method and simulate the semantics of the bytecode instructions by operating on these data structures in a big loop as illustrated in Figure 2.3. The interpreter is usually slow. It needs to resolve the object type and field offset every time an object is accessed. It needs to lookup in the function table using the function name every time a function is called. All such interpretation overheads will largely slow down the program execution. To improve

the Java execution performance, the compiler-based approach was introduced.

2.1.2 Just-in-Time Compiler

There are two main compiler-based solutions for the Java bytecode. The first one is static compiler that compiles the whole Java source code before execution. The result of the compilation is the stand-alone binary program, which can run directly on the target platform. The approach is similar to traditional C/C++ compilers. The static bytecode compiler requires to have all the bytecodes available for the offline compilation. The Caffeine [41] bytecode-to-machine-code translator is an example of static Java compilers. Microsoft's Marmot [32] is another example of static optimizing compilers.

However, the Java's dynamic characteristic allows the program to start execution without all classes being available. This makes the static offline compilation approach not appealing for Java bytecode that requires dynamic class loading [26]. The concept of Java JIT compilation is therefore introduced. The JIT compiler, sometimes called dynamic compiler, is a compiler that produces native code from Java bytecodes during program execution. The introduction of JIT compilation has raised new challenges for the compiler designers.

A JIT compiler compiles Java methods on demand. The method pointers in the virtual method table will be set to the JIT compiler. The first time a method is called, the JIT compiler is invoked to compile the method. The method pointer then points to the compiled native code so that future calls to the method will jump to the native code directly.

As the JIT compiler translates the bytecode into native code at runtime, the overall execution time of JIT must include the JIT compilation time. Thus JIT compiler itself must be lightweight so that it can be executed in fast speed and use only small memory footprints. Researchers on JIT compiler are seeking lightweight approaches for the following technique areas.

Register allocation

Register allocation is an old topic in compiler code generation. A famous technique is to use *Graph Coloring* [3], a systematic technique for allocating registers and managing register *spills* (store the register into a memory location in order to free up a register). The priority based graph coloring requires $O(n^2)$ time and the Youktown allocator requires $O(n \log n)$ time [21].

As the JIT compiler requires that the allocation algorithm be both effective and lightweight, non-linear global register allocation algorithms, such as Graph Coloring, are seldom used in the existing JIT compilers.

A priority-based scheme for register allocation is proposed for Intel JIT compiler [2] for fast native code generation in Intel platform. In Latte JIT compiler [81], a local register allocation based on the left-edge greedy interval coloring algorithm is used for allocating registers inside a tree region.

Efficient register allocations are critical to the execution performance of Java programs. If the operand in one instruction directly uses a hardware register, the operation can be done inside CPU without the delay of fetching data from the memory cache or memory.

Eliminating dynamic dispatch and method inlining

One of the key features of OO languages like Java is to encourage programmers to reuse the programs as much as possible. The other common phenomenon is that programs written in OO languages especially Java usually use many small method invocations for good modularity. Both the dynamic dispatching of method and small method invocations will introduce large function call overheads.

For the dynamic dispatching overhead, the static resolution analysis on class hierarchy is often employed. Bacon proposed Rapid Type Analysis (RTA) algorithm for computing live procedures and live classes in for OO languages in his dissertation [11]. The subsequence optimization uses the information produced by RTA algorithm to resolve virtual function calls. However, the RTA algorithm need to have the entire program for analysis, thus it is not directly applicable to Java JIT compilers. In dynamic languages like Java that supports dynamically loading code during runtime, however, such analysis can be invalidated by later class loading and requires re-computing the analysis results.

To eliminate the small method invocation overhead, *method inlining* is often used. It tries to reduce the indirect method invocation overheads by expanding the body of a method at the place of a method invocation. In IBM's JIT compiler [67], static and nonvirtual method invocations are always inlined. For the virtual method invocations, they choose to replace the call to the target virtual method with two paths. The fast path uses the inlined code and the slow path uses the original virtual method call. The fast path is executed when it matches the actual method invoked at runtime. Otherwise when other methods are called instead, the slow path will be used.

Exception handling

The exception handling is heavily used in Java, for example, the array bound checking, null object checking, etc. For JIT compiler, the generated code must preserve the semantics of the exception handling. Besides that, people are searching for methods to eliminate unnecessary checking to improve the running performance.

The Jalapeño JIT compiler [22] used hardware interrupt to catch the null object reference and out-of-bound exception for arrays by arranging objects and arrays in a special layout.

The basic idea of eliminating the array bound check is that if it can be proved that the index is always within the correct range or that the earlier check throws an exception, the bound check of the array can be eliminated. The Intel's JIT [2] uses a simple mechanism to eliminate bound checks of indices that are constant. Gupta's algorithm [38] reduces the program execution time through elimination of checks and propagation of checks out of loops. The IBM's JIT Compiler [67] developed a new algorithm by extending Gupta's algorithm and eliminated a broader range of cases of array access.

Variable escape analysis

The *escape analysis* for variables is to determine whether an object is local to a method. If an object is local, it can be allocated on stack instead of heap to save Garbage Collection so that it can achieve good locality.

Choi et. al. [25] present a simple and efficient data flow algorithm for escape analysis of objects in Java program to determine if an object can be allocated on the stack or if an object is accessed only by a single thread during its lifetime so that synchronization on it can be removed. The technology is used in the Jalapeño JIT compiler [22].

Program hotspot detection

In the area of Java JIT compiler, to compile everything or to selectively compile execution "hotspots" are two alternative approaches. The concept of execution hotspot comes from the popular saying that ninety percent of the execution time is spent on ten percent of the code [3], i.e., the hotspot.

As the JIT compilation time is counted in the overall execution time of the whole program, it is appealing for a JIT compiler to wisely choose the execution hotspot for compilation. A famous example using selective compilation is Sun's Java HotSpot JIT compiler [61]. The philosophy behind Sun's HotSpot is "Don't optimize until you know you have a problem". In HotSpot, it will start interpretation first. Only after some profiling information is gathered, will it start to select some class methods for compilation to generate optimized native code. The method in detecting program hotspot in HotSpot simply counts the number of method invocations in the interpretive mode and sets a threshold for activating compilation.

There are other JIT compilers that compile everything before execution. For example, in OpenJIT [53], when a method is invoked, it is hooked to compile. After the compilation, the program will jump to the entry of the compiled codes. IBM's Jalapeño JVM [22] also compiles everything.

Summary

The JIT compiler is important for the performance a JVM. It is also the case in a DJVM. In DJVM, to preserve the high-performance execution in the local node, the efficiency of register allocation in JIT compilers should not be affected in generating native code that supports the distributed functions. For example, to distinguish a local object from a remote object in DJVM, the new native code generated by the JIT compiler should make efficient use of registers to avoid writing data to memory heavily. The method inlining should also be considered in DJVM if thread context is examined. The frame created by the inlined method should be recovered in the walk of stacks. The program hotspot detection can be used in DJVM for supporting the migration of computation-intensive part of a thread. Escape analysis is important for DJVM in its optimization of object access performance.

2.1.3 High-performance JVM on SMP

An *Symmetric Multiprocessing* (SMP) machine though expensive, can be a potential high-performance platform for executing multithreaded Java applications. Besides the effort to map different Java threads in JVM to different processors on an SMP machine, research in JVM has introduced several optimization techniques in supporting scalable performance on SMP servers.

The mapping of different threads to different processors in JVM usually relies on the binding of native SMP-enabled thread libraries with the JVM threading system. For example, the Sun Hotspot JVM [61] on Linux has introduced Next Generation Posix Threads library, and Native Posix Thread Library to enable fast thread scheduling on SMP machines.

The other effort is on reducing the synchronization overhead. The escape analysis technique mentioned in previous section can also be used in saving optimization overhead. This is because locking on thread-local objects can be eliminated [25]. Less synchronization will make different processors on an SMP machine be blocked less frequently.

On an SMP machine, with multiple processors sharing a memory, the memory becomes a precious resource. Efficient Garbage Collection (GC) in JVM is critical for achieving high-performance on SMP machines. GC should be incremental in such systems to avoid long pause time of the processors. A parallel incremental GC was designed by IBM for SMP servers [59]. In Sun's JDK 1.4.1, a new parallel Garbage Collector is introduced [56]. The parallel GC collector tries to parallelize and scale young generation collections by using multiple GC threads.

2.1.4 Clusters and Single-System Image

A computer cluster [5, 13, 45] is a group of computer nodes connected by a fast network such as Ethernet or Myrinet that appear as a parallel computer. Each cluster node is usually a commodity computer, such as a PC or a workstation, which has its own processor and physical memory. The nodes do not share a physical memory.

By using off-the-shelf components, a cluster provides a cost-effective alternative to a traditional supercomputer. A supercomputer tends to have large memory and I/O capacity to support the processing of large-scale technical computing problems. A single node in a cluster is limited in its computing power and memory. However, a collection of nodes can provide much larger computing power, memory, and aggregate disk and memory bandwidth. The factor that limits the cluster to be a supercomputer is the network latency. A cluster usually needs about 100–500 microseconds in delivering a single message, while a supercomputer usually takes about 10 microseconds.

The research in clusters is towards making clusters as easy to use as a single machine. A key requirement of a cluster is the Single-System Image (SSI) [42]. An SSI is the illusion that a group of computing elements appear to be a single resource. SSI can be implemented at different layers such as the hardware, the operating systems, or the middleware.

The SSI on clusters offers the a number of benefits. For example, the usage of the system resources is totally transparent to the user. The user don't need to care about the location of the computing resources. Therefore the reliability and availability for computation can be improved. When a computer node fails, the other node in the cluster can transparently take the job automatically. The SSI also simplifies the management of clusters. The operator only needs to log on to a single machine to handle the entire cluster. This can reduce the risk of operation errors.

2.1.5 Programming paradigms for parallel computing

Many programming paradigms exist for the parallelization of applications on parallel computing architectures. The three major paradigms are *data parallel*, *message passing*, and *shared memory*. In this section, we mainly introduce examples using Java to explain these paradigms.

Data parallel

The data parallel paradigm is to apply the same operation on different data sets residing on different cluster nodes.

One example language supporting the data parallel programming paradigm is the HPJava language [24]. It extends ordinary Java with some shorthand syntax for describing how arrays are to be distributed across processes. HPJava has no explicit interfaces for communication among processes. The message passing code is generated transparently by the HPJava compiler for communication.

The shortcoming of HPJava is that Java programmers need to master HPJava's specific syntax in order to exploit data parallelism and leverage the cluster computing capability. Nevertheless, due to the high portability of Java, HPJava could be favored by those who are familiar with the data parallel paradigm and are willing to try Java.

Message passing

Message passing is probably the most popular paradigm for parallel programming on clusters. In this paradigm, the programmers write explicit code to send and receive messages for the communication and coordination among processes in a parallel application. Besides the famous socket interface to support TCP/IP communication, Java programmers can also use some additional high-level message passing mechanisms such as *Message Passing Interface* (MPI), *Remote Method Invocation* (RMI) [20], and *Common Object Request Broker Architecture* (COBRA) [58].

MPI is a widely accepted interface standard for communication. One implementation of MPI on Java is the mpiJava library [12]. It enables communication in Java programs by introducing a new class called *MPI*. Using mpiJava, the programmers need to handle data communication explicitly, which is usually a complicated and error-prone task.

Java RMI is similar to remote procedure call, i.e., it enables a Java program to invoke methods of an object in another JVM. RMI applications use the client/server model. An RMI server application creates some objects, and publishes them for the remote RMI clients to invoke these objects' methods. RMI provides the mechanism by which the server and the client can communicate.

CORBA is an open, vendor-independent architecture for interfacing different applications over networks. Java also provides the Interface Description Language (IDL) to enable interaction between Java programs and CORBA-compliant distributed applications. However, COBRA is also difficult for programmers to master.

Shared memory

The shared memory paradigm assumes a shared memory space among the cooperative computation tasks. The programmer does not need to write explicit send or receive operations. The access to shared data is through the normal read or read operations.

On an SMP where the physical memory is shared by nature, the mapping of shared memory programming to the hardware is straightforward. A famous support for multithreading is the OpenMP specification [57]. OpenMP defines a set of compiler directives, together with the runtime support for shared parallel memory programming using C/C++ or Fortan on SMP machines.

Clusters, on the other hand, do not possess a physically shared memory. The communication between different nodes needs to use the network interface. To support shared memory programming on clusters, software DSM was proposed and it has been studied extensively during the past decade. Orca [14] is one object-based DSM that uses a combination of compile-time and runtime techniques to determine the placement of objects. TreadMarks is a page-based DSM [4] that adopts lazy release consistency protocols and allows multiple concurrent writers on a same page. Treadmarks uses the hardware page-faulting support, therefore it can eliminate the overhead of software checking on object status. Most of these systems support the C programming language for writing parallel programs.

The research interest in supporting shared memory programming using Java on clusters has increased. One important reason is that the multithreading feature of Java fits this paradigm well in a single-node environment. Unlike C programs that need external libraries to support multithreading, Java includes this in its language definition. This makes the multithreaded program uniform and highly portable. In a Java program, threads create their objects in the heap. All the threads in the program can then access all objects through object references to the heap. Two threads can share an object if they are both given the object reference. Java provides mechanisms to protect the critical section of accessing the same object. The shared memory paradigm is easy and convenient for normal Java programmers as they don't need to care about the explicit message sending and receiving operations. All they need is just to use the Java language to declare a critical section when accessing the shared object.

However, the current standard JVMs can only achieve limited parallelism for multithreaded programs because they can only run on a single machine, such as an SMP machine with multiple CPUs. In recent years, there is an increased research interest in building a JVM that can run on clusters. We call such a JVM Distributed JVM. A few prototypes have been proposed [8, 52, 82, 76, 7, 73, 31]. This dissertation will provide the in-depth discussion on the design, implementation and performance of the area.

2.2 DJVM

Multithreaded programming in Java has become a norm, especially for server side programming. The Java multithreading realizes a shared memory programming model where Java threads access the created objects in a heap without explicit data partitioning and message passing. Concurrent programming in Java is much easier and more natural than in most other parallel languages or runtime supports, such as Parallel Virtual Machine (PVM) [36], MPI [49], or software Distributed Shared Memory (DSM) [4, 18, 47, 64, 66], which rely on explicit data partition for different processors to achieve parallel execution.

Commercial pure Java servers are emerging in recent years to support large-scale client connections and complex service logic. Pure Java applications [54] are those that depend on core Java APIs without using any native methods and hardware platform-specific constants. Usually they are written using Java's multithreading feature, with each thread handling one client connection. The service provided is usually computation-intensive. Different threads will be loosely synchronous, i.e., they can carry out their task independently without a global synchronization. For example, the W3C's Java Web Server Jigsaw [1] is written in Java. In Jigsaw Java threads is used to handle simultaneous connections. JBoss [33] and Tomcat [34] are two multithreaded Java application servers. They use thread pool to handle the requests from their clients.

A DJVM is a cluster-wide virtual machine that supports parallel execution of a multithreaded Java application. The DJVM creates an SSI illusion for Java threads running on clusters. With the SSI created by the DJVM, a multithreaded Java application runs on a cluster just as if it were running on a single machine with improved computation power. The DJVM provides the same virtual machine services to the Java threads inside the application. Moreover, DJVM supports the scheduling of Java threads on cluster nodes and provides location transparency on object access and I/O operations for Java threads.

The analogical example is the *distributed operating systems* (DOS) [68] to support the SSI view for processes. In the DOS, such as Sprite [60] and Amoeba [69], the process can transparently access the resources of all the distributed computer nodes. The transparency means that no user involvement is needed, such as using new APIs or new names to access a remote resource. Though DOS and DJVM differs in their applications, both DOS and DJVM share some of the common issues, such as performance, transparency, flexibility, reliability, and scalability. Table 2.1 summarizes the comparison between DOS and DJVM.

	DOS	**DJVM**
Goal	SSI for multiprocesing	SSI for multithreading
Application model	Many applications on one DOS	An application on one DJVM
Scheduling unit	Process	Thread
Sharing	Explicit sharing through IPC	Default object sharing
Implementation	Kernel-level	User-level
Common design issues	Performance, transparency, flexibility, reliability, and scalability	

Table 2.1: Distributed Operating Systems versus Distributed Java Virtual Machine

DJVM research is valuable for supporting high-performance multithreaded computing. Java provides a highly portable language environment and built-in concurrent multithreading feature. A DJVM represents a more portable and more user-friendly parallel shared memory environment. It therefore inherits the merits of shared memory programming compared to the message passing programming for Java programmers.

The research on DJVM has some differences compared to that of JVM on SMP machines. In general, more focuses are on the large-scale thread scheduling and efficient distributed object sharing. It targets at the large-scale multithreaded server applications. Table 2.2 summarizes the comparisons between JVM on SMP and DJVM on clusters.

	JVM on SMP	**DJVM on clusters**
Goal	High-performance multithreading support	
Hardware	10s of processors, more $, physically shared memory (a few GB), low communication latency	100s-1000s of nodes, less $, physically distributed memory (4GB x number of nodes), high communication latency
Research focus	JIT compiler, GC, synchronization removal, thread scheduling, Java libraries	large-scale thread scheduling, efficient distributed object sharing
Applications	small-scale server applications	large-scale server applications

Table 2.2: JVM versus DJVM.

There exist a number of DJVM prototypes, including cJVM [8], JESSICA [52], Java/DSM [82], Hyperions [7], Jackal [73], JavaSplit [31], JavaParty [62], etc. Some of them are not transparent to the Java programmers and do not provide a runtime virtual machine, therefore in strict sense, they are not DJVM. But they aim at the same purpose of supporting the high-performance execution of multithreaded Java applications on clusters by distributing threads among cluster nodes. Below we try to have a closer look at the DJVM from the viewpoint of the programmers and the designers.

2.2.1 Programmer's view of DJVM

From the programmers' viewpoint, the ideal DJVM should be able to run existing multithreaded programs without any modification to the source code. The programmer therefore follows the usual manners without any new syntax or library extension. The less ideal case is to provide new APIs or libraries for the user to declare the shared code or objects that can be parallelized. In this case, the programmer needs to manually specify the distributed aspects in the program.

Besides the programming issues, the other aspect that the programmers care about is the execution mode. Static compilation, which transforms the Java program distributed in source code or bytecode format into an equivalent stand-alone parallel binary program, is one choice for DJVM. The parallel binary program is executed directly on the hardware platform. Such an approach does not realize a real DJVM. Its shortcoming is that the complete program is needed before execution. It is therefore hard to support Java's dynamic feature such as dynamic class loading. On the other hand, virtual machine support at runtime supports the dynamic features and is fully compliant with the Java langauge. The DJVM of this type consists of a collection of cooperated JVMs running on different cluster nodes. The Java program can be dynamically loaded for execution in such DJVMs.

2.2.2 Designer's view of DJVM

A JVM includes three main building blocks, i.e., the thread system, the heap and the execution engine. From the viewpoint of DJVM designer, the three major components need to be extended to enable the parallel execution of Java threads in the distributed environment.

- *Distributed thread scheduling.* The thread scheduler decides which thread to grasp the CPU and switches thread contexts inside the virtual machine. The scheduler of a DJVM requires modifications to the virtual machine's kernel. It should schedule the Java threads on cluster nodes efficiently. Efficient scheduling requires the workloads of all threads be evenly divided among the cluster nodes. Inefficient scheduling can lead to an imbalanced state across the nodes, which results in poor execution performance.

- *Distributed shared heap.* The heap is the shared memory space for Java threads to store the created objects. The physical disjointness of memory spaces among cluster nodes is in conflict with the shared memory view of Java threads. A shared memory abstraction should be reflected in a DJVM so that threads among different nodes can still share the Java objects. Efficient management of Java objects on clusters is critical to the reduction of communication overheads. Garbage collection, operations on I/O objects should be included in the distributed shared heap design.

- *Execution engine.* The execution engine is the processor of Java bytecode. To create a high-performance DJVM, it is usually necessary to have a fast execution engine. Therefore the execution of Java threads in native mode is a must. As the threads and heap are distributed, the execution engine should be able to distinguish local objects from remote objects and carry out the appropriate actions accordingly.

2.3 Summary

The technologies of Java, distributed shared memory, Single-System Image, and clusters, together form the context for the research in Distributed Java Virtual Machine. Java bytecode programs are running on top of the Java Virtual Machine. To improve the execution speed, JIT compilation is used to compile Java bytecode on demand. JIT compilation emphasizes lightweight techniques to reduce the additional overheads in dynamic compilation charged to the program execution. Java's multithreading feature provides a friendly way to write parallel programs using shared memory paradigm. SSI is the ultimate goal of clusters. DJVM must provide SSI illusion for Java programs on clusters.

Chapter 3

Related work

In this chapter, we discuss the details of several existing related DJVM projects. Finally we use the classification parameters discussed in previous section to distinguish these systems.

3.1 cJVM

cJVM [8] was developed by the IBM Haifa Research Labs. It is a cluster-aware JVM that provides SSI view of a traditional JVM for pure Java applications on cluster environments. Its purpose is to obtain improved scalability for a class of multithreaded Java server applications on clusters such as Jigsaw [1]. The pure multithreaded Java applications can be directly run on cJVM without explicit coding or pre-processing.

The cJVM prototype was implemented by modifying the Sun's reference implementation for the JDK 1.2. It consists of a collection of cooperated JVM processes on each cluster node. Each process runs in the Java interpreter loop to execute the Java threads.

When a thread is created, a load balancing function will determine the appropriate location for the thread. Then the thread is spawned on the chosen node. The scheme belongs to the initial placement scheme in distributed thread scheduling. Initial placement scheme is the action of placing a thread on one node when it is first created, and keeping the thread executing on the same node during its whole life time.

To support the access to shared objects in the application by the Java threads, cJVM uses a master-proxy model. In cJVM, when an object is created in a node, the data on the node is called the master copy of the object. Other nodes access the object via a proxy stored in their heap. Therefore all the heaps in each cJVM process will work together to provide the single monolithic heap illusion to the Java threads. The proxy and the master are using the same object layout so that the thread can access the object in a uniform way. The distinction of master and proxy is only visible to the cJVM kernel and is transparent to the Java application.

To reduce the communication overhead in accessing a remote object, cJVM uses *smart proxies* for accessing different objects. Smart proxies allow multiple implementations for different objects of a given class. In cJVM, three types of proxies are implemented, namely, the simple proxy, read-only proxy, and proxy with locally invoked stateless methods. The types of proxies applied to an object rely on the analysis done at the time of class loading. The simple proxy is the default action that transfers every operation on an object to the master. The read-only proxy assumes that the object is only modified in the constructor, and it applies every read operation locally. The proxy with locally invoked stateless methods is used for the operation of invoking a method on an object. Normally cJVM invokes a method on the master object node, which is called *method shipping*. When a method is stateless, i.e., it does not access to any field of an object, it is applied locally on the proxy object.

Our system JESSICA2 differs from cJVM in supporting JIT compilation, thread migration, and single I/O space support. In our system, the distributed shared object support in GOS uses an adaptive home protocol. It is different from the cJVM's master-proxy model since cJVM's proxy model fixes the location of the master copy of the objects.

3.2 JESSICA

JESSICA [52] was developed by the System Research Group in the Department of Computer Science of Hong Kong University. JESSICA stands for "Java-Enabled Single-System Image Computing Architecture". It is a middleware that runs on top of the standard UNIX operating system to support parallel execution of multithreaded Java applications on clusters.

Similar to cJVM, JESSICA can transparently run multithreaded Java applications written in normal fashion. The Java bytecode can be directly run on JESSICA without any pre-precessing, and the threads are automatically redistributed across the cluster to exploit real parallelism.

JESSICA supports preemptive thread migration which allows a thread to freely move between machines during its execution. The migration scheme is based on the delta execution [51]. The approach is to segment the execution context of a migrating thread into sets of consecutive machine-independent and machine-dependent frames. The migration unit is a set of machine-independent frames, known as a delta set. Once the set of frames is consumed on the remote machine, the control will return to the original node to continue the execution of the machine-dependent frames. Through delta execution, JESSICA can support the use of native methods in the multi-threaded applications, since the native methods are executed on the original nodes.

JESSICA achieves global object sharing based on the Treadmark DSM [4]. In JESSICA, the DSM system is initialized among the set of nodes involved in the execution of a multithreaded Java application. The heap objects created by the threads are allocated from the underlying page-based DSM. The Java monitors are implemented through the use of aquire/release APIs in the DSM.

The execution engine of JESSICA is the modified interpreter of Kaffe [80] Version 0.9.1. JESSICA provides independent bytecode execution on each node of the cluster. Multiple execution engines can proceed simultaneously by having each engine operating on the execution stack of one of the threads. The thread migration is implemented inside the execution engine. When the bytecode for accessing object data, such as the GETFIELD, PUTFIELD, etc., the interpreter will call the appropriate DSM functions in Treadmark to fulfill the task.

Compared to JESSICA, our new system JESSICA2 supports JIT compilation. The thread migration system in JESSICA2 introduces new technologies to extract bytecode-oriented thread context from the native execution stack. Unlike JESSICA, which realizes distributed object sharing based on page-based DSM, JESSICA2 integrates the distributed object sharing in JVM kernel and provides more optimizations to save communication costs.

3.3 Java/DSM

Java/DSM [82] is a DJVM that runs on a cluster of heterogeneous computers based on an underlying Treadmark page-based DSM. Java/DSM requires the threads in the Java program be modified to specify the location to run. This violates the transparency or SSI requirement of DJVM. For the execution, Java/DSM provides the virtual machine service to the Java bytecode without any pre-compilation.

The design was based on the JDK 1.0.2 JVM. In Java/DSM, load distribution is achieved by remote invocation of Java threads. This belongs to the initial placement scheme. Different from cJVM, the initial placement of Java/DSM is not transparent.

The DSM is used to realize the distributed shared heap and class repository for Java program. Java/DSM relies heavily on the underlying DSM to maintain the consistency of shared data. In Java/DSM, the heap is allocated using Treadmark's shared memory allocation routine. All the classes are also loaded into the Treadmark's shared memory. Since Java/DSM aims to support heterogeneous platform, it requires data conversion in the communication between different machines.

Java/DSM also uses the interpreter for its execution engine. Since the distinction of local objects and remote objects are done by the memory management module, the interpreter of the original JVM is virtually untouched.

Compared to Java/DSM, JESSICA2 supports JIT compilation. JESSICA2 does not need to manually specify the location of a thread, and is more flexible and efficient because it can distribute the threads according to the runtime system workload. The existing multithreaded Java applications can run on JESSICA2 without any modification or preprocessing. JESSICA2 uses a global object space support in JVM to support object sharing among the cluster nodes, which is different from Treadmark which uses a page-based DSM for the same purpose.

3.4 Jackal

Jackal [73] was implemented by Vrije University. It is a compiler-driven distributed shared memory implementation of the Java programming language. Jackal adopts the static compilation approach. The multithreaded Java application can be written in normal way, therefore it is friendly to the programmer like cJVM and JESSICA. However it needs the complete Java application before execution.

The front-end of Jackal accepts the unmodified Java code in bytecode or source code format. The front-end will then generate an intermediate language called Liberally Assembly Language (LASM) and feed it into the back-end compiler of Jackal. The back-end will then perform multiple code optimizations for sequential and parallel performance. The output of the Jackal back-end will be the assembly code. The assembly code is then assembled and linked into the parallel native code using the native assembler and linker respectively.

The thread management in Jackal follows the initial placement scheme. When a Java thread is created, the Jackal runtime system intercepts the events and finds an idle machine in the network in the round-robin fashion to place the threads. Once the thread is placed, it stays in the same node during its whole life time except some synchronization methods can be executed on another node.

Jackal contains a runtime system that implements a DSM protocol for variable-sized memory regions. The Jackal compiler stores Java objects in the shared regions and augments the programs with access checks. These access checks drive the memory consistency protocol.

The Jackal compiler implements several optimizations to reduce the overhead of these software access checks. The Jackal source level analyzer enables two fine-grained DSM optimizations. They are the object-graph aggregation and synchronization method migration.

By using inter-procedure data access analysis, the static Jackal front-end source compiler will detect situations where an access to some object is always followed by the access to its child objects. An object is called the *child object* if its reference is included in the fields on its *parent object*. In this case, Jackal groups the parent object and its child objects as an object graph. The object graph is then used as one unit for software access check.

The synchronization method migration combines multiple messages during the execution of a synchronized Java method. Usually, a synchronized method represents a critical section. It needs to acquire a lock, to access the shared data, and to release the lock. All these operations will need many network round trips. In Jackal, a special code will be generated for each synchronized method to execute at runtime. The code will direct the execution of the synchronized method to the node of the object that owns the lock. The live variables will be packed during the migration of the method.

As Jackal uses native code to run the application, there is no virtual machine available during its execution. Therefore the execution engine of Jackal, i.e., the static compiler, is only used during compilation time.

The major difference between Jackal and our system JESSICA2 is that we use a virtual machine at runtime so that it can support Java's dynamic features. Similar to Jackal, JESSICA2 runs Java applications in native mode. But JESSICA2 achieves this goal by using JIT compilation instead of static compilation. There is no need of preprocessing of Java programs before execution in JESSICA2. Also JESSICA2 supports thread migration.

3.5 JavaSplit

JavaSplit [31] was jointly developed by The IBM Haifa research and the Israel Institute of Technology. JavaSplit is a bytecode rewriting compiler. It is similar to static compilers such as Jackal and Hyperion. The major difference between JavaSplit and the previous static compilers is that it transforms the multithreaded Java bytecode into parallel bytecode instead of native code. The resulting bytecode can then run on commercial advanced JVMs. Therefore it can achieve high portability across existing platforms and allow each node to run the JVM in its maximum performance.

JavaSplit supports the pure Java applications. During its execution, JavaSplit provides a runtime environment to administrate a pool of worker nodes. When a multithreaded Java application is submitted, it is first transformed and combined with the runtime modules. The resulting code is then sent to one of the worker nodes that starts a JVM to run the application.

The thread management in JavaSplit uses the simple initial placement scheme. Each newly created thread is placed on one of the worker nodes according to a plug-in load balancing function.

In the design of the distributed shared heap for the distributed threads, JavaSplit integrates an object-based DSM implementing Lazy Release Consistency.

The execution engine of JavaSplit exists in its compilation phase. It is based on the Bytecode Engineering Library [28]. The engine rewrites the bytecode by linearly transforming each Java class into another class prefixed with "javasplit". For example, a class named "TestClass" will be renamed to "javasplit.TestClass". All the referenced class names are replaced with the new javasplit names. The transformation on the bytecode performs five tasks. It first replaces the creation of thread as the entry to a handler to ship the thread to a remote node. Then it replaces the bytecode instruction for synchronization, i.e., the *MONITORENTER* and *MONITOREXIT*, with the appropriate synchronization handlers. Next, it replaces the object data access bytecode such as *GETFIELD*, *PUTFIELD*, *IALOAD*, etc., with a sequence of bytecode instructions that perform the state checking for memory consistency. The following task is to augment the fields indicating the object state. The final task is to insert the DSM utility methods in the new javasplit class.

Different from JavaSplit, JESSICA2 supports the multithreaded Java application using a virtual machine at runtime and supports thread migration. The distributed object sharing is embedded in JVM, unlike JavaSplit, which insert the consistency protocol in bytecode before executing the Java application on clusters.

3.6 Others

Hyperion [7] provides a running support for multithreaded Java applications upon an object-based DSM. Hyperion takes a static compiling approach. The multithreaded Java source code is first compiled by the normal Java compiler into bytecode. Then Hyperion provides a java2c compiler to compile the bytecode into a parallel C program. Then the C program is compiled by the C compiler and is linked with the PM2 run-time library into a parallel native application. Hyperion also adopts the initial placement for thread creation and scheduling.

JavaParty [62] provides a pre-compiler and a runtime environment to support the execution of multithreaded Java applications on clusters. It is built on top of Java RMI. In the multithreaded Java program, JavaParty introduces a new keyword "remote" to indicate that a class or a thread should be distributed to remote cluster nodes. Therefore it requires the programmer to manually partition the work into local operations and remote operations. The JavaParty pre-compiler will generate new classes with Java RMI hooks for those classes declared as remote class. JavaParty uses standard JVM to run the generated new classes on clusters, with each node running a JVM. The thread is created following the initial placement scheme. The distributed shared heap is simply implemented on top of RMI following a strategy similar to sequential consistency.

3.7 Classification of existing DJVM projects

By adopting the parameters from the viewpoint of the programmers and the designers, we can classified the existing DJVM projects in Table 3.1.

From the table, we can see that most existing DJVMs aim to support the transparent execution of multithreaded Java application without introducing special APIs or syntax. The multithreaded programming is therefore in its usual manners. The exception cases are the Java/DSM and JavaParty. Java/DSM needs the manual coding of where to put the thread, while JavaParty requires the programmer to specify the local and remote classes using the "remote" keyword.

For the execution mode, two main approaches exist. The cJVM, JESSICA, and Java/DSM follow the virtual machine approach by using a group of dedicated JVMs to run the multithreaded Java application. JESSICA and cJVM provide the SSI illusion for Java applications. Java/DSM, instead,

DJVM	**Programming (new API / syntax)**	**Execution environment**	**Thread scheduling**	**Heap**	**Execution engine**
cJVM [8]	No	SSI VM	Initial placement	Built-in master-proxy model for object sharing.	Interpreter
JESSICA [52]	No	SSI VM	Thread migration	Treadmark page-based DSM.	Interpreter
Java/DSM [82]	Additional APIs	VM	Initial placement, manually specifying the location.	Treadmark page-based DSM	Interpreter
Jackal [74]	No	Static compilation, standalone parallel binary	Initial placement	Built-in OO-based DSM	Static compiler
JavaSplit [31]	No	Static compilation, VM	Initial placement	Built-in OO-based DSM	Static compiler
Hyperion [7]	No	Static compilation, standalone parallel binary	Initial placement	Linked with OO-based DSM	Static compiler
JavaParty [62]	Additional Java syntax	Pre-compilation, VM	Initial placement	RMI-based object sharing	Static pre-compilers

Table 3.1: Classification of existing DJVM projects.

does not provide such an illusion. The static compilation approach include Jackal, Hyperion, JavaParty and JavaSplit. They use new pre-compiler or static compiler to compile the multithreaded Java applications into a parallel program. The slight differences between these projects are the resulting code. Jackal and Hyperion generate the native code so that the compiled result can be directly run on the cluster. JavaParty and JavaSplit generate the Java bytecode instead. Therefore they can be executed on a group of standard JVMs running on clusters.

Most of the DJVM projects just use the simple initial placement method to schedule the remote thread. The exceptional case is our previous project JESSICA, which adopts thread migration inside the JVM. It is promising for supporting more advanced load balancing strategy.

In the heap design, many DJVMs adopt the existing DSM to provide the shared memory service. Java/DSM and JESSICA use a page-based DSM and Hyperion uses an OO-based DSM. Such a design makes the system simple but makes the optimization difficult. Some other systems implement their own distributed shared heap. cJVM adopts a master-proxy model in building its distributed shared heap. Jackal and JavaSplit implement similar services as done in a OO-based DSM in its runtime environment.

In the static compilation based DJVM, the execution engine is the static compiler, which transforms the bytecode into the parallel code. They include Jackal, JavaSplit, Hyperion and JavaParty. Others using virtual machine approach such as cJVM, JESSICA and Java/DSM mainly use the interpreter as it is simple.

3.8 Summary

In this chapter, we studied the related research work on DJVM. cJVM , Java/DSM, and JESSICA use a runtime virtual machine to realize the SSI illusion for multithreaded Java applications. Jackal, Hyperion, and JavaSplit use a static compilation approach. We compared these systems with our JESSICA2 DJVM. Classification of these projects has been provided, according to the different views of DJVM.

Chapter 4

JESSICA2 Distributed Java Virtual Machine

4.1 Our design rational

Our design will extend the JVM to be cluster-aware so that a set of cluster-aware JVMs can form a DJVM to cooperate in one single computation task. The rational is similar to that of a distributed operating systems [68]. The extension will be made at the JVM kernel level, involving the modifications of the execution engine, thread scheduler, and the heap.

4.1.1 Execution engine

A DJVM execution engine can be classified into four types: interpreter, Just-in-Time (JIT) compiler, mixed-mode execution engine, and static compiler. A Java bytecode interpreter for DJVM is relatively simple to implement. Yet it suffers from the slow Java execution in interpretative mode and thus may not be efficient enough for solving computation-intensive problems which are the main targets of a DJVM. However, several DJVMs still use it, such as the cJVM, Java/DSM and JESSICA, due to its simpleness. Static compilers, as used in Jackal [74] and Hyperion [7], although can achieve high-performance native execution of Java threads, usually miss the dynamic JVM functionalities such as loading new Java classes from remote machines during runtime. The mixed-mode execution engine, which is first introduced in Sun's hotspot compiler [61], is another possible solution to be adopted in a DJVM. But rare existing DJVM adopts such an engine due to its complex structure in JIT compilers and large overheads in switching between interpretive mode and native mode, compared to the pure JIT compilation.

Dynamic or JIT compilers are rarely considered or exploited in previous DJVM projects. The JIT compiler is simpler than the full-fledged static compilers, but it can still achieve high execution performance. Therefore in our DJVM, we adopt the JIT compiler as the execution engine.

4.1.2 Thread scheduler

In the DJVM design, two schemes are usually employed. They are *initial placement* and *thread migration*. Initial placement is to place a thread on a chosen node when it is first created. Once the thread is created on a node, its location is fixed until it dies. Thread migration is to move a thread from one node to another node during its execution. In this case, the thread context should be able to cross the machine boundaries inside a cluster. In our design for distributed thread scheduling, we combine the initial placement and thread migration together to achieve higher resource utilization and load balancing.

Many multithreaded applications usually have imbalanced workload for each thread. For example, a Java stock server application may have many threads computing different stock values and their execution time may vary significantly. To achieve load balancing for these applications, we adopt a lightweight and transparent Java thread migration mechanism to enable the threads to move from one node to another during its execution.

Our strategy is to initially place the threads evenly on different cluster nodes when threads are created. When later the workloads of nodes become imbalanced, thread migration will be used to help balance the whole system workload. In other words, the thread migration mechanism in the system helps to relocate threads after the initial placement. There are two key observations in our thread migration system. Firstly, larger stack size which results in large migration overhead. The overhead comes from the time to analyze the stack frames. Secondly, the thread migration not only affects the CPU workload, but also introduces uncertainties in the communication. For example, the data objects will be randomly located on other nodes different from the node on which the thread that needs to access them is currently running due to the migration of the thread. This makes it difficult for the design of efficient load balancing policies. In particular, we propose the migration of threads that are executing its hotspot, a computation intensive part without much distributed object access. The hotspot here is different from general program hotspot that it tries to covers the distributed nature of the Java program on DJVM. In our system, we employ simple heuristic together with the bytecode analysis to support the hotspot detection and migration.

4.1.3 Heap

When the threads are executing on different nodes, they need a distributed shared heap so that the objects created by one thread can be seen by all. The design of the heap provides distributed shared object services similar to those of a multithreaded software DSM.

There exist a number of DJVMs [52, 82] that are directly built on top of an unmodified software DSM. This approach simplifies the design and implementation of the distributed shared heap in DJVM as it only needs to call the APIs of the DSM without worrying about the memory consistency issues. For these systems, there is the issue of achieving good performance because there tends to be a mismatch between the *Java Memory Model* (JMM) and that of the underlying DSM. The page-based DSM differs from Java memory model in the access unit. Even an OO-based DSM is used, the runtime information at the JVM level cannot be easily channelled to the underlying DSM, which could lead to poor running performance.

The other approach opts for a customized built-in distributed shared heap that realizes the JMM. Because it is tightly coupled with the DJVM kernel, it is possible to make use of the runtime information inside the DJVM to reduce object access overheads. In current DJVM prototypes, cJVM and JESSICA2 use such an approach. Several different optimization techniques in reducing the communication overhead in these systems are proposed. The shortcoming of this approach is that it needs to re-design the heap, which makes the system more complex.

In our system, we go for a built-in distributed shared heap that realizes the JMM. Our approach can make use of the runtime information inside the DJVM to reduce the object access overheads as the mechanism is tightly coupled with the DJVM kernel.

4.2 System architecture

JESSICA2 provides a user friendly Java computing environment. It accepts normal multithreaded Java applications written in its usual fashion. The programmer just assumes that the program is only run on one single JVM. JESSICA2 can automatically exploit the strength of cluster to accelerate the execution of the application while preserving the SSI illusion. There is no need to pre-process or pre-compile the source code or class files for JESSICA2 to run a Java application.

Figure 4.1 shows the overall architecture of JESSICA2. The system runs on clusters and it consists of a collection of modified JVMs that run in different nodes and communicate with each other using TCP connections. As the JVM is running on top of the operating systems, it is considered as a

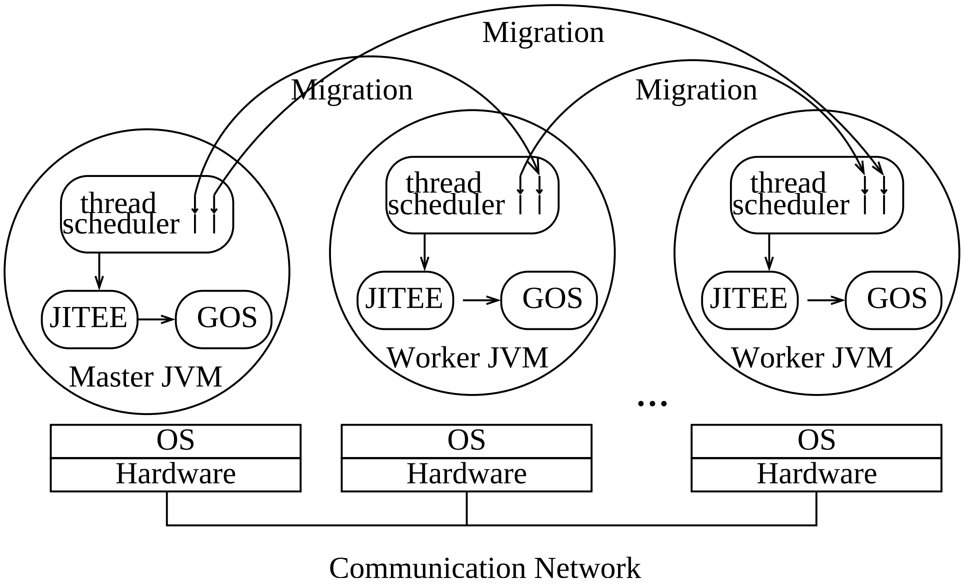


Figure 4.1: The system architecture.

middleware. Therefore our system provides the SSI view at the middleware level without any modification to the underlying operating systems.

We call a node that starts the Java program the *master node* and the JVM running on it the *master JVM*. All the other nodes in the cluster are *worker nodes*, each running a worker JVM to participate in the execution of a Java application. The worker JVMs can dynamically join the execution group. The master JVM and worker JVMs are all full-fledged JVMs. They differ only in the startup. The master JVM starts the Java application during startup, while the worker JVMs wait in a loop for accepting incoming threads from the master JVM. JESSICA2 supports running multiple threads in each JVM. Therefore the number of threads is not limited to the number of nodes in the cluster.

Inside the modified JVM, there are three main building blocks, namely, the thread scheduler, the modified execution engine called JITEE, and the Global Object Space (GOS) module.

The thread scheduler is in charge of applying the load balancing policy.

It is only through maintaining a balanced load will the system be able to achieve maximum efficiency for its applications. The policy is supported by the thread migration mechanism, which moves the context of a thread from one node to another node. Thread migration support is built in the JVM kernel, by the cooperation of threading system and the execution engine. Being transparent, the migration operation is done without explicit migration instructions to be inserted in the source program explicitly by the programmers.

The JITEE is to process the loaded Java bytecode and to compile it into native code. It is the a key component in the JVM to hide the physical cluster environment from the application. For example, by translating the bytecode MONITORENTER, which enters a critical section, to a call for distributed synchronization function, the Java thread running the code can extend its synchronization capacity from a single node to a cluster. The JITEE also generates native code to link the object access to the GOS service functions. Moreover, thread migration in JESSICA2 heavily relies on JITEE to generate supporting native code in order to extract and restore the thread context.

The GOS provides the distributed single heap to the Java threads. The GOS preservers the Java memory model for concurrent access to the data object. The detailed of the memory model will be discussed in Section 4.4. Also GOS supports the distributed synchronization and provides support for Java objects that are related to I/O operations including the file system I/O and the networking I/O.

4.3 Thread migration

To efficiently schedule Java threads on clusters, one of the desired features is to support the transparent thread migration. Thread is the fine-grained computation unit. A thread has its own context, which includes the PC, the stack, the register sets and the objects data it accesses. Thread migration is to support the movement of thread context across machine boundaries. The mechanism offers the following benefits. Firstly, the system can maintain a balanced workload by dynamically migrating threads from overloaded nodes to underloaded nodes. Secondly, it can dynamically harvest newly available nodes to join the parallel computation without the need to statically fix the number of nodes before execution. Besides, thread migration can be useful for providing fault tolerance, improving data access locality, etc [55].

Java thread migration can be realized in a non-transparent way. With modest communication APIs such as *send* and *receive*, the programmer can explicitly write the migration supporting code that packs the thread context and sends them to a remote machine. On the remote machine, the programmer can write an unpack routine to restore the execution of the thread. For example, consider a small Java code segment in Figure 4.2, which computes some function using the values of i and j, and stores the result in an object field val. When the migration happens after the second sentence, the programmer can explicitly pack the value of i and j, together with the PC. When the package is sent to a remote machine, a small routine can be used to assign the packed values to the variables and jump to the stopped PC to continue the execution.

```
// Java program example
1:  i=1;
2:  j=2;
// Migration here
3:  obj.val = compute(i,j);
```

Figure 4.2: An example Java code used for thread migration.

A static compiler can be used to automatically generate the migration supporting code at some potential migration points, e.g., the location immediately before a method invocation. Many existing Java thread migration systems follow such an idea [65, 72]. In these systems, often the Java source code or bytecode is statically analyzed and instrumented with the migration supporting code. Such code usually supports the capturing of the snapshot of the thread context and the restoration of the captured context. In this way, the thread migration becomes transparent to users. The transparency can save the programmer a lot of efforts in the work partition and data packing. Such an approach is portable since the resulted mobile code can be run on all standard JVMs. However, it fails to support the dynamic feature of Java such as dynamic class loading. The static compiler needs all the Java class files available before the instrumentation because the instrumentation can not be delayed until runtime. Also the migration supporting code needs to be inserted at every potential migration point because there is no knowledge of where to stop for migration at compilation time. Such code usually causes significant runtime overheads in both time and space when there is no thread migration. The time overhead comes from the execution of the supporting code that checks the migration event and checkpoints the execution trace. The space overhead comes from the blowup of the compiled code size after the instrumentation. The other shortcoming of such systems is that it is difficult for the Java source code or bytecode to directly manipulate the stack frames of the thread. It often needs a lengthy sequence of bytecode to simulate the operations such as capturing variable data and restoration of thread stack [72].

The Java migration mechanism can be implemented efficiently by modifying a JVM. In this way, instead of using the application code to handle the thread context, the JVM will deal with the capturing and restoration of a thread's context during runtime. Java's dynamic class loading is therefore preserved. Also the JVM can use its fast native code to support the context saving and restoration. In an interpreter-based JVM, the approach is rather straightforward [52, 63]. The JVM can delay the collection of context data until the migration happens. It only needs one simple checking in the interpreter loop of the JVM. However, interpreter-based JVMs are relatively slow.

It is therefore desirable to build the thread migration in a JIT compiler-based JVM.

In our system, we introduce two new approaches in the JIT compiler-based JVM. The first approach uses dynamic native code instrumentation in a JIT compiler that mimics the bytecode instrumentation in a static compiler. The Java methods are instrumented using optimized native code when it is first translated into native code. The other approach is to use a recompilation technique to derive the thread context from its runtime stack upon migration request without the need to insert migration supporting code in the compiled methods.

Both approaches distinguish our system from previous static compiler-based bytecode instrumentation solutions [65, 72] and other JVM-level solutions [52, 63]. It guarantees the high-performance JVM execution with JIT compilation enabled.

4.4 Memory model

The JVM specification [48] specifies the JMM semantics of memory operations issued by Java threads. A Java program assumes that there is a single heap visible to all the threads, which stores all the master copies of objects. Each thread has a local working memory to keep the copies of objects from the heap that it must access. In JIT-enabled JVM, this working memory can be regarded as the machine registers. When the thread starts execution, it operates on the data in its local working memory.

The Java memory consistency model describes how threads are allowed to see the memory update in the distributed shared heap during the interactions with other threads. The interaction is the thread synchronization. Java threads use *monitors* to synchronize the concurrent thread execution in a critical section. When entering a monitor, a lock operation is used, and an unlock operation is used to exit the monitor. The thread synchronization order determines what a thread can read from an object. In the JMM definition, two rules are used for the interaction of lock and variable access. The variable V should be considered as the object field below.

- Between a lock operation on a lock L by thread T and a subsequent access to a variable V, thread T must load the master copy of V from the heap into its working memory.
- Between a write on V and an unlock on L by thread T, thread T must write back the modification of V into the heap.

The above definition specifies how a thread interacts with the memory using lock. It does not explicitly tell how the Java threads share the objects with each other. An equivalent description of the memory model is to use the interaction of Java threads. If a thread $T1$ wants to see the previous modification of a Java object by another thread $T2$, it requires that $T2$ exit the critical section before $T1$ enters. When $T2$ exits the critical section (unlock), its modification will be reflected in the heap, and $T1$ can see the modification when it enters the critical section (lock). Figure 4.3 illustrates the Java memory consistency. To see the write on variable X by thread $T1$, thread $T2$ needs to acquire the lock, otherwise the update on X will be invisible to it.

Java's memory consistency model is similar to the release memory consistency [4]. However there exists slight difference. The locking and unlocking operations both by $T1$ and $T2$ can be done on different objects. In release memory consistency, the locking and unlocking should be done on the same object.

JMM sets a contract between the Java programmer and the JVM implementor. On one hand, JMM defines what behaviors the Java programmer should expect from the multithreaded programs. The programmer should ensure that proper synchronization be used to achieve the memory sharing in the Java program. On the other hand, the JVM implementor should meet

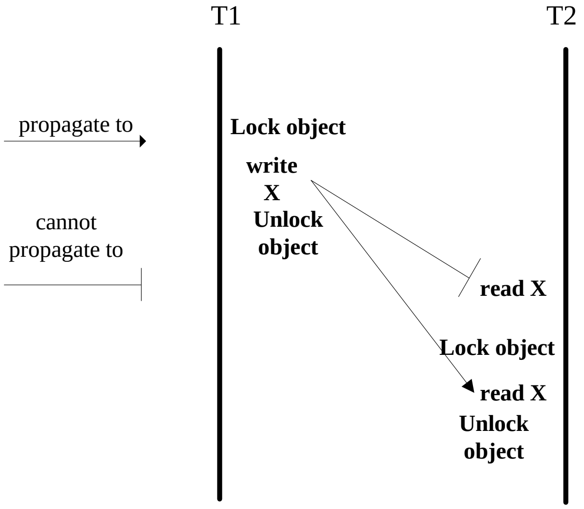


Figure 4.3: Java memory consistency.

the memory model constraints in the design. In JVM implementation, to execute the program efficiently, it usually relies on the extensive use of the fast local working memory such as hardware registers for the thread. JMM, however, requires threads to flush and reload its working memory upon synchronization.

As DJVM tries to guarantee the SSI view for Java applications, the memory model of DJVM should be equivalent to that of JVM. The key to provide JMM in DJVM is to generalize the concept of thread working memory from registers to the local memory in the cluster node.

In the software DSM-based approach to realize the distributed shared heap, the heap is initially allocated on the DSM. Later memory operations are done on the allocated memory. The heap data structure of the original JVM is barely modified.

In our system, the supporting of distributed shared heap is done by the GOS layer. The GOS layer is embedded in the DJVM and becomes an extension to JVM kernel service. It can use the JVM's threaded communication functions to transfer the shared data. Therefore the GOS is by nature multithreaded, which is superior to the DSM-based approach whose threading system usually has a gap with the JVM's threading system.

The implementation of the distributed shared heap does not need to touch the original JVM heap structure. The extension can be done only on the Java object header. By adding additional fields to support the memory consistency, the original JVM heap and its operations, such as creation of an object or accessing the object fields, can be kept unmodified.

The remote objects will be cached in a local JVM. The cached objects are also kept in the local heap just as if they were normal objects. Therefore the heap is logically divided into two areas. One is for keeping the normal objects or the master objects, and the other for keeping the cached objects. The distinction of local objects and remote objects can therefore be made via checking the object header. When a remote object is accessed, it is directed to the GOS functions, which in turn apply the memory consistency model on the access and communicate with remote JVM when necessary.

Many state-of-the-art optimizing caching protocols are adopted in our implementation. We employ an adaptive object home migration protocol to address the problem of frequent write accesses to an object remotely, and a time stamp based fetching protocol to prevent the redundant fetching of remote objects.

4.5 Summary

In this chapter, we introduce the design rational, as well as the architecture of JESSICA2 DJVM. JESSICA2 DJVM enables JIT compilation to speedup the local execution performance in native mode. Transparent thread migration is provided in JESSICA2 to support load balancing. Global object space is embedded in JESSICA2 to support efficient distributed object access.

Chapter 5

Transparent thread migration

5.1 System overview

We apply the JIT compilation technology to support thread migration. The solution using JIT compilers is able to preserve the important features of Java such as dynamic class loading. Two approaches are proposed in supporting the thread state capturing and restoration. They are the *Dynamic Native Code Instrumentation* (DNCI) and the *JIT recompilation* (JITR). DNCI instruments fine-grained native code to support thread state capturing only when a Java method is compiled by the JIT compiler during execution time. It is different from existing static compilation approaches which insert Java source code or bytecode for supporting thread state capturing before execution. JITR further eliminates the cost of instrumented code by re-running the JIT compiler to extract the bytecode-oriented thread context only at the time of thread migration. In the JIT recompilation approach, we also introduce latency hiding technique that overlaps the remote class loading and the local recompilation during the migration operation.

The thread system provides multithreading support for Java. The scheduler is extended with the capability of thread migration, i.e., scheduling thread from one node to another. The thread schedule now needs to scan its thread queue and select proper threads for migration. When a Java thread is migrated to one node, a new native thread is bound to it. The native thread will then be scheduled by the local thread scheduler.

Additional supports for thread context management are needed in the threading system. The data structures for maintaining the thread state and the thread queue, together with the thread synchronization functions, are therefore needed to be changed. The thread state includes the new thread flags, the distributed environment information, such as the source node and target node of the thread, and the native thread IDs on different nodes. These supports work together with the JIT compiler to provide a full thread migration mechanism.

5.1.1 Java thread context in presence of JIT compilers

The JVM [48] is a stack-oriented and multithreaded virtual machine. It supports the execution of simultaneous control flows, i.e., threads. Each thread has a runtime data structure called Java stack to hold a sequence of frames. A frame is pushed onto the Java stack upon a method invocation and is popped off the stack when the method returns. A frame consists of the following items: PC, the local variables in use by the current method, the operand stack serving as a work space for bytecode instructions, and the stack pointer of the operand stack.

As the JIT-enabled JVM runs the native code generated by the JIT compiler, the thread context is native, i.e., in a system-dependent form. The PC will be the real instruction pointer of the target hardware architecture. The thread stack will be native thread stack. Also hardware machine registers will be allocated to hold the values of the variables. We call such a context as a *Raw Thread Context* (RTC).

Consider the example program in Figure 4.2. Its simplified native code on i386 platform is shown in Figure 5.1. When the migration request arrives before the execution of the *compute*() method, i.e., at the first instruction, the

```
;start migration here
1:push   $0x2
2:push   $0x1
3:push   %esi     ;%esi contains the object pointer "this"
4:call   *%eax    ;assuming %eax contains the method
                 ;pointer to compute(int, int)
5:add    $0x10,%esp
6:mov    %eax,0x8(%esi)
```

Figure 5.1: Native code corresponding to the example Java program.

whole thread context at this point is scattered among the hardware registers and memory. The PC is kept in the i386 register *%eip*. The object reference "this" is kept in register *%esi* as shown in line 3. The stack variables and local variables are kept in corresponding registers and memory slots too. The situation is different from that of an interpreter-based JVM that has well-defined data structures in the memory to store the thread context.

5.1.2 Design

In our design, we seek a portable thread context as the interface to glue together independent JVMs running in different nodes. We call the context *bytecode-oriented thread context* (BTC). The BTC consists of the identification of the Java thread, followed by a sequence of frames. Each frame contains the class name, the method signature, and the activation record of the method. The activation record consists of bytecode PC, operand stack pointer, operand stack variables, and the local variables encoded in a JVM-independent format.

The BTC has the benefits of having the same portability as the bytecode. Such a design will eliminate the JVM-dependent hardware context during the migration operation. It shortens the thread context in an higher level format thus saves the communication overhead during thread migration. It also makes it easy to relocate Java objects and methods when the execution of a thread is restored on the destination JVM. Figure 5.2 shows the idea of our design.

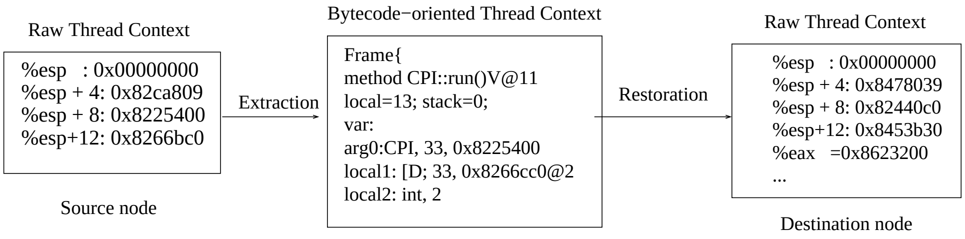


Figure 5.2: Transparent thread migration using JIT compilers.

Such a design raises two main challenges, i.e., how to transform the RTC into BTC in a JIT compiler and vice versa. In both cases, we need to make sure that the two types of thread context must be equivalent in semantics.

While transforming RTC to BTC, a thread is stopped for migration, but its native PC in RTC may not be equivalent to a bytecode PC in the BTC. Also the values of local variables in the RTC may be kept in machine registers. As the BTC does not have the equivalent register sets, the transformation needs to move the latest values of local variables from the registers to the equivalent variables in the BTC.

Another difficulty is to determine the types of native stack operands. The transformation of a value in the native operand stack needs the knowledge of its type. The native operand stack only tells that an operand value is kept in a specific stack slot, but it does not tell its type. A specific memory slot could contain an object reference, an integer value, a float value, or other types. For an object reference, we need to retrieve its type and keep the type information together with the value in the BTC so that it can be resolved on target node. Therefore we cannot simply copy all the values from the native operand stack into BTC. During the execution of a Java thread the stack operands are dynamically pushed into or popped from the thread stack. For a same stack slot, the type of its variable varies from time to time. For example, the bytecode instruction "*f2d*"(convert *float* value to *double* value) will pop off a *float* variable from operand stack and push a *double* operand on the stack top. As a result, there is no static type information for the operands stored in the native thread context. We need a way to find out the type information of the operand stack at the stopped native PC.

In the RTC-BTC transform, or less formally, capturing, we proposed the following two approaches by using the JIT compiler.

- DNCI. The Java methods are instrumented using optimized native code to support the RTC-BTC transform when it is first translated into native code by the JIT compiler.

- JITR. The RTC-BTC transform is done through the re-invocation of the JIT compilation to derive the bytecode level information.

Intuitively, the DNCI approach mimics the static code instrumentation that uses migration supporting code to save the execution trace during thread's execution. It differs from the static counterpart in using JIT compiler to generate the migration supporting code only for those methods actually being executed. It tries to amortize the cost of the thread migration operation in the thread's execution. On the other hand, the JITR approach allows the thread be executed in full speed in normal case without activating any thread migration operation. Only when there is a migration request, will the overhead be charged to the thread's execution time.

In the second transformation direction, the higher-level BTC needs to be converted to its native form, the RTC. The transformation needs to restore the frames based on the calling orders in the BTC. The native PC in each frame needs to be set according to the value of bytecode PC in the BTC. Also the register contents at the restoration point should be recovered. Our solution of the BTC-RTC transformation is based on JITR. In the BTC-RTC transform, we use recompilation techniques again to restore the native thread context. The register context is dynamically patched by code stubs in each frame that move the values in the input context into the machine registers based on the recompilation of the Java methods given in the BTC. Therefore both DNCI and JITR in RTC-BTC transformation share a common reverse transformation. Since both DNCI and JITR generate the same bytecode context, the BTR-RTC transformation works well with both DNCI and JITR in the RTC-BTC transformation.

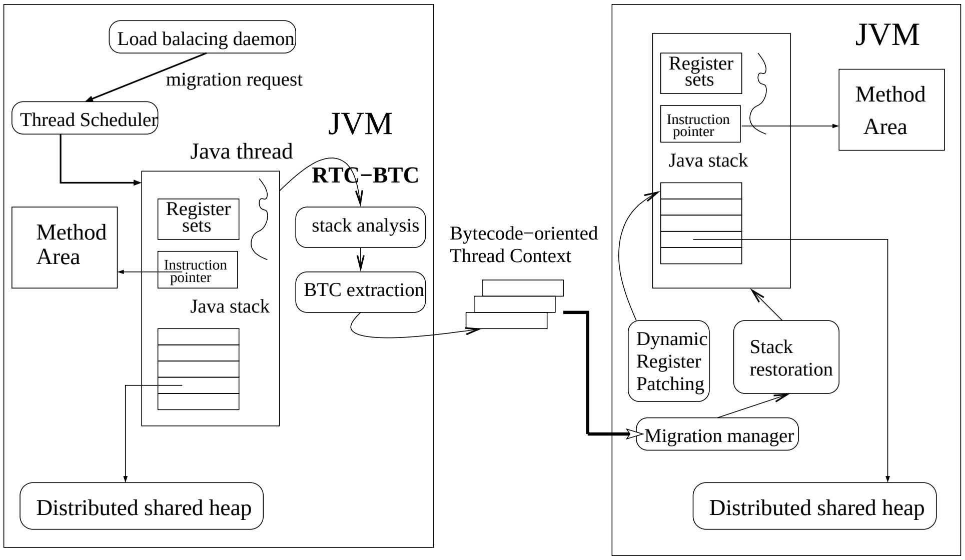


Figure 5.3: System architecutre of thread migration system.

Figure 5.3 shows the architecture of the thread migration system. Two JVMs are shown in the figure representing a source JVM (left) and a destination JVM (right) respectively.

The load balancing daemon is a thread inside the JVM. It is responsible for applying load balancing policy. When it decides to migrate one thread, the signal is sent to the thread scheduler. The thread scheduler will then select a Java thread for migration based on some heuristics such as choosing the thread with larger frame size, less object access, and longer execution time. The chosen thread will then be removed from the JVM's ready queue into a migration queue. After the selection, the RTC-BTC transformation will be performed on the thread stack. The resulted BTC is then sent to the destination JVM through TCP/IP communication. The destination JVM, when receiving the incoming thread context BTC, will spawn a new thread. The thread will load the necessary classes in the thread stack context and resolve the reference variables. Native thread stacks will be restored by the BTC-RTC transformation. After that, the migratory thread resumes its execution on the destination node.

5.2 The DNCI approach

The general idea of DNCI is to use the JIT compiler to insert supporting native code, which helps the RTC-BTC transformation, into the compiled native code. The supporting code will save the most recent information of variables in the stack during thread execution at some points, i.e., the latest values will be written back to memory from registers. When the migration request arrives, the thread scheduler can perform on-stack scanning to derive the BTC from the RTC. During this process, we emphasize simple and efficient solutions that solve the Java thread migration problem without introducing large volume of auxiliary data structures and costly or unnecessary transform functions.

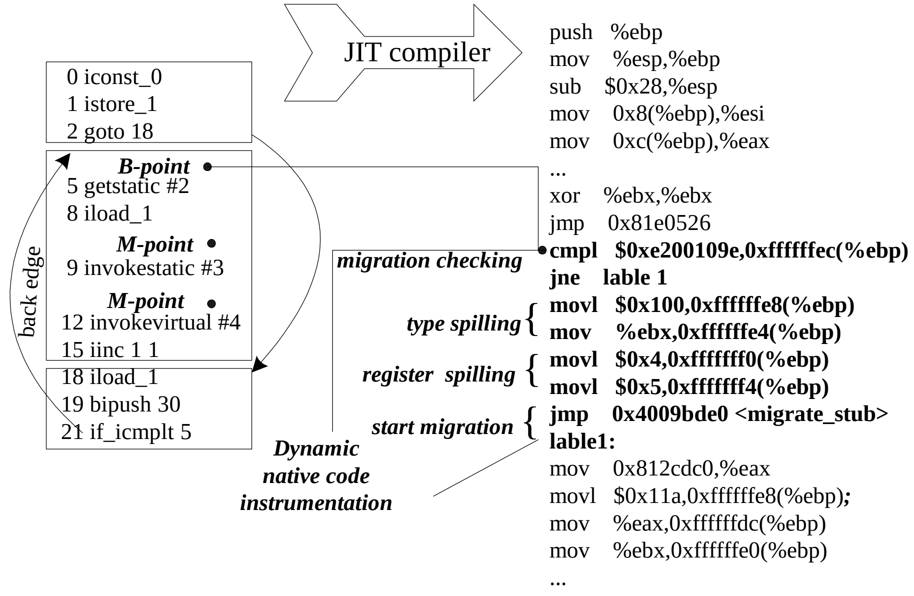
5.2.1 Migration points and pseudo-inlining

The BTC requires that the bytecode PC be well-defined so that a thread must be stopped at a point that has an equivalent bytecode PC. In other words, the stopped point should be at the bytecode boundary. However, when a thread is stopped by the scheduler and is chosen to be the migration candidate, it is most likely running at some point of native code that is not at the bytecode boundary. Since the stopped thread does not gain the control of CPU, it is hard for the scheduler to control the stopped thread to “slide” the execution of stopped thread by simulating the execution of native instructions from the stopped point to the next immediate bytecode boundary. In our system, we insert checking at some specific points in the native code of the stopped thread. Such points are called *migration points*. The BTC will be consistent with the RTC at such points, i.e., the semantics of the stack context are identical to both BTC and RTC at the migration points. When the migration request is issued by JVM, the thread will delay the acknowledgement until it reaches the next migration point.

Generally, all points at the bytecode boundary can be chosen as the migration points. However, checking at all points will degrade the execution performance dramatically. We choose two types of migration points in our system. The first type (referred to as M-point) is the site that invokes a Java method. The second type (referred to as B-point) is the beginning of a bytecode basic block pointed by a back edge, which is usually the header of a loop. The concepts of migration points and dynamic code instrumentation are illustrated in Figure 5.4. At the migration points, the register spilling is to save the variable data from the registers to memory. This is needed as the native code generated by JIT compiler makes use of the registers to hold the variables. As stated in Section 5.1.2, type information of the stack variables is needed for extracting the stack variable values from the memory. Type spilling, which saves the types into memory, is introduced and it will be discussed in detail in next section.

The insertion of M-point is necessary because we need to make sure that a frame should have a consistent BTC before it is pushed in the stack so that later capturing can get the correct BTC from the pushed stack frame. At such points, we need to spill the values and types of variables, bytecode PC and stack pointer to the memory slots in the thread stack. We also have one test instruction to check if the migration request is issued.

The insertion of M-point will add overheads to the thread execution and too many migration points inserted will lead to a performance degradation caused by the blowup in code size. We observe that skipping migration check-



```
0 iconst_0
1 istore_1
2 goto 18
5 getstatic #2
8 iload_1
9 invokestatic #3
12 invokevirtual #4
15 iinc 1 1
18 iload_1
19 bipush 30
21 if_icmplt 5
```

```
push   %ebp
mov    %esp,%ebp
sub    $0x28,%esp
mov    0x8(%ebp),%esi
mov    0xc(%ebp),%eax
...
xor    %ebx,%ebx
jmp    0x81e0526
cmpl   $0xe200109e,0xffffffec(%ebp)
jne    lable 1
movl   $0x100,0xffffffe8(%ebp)
mov    %ebx,0xffffffe4(%ebp)
movl   $0x4,0xfffffff0(%ebp)
movl   $0x5,0xfffffff4(%ebp)
jmp    0x4009bde0 <migrate_stub>
lable1:
mov    0x812cdc0,%eax
movl   $0x11a,0xffffffe8(%ebp);
mov    %eax,0xffffffdc(%ebp)
mov    %ebx,0xffffffe0(%ebp)
...
```

Figure 5.4: Dynamic native code instrumentation.

ing in short methods will not delay the migration response time too much. Therefore we make the following decisions in our system: We treat Java library method invocations, which usually last for a relatively short time, as "straight" code sequences, i.e., no migration points will be inserted before such method invocations. Nevertheless, the advantage of such a decision is that the context will become more portable as the context contains only application methods. Moreover, such a decision can be generalized to inlined methods which are typically tiny. As migration will not happen inside an inlined methods, no additional efforts are needed to transform an inlined stack to a normal stack, unlike the deoptimization technique [40] which supports the user's request to breakpoint at inlined methods.

The B-point is selected to make a thread be able to respond to the migration request in a reasonable time when it is running inside a loop. However, if we follow exactly the same techniques used in M-point, it will be much costly for JIT compilers to perform so many memory operations at each iteration in a loop. We observed that no spilling is needed if no migration request is issued. At the B-point, we check the migration request in first native instruction. If no request happens at the migration point, no spilling will be performed. Therefore during normal execution, each iteration in a loop needs only one additional flag checking. Note that M-point can not have such optimization, because not saving the most recent data in all the previous frames in the stack context will result in incorrect stack capturing. For example, consider the stack frames in Figure 5.5. The local variable i of method $a()$ is kept in register %ecx. If we do not synchronize the value of i before entering method $b()$, when a migration request arrives during the execution of method $b()$, the value i stored in frame created by method $a()$ will be incorrect.

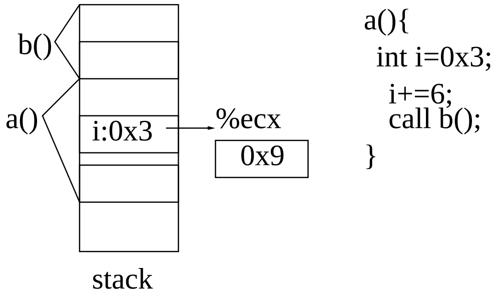


Figure 5.5: An example of stack frames.

As Java applications typically have many small-sized methods, if a JIT compiler has inlining optimization, the migration checking can be eliminated dramatically as many M-points will be eliminated. The threshold of the size can be configured by the user before execution. For a JIT compiler that does not introduce method inlining optimization, we propose a *pseudo-inlining* technique to eliminate the checking overheads with the same effect as inlining optimization. “Pseudo” means that the method is not actually inlined by the compiler. Rather our M-point checking treats it as if it was inlined (see Figure 5.6). A small-sized method is considered as an pseudo-

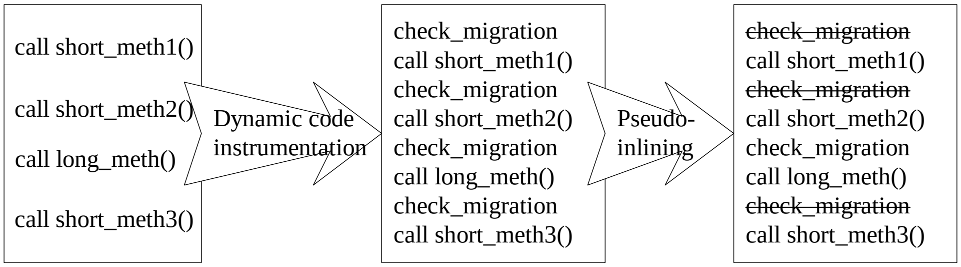


Figure 5.6: An example of Pseudo-inlining.

inlined candidate if the method contains no further method invocations. We will not place any checking and spilling if the callee is an inlined candidate at M-points. Also, no B-point checking will be inserted in an inlined candidate.

5.2.2 Type spilling

The thread context includes the values of stack operands together with their types. In Sumatra [63], it is proposed to use a separated type stack operating synchronously in the JVM interpreter during thread execution, so that at the time of migration the operand type can be known. Although such a method can be used in the case of JIT compilers, it doubles the operation time to access the stack operand.

During dynamic native code instrumentation, we choose to perform the type spilling at the migration points. The type information of stack operands at migration points will be gathered at the time of bytecode verification before compiling the Java methods. We use one single type to encode the reference type of the stack operand as we can deduce the real type of a Java object from the object header. We choose one encoding for each of primitive types. Therefore, we can compress one type into 4-bit data. Eight compressed types will be bound in a word, and an instruction to store this 32-bit machine word will be generated at the migration points to spill the information to appropriate location in the current method frame. Figure 5.7 shows an example of saving the types of eight stack operands into current

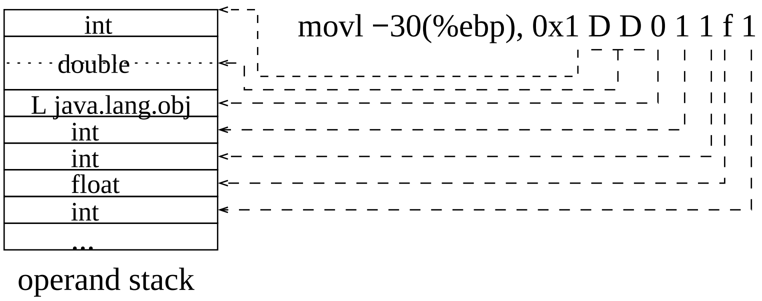


Figure 5.7: A type spilling example.

method frame. We use “1” to for the encoding of integer type including *byte*, *char*, *short*, *int*, “D” for *double*, “F” for *float*, “2” for *long*, and “0”(zero) for object reference. For typical Java methods, only a few instructions are needed to spill the type information of stack operands in a method, which results in better performance improvement than the synchronous type stack method used in Sumatra.

5.3 The JITR approach

Another approach for the RTC-BTC transformation is to use JIT recompilation. This section describes our experiences in the JIT recompilation technique. Using this approach, the normal execution of Java thread will not run any redundant migration supporting code as in the case of the previously discussed DNCI approach. The capturing operation will happen only when there is a migration request. The general idea of JIT recompilation is to playback the compilation of the stack frames and collect the BTC information from the JIT compiler.

Figure 5.8 illustrates the detailed steps of the JIT recompilation. The shadeless box represents the operations in the source JVM while the shaded box represents the operations in the destination JVM.

The JIT recompilation consists of seven steps: the stack walk, frame seg-

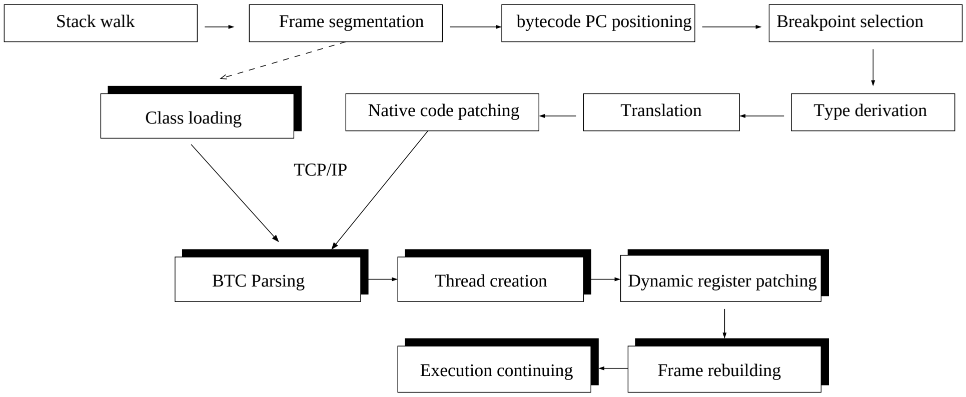


Figure 5.8: Transparent Java thread migration using JIT recompilation.

mentation, bytecode PC positioning, breakpoint selection, type derivation, translation, and native code patching. We will elaborate them in details in the following.

5.3.1 Stack walk

The stack walk is to traverse the native stack of the thread to be migrated and collect the information of each frame into a frame link list. The information includes the *frame pointer* (FP), the *stack pointer* (SP), the saved native PC, and the address of stack slot storing the saved native PC. The information will be used for later steps. During the walk, all the native frames including those used by the JVM internal functions or the signal handlers will be collected. Normally, a native frame is linked by its FP saved at the beginning of a frame. We can then traverse the frame by following the FP pointer. However, special handling is needed when there exist signal handlers inside the stack. In Linux, the return address of the signal handler points to a code segment on the stack. In the stack walk, we need to identify such a signal handler by matching the native code segment and retrieve the original PC from *sigcontext* parameter of the signal handler.

5.3.2 Frame segmentation

The frame segmentation identifies the frames created by pure Java methods. We call them *Java-frames*. The other frames created by the JVM internal functions or Java native methods are called *C-frames*. The topmost consecutive Java-frames will be chosen for migration. Normally the frame layout of a typical Java thread will be “C-frames, Java-frames, and C-frames”. The topmost C-frames include the scheduling function and other JVM internal functions. The followed Java-frames are the main body of the thread. The bottom C-frames include some initialization functions of the JVM threading system.

The identification of Java-frames is done by matching the native PC with the code range in the Java method cache, which stores the native code information of the compiled methods. When the native PC is found in one Java method’s native code range, the Java method is identified. After the consecutive Java-frames are identified, a filter function will be applied on them to mark those frames to be migrated. The filter allows only a part of the frames be migrated. For example, only the frames created by the methods that contain computation loops will be chosen. The default setting is to let all the Java-frames be migrated.

The frame segmentation result is used by the thread scheduler to select an appropriate thread for migration. Upon a migration request, the scheduler will walk the stacks of all threads, and segment their frames. Simple heuristic is used to detect the threads with a hotspot in the Java frames, by counting the loop and the number of object access in the stack frames. Frames with small number of object access and large number of loops will be considered as hotspot. The thread with largest hotspot can be the candidates for migration.

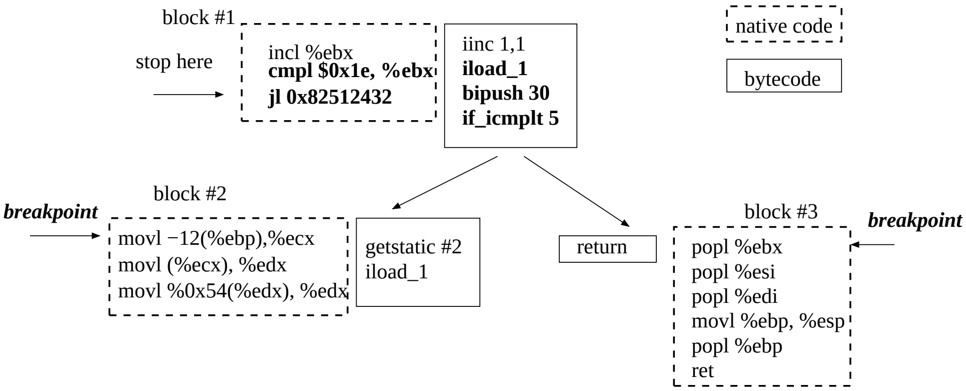


Figure 5.9: An example of native code PC positioning and breakpoint selection.

5.3.3 Bytecode PC positioning

The third step is to position the bytecode PC on the selected Java-frames. The reason is that only when the bytecode PC is known can we get the other information such as the operand stack size and the variable types in the operand stack.

When the Java bytecode is compiled into native code, one bytecode instruction may correspond to a few native code instructions. There is no simple one-to-one mapping of a bytecode instruction to a native code instruction. When the JIT compiler performs code optimization, the mapping becomes many-to-many, i.e., several consecutive bytecode instructions will match a sequence of native code instructions. In other words, a bytecode block will match a native code block.

For example, in Figure 5.9, the two native code instructions in bold face in the first native code box, i.e., "cmpl $0x1e, %ebx" and "jl 0x82512432", correspond to three bytecode instructions, "iload_1", "bipush 30" and "if_icmplt 5". When the thread is stopped at the instruction "jl 0x82512432" in the native code block, the native code PC does not match one single bytecode PC in the bytecode block.

One approach to solve the positioning problem is to save mapping when the method is first compiled by the JIT compiler. However it needs to consume much memory because all Java methods compiled need the storage even if they are not involved in a migration act. To fit our purpose of full-speed normal execution, we delay the extraction of this mapping until the method is used in the migration frames.

When a thread is stopped at a native code PC, often it is unlikely to hit the head boundary of the native code block to which the current PC belongs. We therefore re-run the JIT compilation on the method, and save the mapping at the offset of current native code PC during the native code generation phase.

5.3.4 Breakpoint selection

At the stopped native code PC, the RTC may not have a corresponding BTC. We need to delay the RTC-to-BTC transformation to the next consistent native code PC, i.e., the next native code PC at which the RTC has the corresponding BTC. The transformation will be carried out by setting breakpoint at the next consistent position. In normal case, the native code PC that maps to the next bytecode PC will be the breakpoint target. But there may be more than one target position in some cases. For example, in Figure 5.9, there are two target native code positions caused by the conditional jump instructions. For the compound branch bytecode instructions *TABLESWITCH* and *LOOKUPSWITCH*, which behave like the *switch statement* in C, there may have many jump targets. All these target bytecode PCs will be collected. After that, the type derivation and migration supporting native code will be generated at these points.

5.3.5 Type derivation

At the breakpoint targets, we need to derive the types of local and stack variables. Unlike our type spilling method used in DNCI, which saves the variable types at some migration points by the generated native code, we derive the type information by simulating the simplified bytecode verification on the Java method. In this step, no real verification on the type consistency will be carried out since the method has been verified to be correct before. Instead only the type information and stack operation will be updated along the verification.

According to the JVM specification [48], at any given point in the program, the operand stack is always of the same size and contains the same types of values no matter what code path is taken to reach it. Therefore we don't need to follow exactly the execution path to reach the breakpoint targets. We use breadth-first-search in the control graph of the bytecode program to reach the breakpoint targets. Once a breakpoint target is reached, the type information of the local and stack variables will be saved. Using the information, we are able to proceed to the next step to generate the migration supporting native code.

5.3.6 Translation

In this step, we run the JIT compilation code generator to generate the new native code for the current method. For all the locations that are not marked as breakpoints, the same native code will be generated as the original compiled result. The new native code will include the supporting code at the breakpoint targets.

At each breakpoint, native instructions that save the bytecode PC, the Java stack pointer, the operand types and values into the thread's private area will be generated. Except the bottom frame, the native instructions will be generated to simulate the epilogue of the current method. The instructions will unwind the thread stack and return the control to the caller. The bottom frame, instead, will return to the migration control function, which will then pack all the thread context and send them to a destination JVM.

5.3.7 Native code patching

After the new code with breakpoints has been generated, the thread's native stack will be patched so that when the thread is scheduled to run again, its execution will base on the new native code for each frame. Here we have a simple requirement on a JIT compiler that it should be is repeatable, i.e., it can re-generate the same instruction operator, same addressing format, at the same offset from the entry of the method, for a same Java method. Based on the original stopped native PC offset, we can get the new native PC.

Recall that we have collected the stack address storing the return address in each frame during the stack walk. The native code patching step then replaces all the native return addresses in the stack frames with the corresponding new native PCs. Hence, when the thread is re-scheduled to run again, the execution will go through the new generated native code. Eventually when the thread reaches one of the breakpoints, the migration handler will start. BTC will be collected by the handler and sent to the destination JVM.

5.3.8 Migration latency hiding

Though our JIT recompilation scheme does not introduce overheads when there is no migration request, the overheads will occur at the time of migration. The overheads include the recompilation of all the methods inside the selected frames. On the other hand, when restoring the thread context, the destination JVM needs to load the Java classes from the local disk or over the network. From our experiments, the class loading step could be the dominant overhead in restoration phase.

We observed that once the frames in the thread context have been chosen to be migrated, we can immediately send the Java class files needed in these frames to the destination JVM. As shown in Figure 5.8, the dashed line from the frame segmentation box to the class loading box represents the transmission of the class files to the destination JVM. After that, while the source JVM is performing the JIT recompilation, the destination JVM can pre-load all the class files into the method area. As a result, the latency of a migration can be hidden by the overlapped operations of the source JVM and destination JVM.

The migration latency hiding approach is not necessary for the DNCI approach because its capturing operation is rather lightweight.

5.4 BTC-to-RTC transformation

The restoration of the thread is done through the BTC-to-RTC transformation. The destination JVM, after accepting the thread context BTC in the JVM-independent text format, will run a parser to interpret the BTC into an internal data structure for later processing.

Next, a new native thread will be created by the migration manager and the created data structure will be assigned to it. The newly created thread becomes the clone of the migrated thread in the destination JVM. The clone thread will start the bootstrapping.

The clone thread then builds a sequence of stack frames with the return addresses and the frame pointers properly linked, according to the call orders in the input BTC. The recompilation of the frames in the destination JVM will be carried out. The technique "dynamic register patching" to rebuild register context just before the control returns to the restored points is shown in Figure 5.10. In the figure, shaded areas represent the native codes. "Ret Addr" is the return address of the current function call and "%ebp" is the i386 frame pointer. The dynamic register patching module will generate a small code stub using the register-variable mapping information at the restored point of each method invocation. The thread execution will switch to the code stub entry point for each method invocation. The last instruction to be executed in the code stub will be a branching instruction to jump to the restored point of the method. To make our solution efficient, we allocate the code stub inside the thread stack so that when the stub jumps to the restored point, the code stub will be automatically freed to avoid memory fragmentation caused by the small-sized code stub.

A trampoline function will then be used to swap the current stack frame with the newly created stack frames. It also makes sure that upon completion the thread will return the control to the closing handling function. The closing handling function will collect the return data and notify the source JVM. Then the thread can terminate its migration journey.

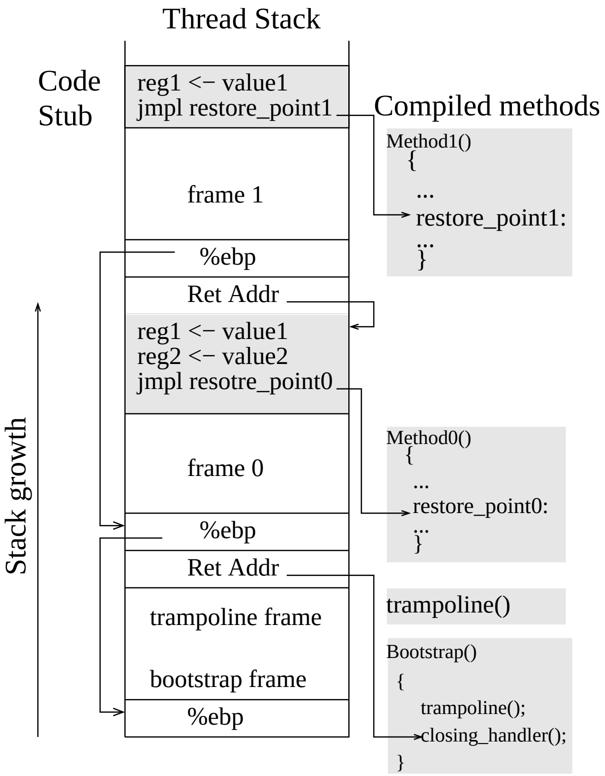


Figure 5.10: An example of dynamic register patching on i386 architecture.

5.5 Load balancing policy based on thread migration

To support the distributed thread scheduling based on thread migration mechanism, the scheduler will need to have the knowledge of system workload. This is done by creating a daemon thread for collecting the workload information. Then the load balancing policy is enforced by the distributed thread scheduler.

The load balancing policy adopts a scheme similar to the work stealing [19]. A lightly loaded JVM will try to acquire computation threads from other heavily loaded nodes periodically. The load information uses the centralized strategy to store the CPU and memory usages on the master node. The master node will periodically process the workload information based on a predefined time interval. All the worker JVMs do not directly contact each other for the exchange of workload information to save bandwidth. Instead, the lightly loaded node will post its advertisement on the master node while the heavily loaded node will try to acquire the information from the master node. The subsequent thread migration will be negotiated between the lightly loaded node and the heavily loaded node.

The worker JVM maintains its own workload by querying the CPU and memory usage in local /proc file system. The state transition in a worker JVM between heavy load and light load resembles the way of charging and discharging the electricity capacity. In the charging phase, the JVM will go in the direction of acquiring threads until some threshold is met. The threshold is determined in the following way. First we average the values of CPU usage, memory usage and the number of active threads over all nodes, and we calculate the deviation of the workload. Then, the threshold will be set to the value at the position configurable by the user, say top 20%, based on the curve of the load distribution. When current node's value is greater than the threshold for two consecutive times, it will switch the state to heavy load. Then the discharging begins by migrating threads to lightly loaded nodes. The change from light load from heavy load is vice versa.

The master node will not be a bottleneck caused by the load information because only those worker nodes that have radical load changes (from heavy load state to light load state or vice versa) will send the messages to it. The radical change is detected by monitoring its active thread number, CPU usage and memory usage in the worker node locally. For example, if active thread number has been decreased due to the completion of a thread, the event can be regarded as a radical change. Moreover, as the load information is piggybacked in the package that exchanges object data between the master node and the worker nodes, the centralized load balancing strategy can scale, even in a cluster with a few hundred nodes.

5.6 Summary

In this chapter, we present the new use of JIT compilers in the JVM to support transparent Java thread migration on clusters. The thread migration system uses a portable interface, the bytecode-oriented thread context, for the movement of Java thread context. Our solution preserves high-performance JIT compilation execution in the presence of thread migration.

Two approaches are proposed in supporting the transformation between the raw thread context and the bytecode-oriented thread context, i.e., the dynamic native code instrumentation and the JIT recompilation. The dynamic native code instrumentation is different from existing static compilation approaches in that it instruments fine-grained native code on demand at runtime so that it is able to preserve the important features of Java such as dynamic class loading.

The JIT recompilation steps further and can eliminate the cost of instrumented code by re-run the JIT compiler to extract the bytecode-oriented thread context at the time of migration. In the JIT recompilation approach, we also introduce latency hiding technique that overlaps the remote class loading and the local recompilation during the migration operation.

The use of JIT compilers in thread migration has taken advantages of the existence of compilers during runtime in a JVM. It therefore provides more flexible support in thread migration than other static compiler based approach because in those environments, it is unlikely to have the whole static compiler embedded in the runtime environment. The JIT recompilation approach can be generalized to support many other useful applications such as debugging, profiling, and checkpointing without wasting spaces in storing the compilation information in advance.

Chapter 6

Global object space

6.1 System architecture of GOS

The goal of GOS is to provide location transparent object access services for the Java threads in the DJVM. Since Java objects in JVM are allocated in the heap, the GOS is built based on the design of the heap. Figure 6.1 shows the architecture of GOS. The heap in each JVM will be divided into two areas logically, namely the *master heap area* and the *cache heap area*. The master heap area is used to store the original Java objects like the unmodified JVM heap. Virtually all the master objects in all the JVM heaps together form the distributed shared heap for JESSICA2. The node that holds the master copy is called the home of the object.

Caching a remote master object in a JVM is adopted to reduce unnecessary network traffics in fetching the master object. The cache heap area in the heap is used for allocating cached objects. The cached object is similar to the master object except that it has different flags in the object header. The operations to a remote object in a thread is then performed in the cached object if the master object is resided on a remote node. The working memory of a thread is then divided into two levels. The first-level working memory is the same as the working memory in a single-node JMM. The second-level

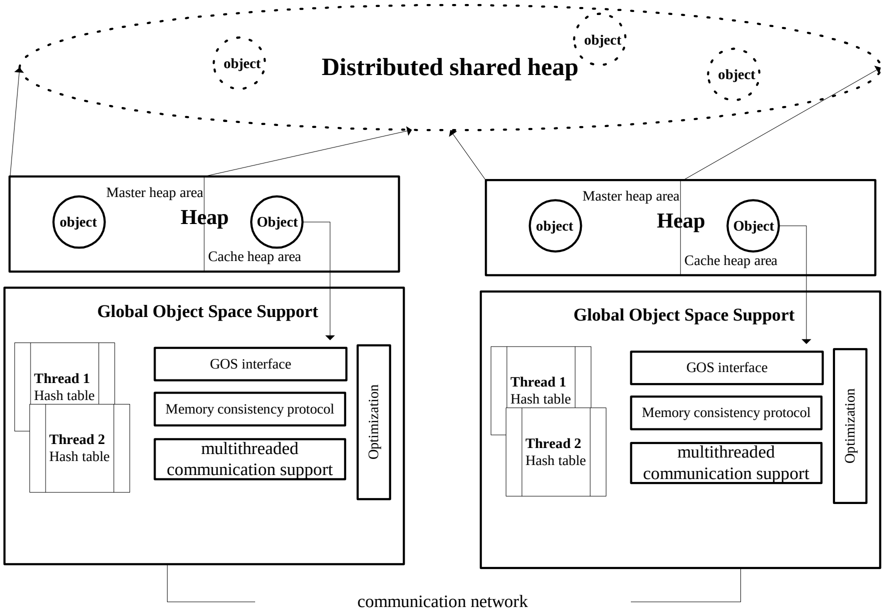


Figure 6.1: The architecture of Global Object Space

working memory corresponds to the cache heap area. A per-thread hash table is used in GOS to support the enforcement of memory consistency of the cached objects and provides quick lookup service for a cached object.

The GOS support is tightly coupled with the JVM kernel. A group of GOS interfaces are provided to support the access to a cached object by the Java threads. Such interfaces will be linked to the thread in the code generation phase. Below the interfaces, GOS provides the memory consistency protocol layer to enforce the JMM. In the design, we adopt an adaptive object home migration protocol. To support the protocol, a number of service functions for memory consistency together with appropriate data structures are provided. The data packages generated by the GOS protocol use a portable format as the representation of the Java object data or class data. Each field in an object uses its index in the object fields as its unique id. The id is more portable than using the memory offset. The unique field id is followed by the data type and the object data. The data packed will be transmitted using the VM's threaded TCP communication functions. Since each Java thread will communicate with a remote JVM using the connection-oriented TCP communication, we cache opened TCP connections to support persistent connections to avoid the TCP startup overheads.

We adopt many state-of-the-art optimization techniques that are used in traditional software DSM systems along all the layers of the GOS. These optimization techniques include object pushing, fast software state checking by exploiting the JVM runtime information and JIT compiler techniques in the implementation of the GOS. Object pushing is a kind of pre-fetching technique that exploits the connectivity of Java objects. Based on the internal field definition of an object, we can aggregate the communication messages to transfer several objects in one single message. The fast software checking is to use the JIT compiler to generate native code for the object state checking instead of simply directing it to the GOS interface functions.

The following sections will elaborate the details of GOS, including the data structures, the protocol layer, and the optimization.

6.2 Memory consistency model and GOS data structures

Release consistency is the most widely accepted relaxed consistency model in the research of software DSM to reduce the remote memory access overheads. Release consistency uses three operations to structure the remote memory access, i.e., ordinary memory access (read or write), *acquire* and *release*. Intuitively, the acquire operation is to gain the exclusive access right of the shared data, while the release operation is to relinquish the control. The acquire and release operations by different processors are linked in the following way. The acquire operation is used when a processor begins to access the data that may depend on other processors and it requires that the update from previous processors be seen by the acquiring processor. When a processor succeeds in the acquire operation, it can perform normal read and write operations on the variables. After the current processor finishes its access to the data, the *release* operation is used and it requires that the current processor's write updates be accessible by any processor issuing a subsequent acquire. The home-based release consistency model is introduced based on the release consistency model to reduce communication traffic in exchanging the updated data, by maintaining a home for each page or object. All the up-to-date data of a page or object can be reached from its home.

JMM is similar to home-based lazy release memory consistency model [43]. It is natural to design of the protocol of GOS using the home-based concept. In GOS design, for each O, there will be only one master copy among all the cluster nodes. For a Java object O, the node that holds its master copy is called the *home* of the object, denoted by $HOME(O)$. A Java object in GOS is then uniquely identified by the id of $HOME(O)$ and the address of its master copy. In other words, the pair ($HOME(O)$, *addr*) identifies a Java object, where *addr* is the address of the object in $HOME(O)$.

Objects can be cached on other nodes, therefore we use a uniform header for both master objects and cached objects. We extend the original Java object header in JVM by adding a cache pointer at the original object header to distinguish a cached object and a master object. Figure 6.2 shows the data structure for Java object header in GOS. For a cached object, the cache pointer will point to a shared cache header for all local threads that need to cache the data from the home of the object. The cache header stores the location information of the master object. The location information includes the object id (the pair of id of the master host and the address of the master object), the type of the object and a link list pointer. We also associate a time stamp counter in the object header.

Unlike some implementations such as Hyperion [7], which use a single cached copy for all the Java threads, we use different cached copies for different local threads. Firstly it is closer to the JMM that each thread has its own working memory. Secondly it will prevent different threads from interfering each other's cached copies. For example, in the case of sharing a cached copy by all local threads, if a thread enters a lock and tries to flush all the cache data, it may flush the data that are still valid for other threads. Thirdly, such a decision can lead to more fine-grained caching, as we have done for huge array object. Each thread can cache the portion of a huge array which it is interested in.

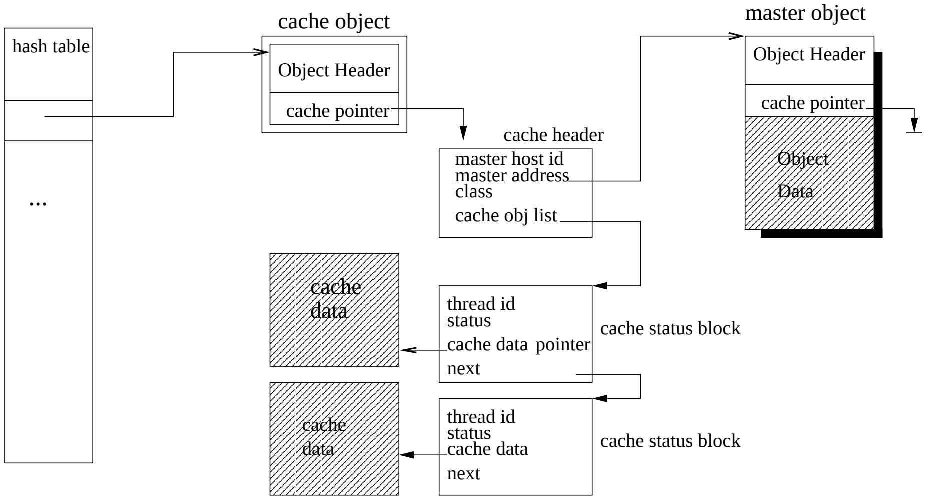


Figure 6.2: Java object header in GOS.

Another important data structure in GOS is the hash table. The hash table is used to record the cached object headers for a Java thread. Whenever a remote object is cached in a local JVM, a cached object will be created for the object as well as the cache header. The cache header will be kept in the hash table. The objects having different home nodes will be kept in different hash tables. Therefore when a cached object is searched, it will first locate the right hash table using the id of its home. The hast table will be scanned when applying the memory consistency protocol.

6.3 Adaptive object home migration protocol

6.3.1 Protocol description

When the GOS interface functions are called, they will call the protocol function in GOS to enforce the JMM model. In JESSICA2, we propose an adaptive object home migration protocol based on the JMM model.

The location of a thread T is denoted by $LOC(T)$. If $LOC(T)$ is equal $HOME(O)$, its access to object O will follow the same rules in the single-node JMM model, i.e., the access to O by T will go through the first-level working memory and then the heap. Otherwise, the access of O by T will go through three layers, first from the $HOME(O)$ to the cache, then from the cache to the first-level working memory of T.

According to the semantics of JMM, when a thread enters a monitor, it should see the previous update on the objects that it will access. When the home of an object is not resided on the same node in the DJVM, a consistency protocol must be used to guarantee the access of correct data. The consistency protocol can be realized by flushing and fetching the object from its home. The flush operation is to invalidate all the cached objects previously accessed by the thread. The fetching operation is to get the master copy from the home node. In this way, when the thread entering the monitor accesses a cached object, it will discover that the cached object is invalid and it will call the protocol service function to fetch the data from the home node. The flush and fetching operation therefore guarantees the JMM semantics.

When an object is intensively accessed by a remote thread, flushing and fetching the object data from the master node can result in significant communication cost, which in turn slows down the overall system performance.

To address such a problem, we introduce the *adaptive object home migration protocol* in the GOS design. Adaptive object home migration means that we dynamically change $HOME(O)$ to another node. We choose simple heuristic method to adaptively update $HOME(O)$: if the number of the write accesses from a thread T dominates the total number of accesses to O by all threads, $HOME(O)$ will be set to $LOC(T)$. After the change, if O is frequently accessed by the threads on the new $HOME(O)$ in a period of time, the communication cost to flush and to re-fetch O will be eliminated since the accessing threads are in the same node as the master objects. This could result in great message reduction during the execution of Java threads in the DJVM.

To avoid the unnecessary fetching of O if C contains up-to-date data, we use a time-stamp based validation protocol. We associate O with a time stamp counter denoted by $O.ts$. Each time O is updated $O.ts$ is increased by one. C will keep the latest $O.ts$ received from $HOME(O)$ in $C.ts$. If $C.ts = O.ts$, C will not be invalidated upon flushing.

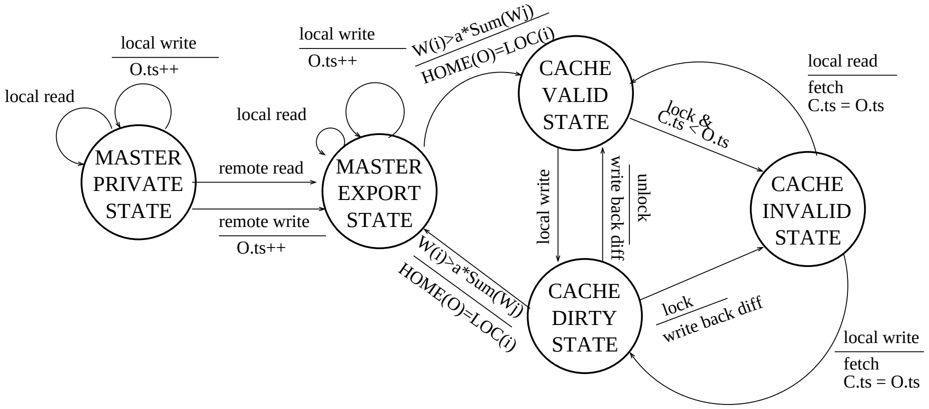


Figure 6.3: Object state transition in GOS.

Figure 6.3 shows the state transition of the object in GOS. Each directed arc denotes a transition and the transition conditions and actions are specified in each arc. The conditions and actions are separated by a solid line. $W(i)$ is the number of write access to object O by thread i. The number a is the threshold between 0 and 1 that will trigger the object home migration decision.

The master object has two states, namely the *PRIVATE* and *EXPORT*.

When the reference of a master object is packed within other object data, the master object is marked as exported, which means that the object will be accessible by remote JVMs. The remote read/write will increase the counter of the master object. When the write access from one remote thread dominates the total write access to the object, the object will be migrated. Therefore its state will change from *EXPORT* to *CACHE VALID* and the header of the master object will be modified to be a cached object.

The cached object has three states: *VALID*, *INVALID*, and *DIRTY*. The state *VALID* means that the cached object is valid for read and it is not modified yet. The *INVALID* state means that the copy is obsolete. Later access to the invalid copy will trigger the fetching of data from the home node that stores the master copy. The *DIRTY* state means that the object is modified.

6.3.2 Implementation

Object access

Each JVM in JESSICA2 has a daemon thread used to handle all the GOS requests. A remote read operation will cause the daemon to locate the master object, pack the data and send back the data. A remote write operation, on the other hand, will cause the daemon thread to update the master object using the data in the incoming messages.

When an object is accessed, the thread check its header to determine if it is a master object or a cached object. The read access to a master object will be done immediately without any additional checking. The write access to the master object will add an additional operation to increase the time stamp counter by one.

When an object is identified as a cached copy, the GOS service function will be called through the GOS interfaces. The GOS service functions realize the state transition according to Figure 6.3.

If the cached object is in *INVALID* state, a remote read operation will be issued through the threaded TCP communication to the remote JVM to fetch the data from the master copy. Once the up-to-date data is received, the state of the cached object will be changed to *VALID*. And the thread can proceed to operate on the cached object normally.

The update of cached object data to the remote JVM uses the *twin* and *diff* technologies that are often used in software DSM [4]. Twin is an identical copy of an object. Diff is an operation to calculate the difference between the twin copy and the original copy. As shown in Figure 6.3, a local twin operation is used when the cached object state switches from *VALID* to *DIRTY*, i.e., when the cached object is first written. The diff operation is performed when the state switches from *DIRTY* back to *VALID*. For example, suppose we have an object called *data* containing two integer fields i, and j. Assume originally $data = (i = 10, j = 20)$. When the execution of a bytecode instruction *PUTFIELD* to set i to 15, a twin copy is created. The modification on the object will be done on the original copy. At this time, $twin = (i = 10, j = 20)$ and $data = (i = 15, j = 20)$. Then $diff = (i = 15)$.

An individual update of a cached object will not be directly sent to the remote JVM. According to the JMM, the thread can keep reading and writing on its working memory unless there is a synchronization operation. Therefore all the updates of the cached objects accumulated in a thread will be propagated to the remote JVM when the synchronization happens. The diffs of all the cached objects in a thread will be calculated at the time of synchronization.

Adaptive object home migration

To support the adaptive object home migration protocol, we maintain a top 10 hit list for those master objects that are accessed by the remote threads. Each item in the hit list maintains the number of remote writes and the writing thread ids. The list can then be used to aid the detection of the *dominant writer* of an object. The list is updated when a node performs a synchronization and flushes its data to the homes of all the objects. The home of the object to be updated will update both the master copy of the data and the hit list. Since only shared objects will be recorded in the list, the size of the list will be small if the object sharing is not heavy. Other objects that have never been shared will not appear in the list.

The adaptive object home migration protocol of JMM will be driven by the thread synchronization events. The thread synchronization in GOS is implemented in a distributed manner. The Java synchronization primitives include lock(), unlock(), wait(), notify() and notifyAll(). The lock() function corresponds to the bytecode instruction MONITORENTER to enter a critical section, and ulock() for MONITOREXIT to exit the critical section. The other functions have no direct corresponding bytecode instructions, but they are supported in the standard package java.lang supplied with the JVM [48]. These include waiting on a monitor (Object.wait) and notifying other threads waiting on a monitor (Object.notifyAll and Object.notify).

Unlike the approach adopted in previous DJVM that uses existing DSM and passes these functions to the corresponding DSM synchronization APIs, we build these synchronization inside the JVM locking mechanism and use the threaded I/O interfaces inside JVM to handle the communication. No broadcast will ever be used in all the functions. As Java synchronization takes place on a Java object, we fix the JVM that owns the master copy to be the lock manager on this object. The threads in the manager will perform the synchronization in the normal way on the object. The threads in the remote JVM, when trying to perform synchronization on a cached object, will send the synchronization request to the manager. The manager will place the synchronization request from the remote threads in the proper queue of the synchronized object and return the synchronization result to the remote threads. Through thread migration and object home migration, we can distribute the homes of objects more evenly among the cluster nodes, therefore the workload arising from the thread system can be much more balanced.

Once a lock is acquired by the thread, its previous updates should be propagated to the master node. The diffs of the cached objects will be calculated node by node. The diffs will be packed. The detection of dominant writers are carried out after the diffs packing. Once an object is identified to be written by a dominant writer following the heuristics in previous section, the object home migration action will be taken. The cached object will be switched to master object by changing its location information in the header to point to the current node. Such information will be packed along with the diffs, and the packages will be sent to remote hosts. On the other end, when the node holding the previous master copy of the object receives the object home migration message, it will allocate the cached header for the object and fill in the new location information.

The object home migration needs to solve an additional problem. The home migration operation is done at the time of critical section when the thread acquires the lock. The reason is that it guarantees the operation to be done atomically without affecting other threads if all the threads are properly synchronized to access that shared object. The object home migration involves only two parties, i.e., the new home and the original home. No broadcasting messages will be sent. Therefore, a third node that is trying to access that master object after the home migration will still request the original master node because it has no knowledge of such an object home migration. To address such a problem, a home redirection message will be replied to the requester by the old home node. Upon receiving such redirected message, the third party can update the home address of the cached object and request the new home for up-to-date data. The home redirection message will cause additional round trips. The problem, however, can be suppressed by the accurate detection of the dominant writer. That is, if the home migration mainly involves the objects that are accessed solely by one thread, it won't cause other thread to encounter such a redirection problem.

Huge array caching

In Java, array is treated as an object conceptually. However special handing is needed for array caching especially when the array is huge. In GOS, a huge array (larger than 64K size) will not totally cached on a node. Instead, only a range of the array will be cached. The cached array will have additional fields in its header to define the range of the array.

The policy for choosing the array range to cache is based on the access locality. The range can be expanded or changed dynamically. For example, suppose we are caching the array range [1..100]. When a new access to the cached array is at index 150, a remote fetch operation will be issued to fetch the new range [150..200]. The two ranges [1..100] and [150..200] will be merged. Thus the cached range becomes [1..200].

6.4 Optimization techniques

6.4.1 Software object state checking

Java threads access an object through an object reference. The reference can be pointed to a master object or a cached object. As the cache granularity of our GOS is an object, it cannot take advantage of the hardware page-fault mechanism as used in the page-based DSM to detect the cache status. To check the different states of a cached copy, we need to perform software checking on every object access.

A naive implementation of global object access may direct all the object access to a function call to handle all the object access no matter the object is a master object or a cached copy. Such checking overheads may not be noticed by the interpreter-based DJVM, because it is neglectable compared with the interpretation overhead for each bytecode instruction. However, in

JIT compiler enabled DJVM, as the thread execution speed improves, the object checking by using a function call exhibits relatively high overheads. The access to the master objects will also suffer from the slow function call. The other extreme is to expand all such checking code in the compiled methods. Such treatment will lead to a significant space overhead in the generated native code which in turn burdens the hardware instruction cache during execution.

Our design is a tradeoff between these two cases, we use JIT compiler to generate native checking code to tell master objects from slave objects. One comparison instruction and one branching instruction are needed in i386 architecture for such checking. The comparison instruction tests the least significant bit (LSB) of the cache field inside the object header. If it is set, then the object is a cached object and should call the checking function to ensure that its state is valid after it returns.

Such a design alleviates the checking overhead on the master objects, however the overhead of accessing a cached object remains. We observe that if an object is cached by only a single local thread and it is set to dirty state, only the cache flushing caused by a lock operation will change the state of the object to invalid. By exploiting such a state transition behavior, we can clear the LSB of the cache pointer pretending that the object is a master object when the cached object is first turned into dirty state. The LSB of the pointer will be set back at the time of flushing. Such method provides a fast state checking for the cached objects.

6.4.2 Object pushing

As our GOS is embedded inside JVM, it is possible to get the definition of an object from the object reference. For example, it is able to tell if a field in a certain offset is an object reference. For page-based DSM, it will be difficult to have such knowledge.

Object pushing is a kind of pre-fetching technique that exploits the connectivity of Java objects. Based on the parsing of the internal field definition of an object, we can aggregate the communication messages to transfer several objects in one single message. For example, if object A contains two reference fields that point to object B and object C respectively. When A is fetched, B and C are also fetched.

As the home copy holds the up-to-date information of the object, the object references inside its fields will be more accurate than those in the cached copies. Therefore we let the decision be made at the requested site on which objects to be aggregated in the message. When remote JVM requests a master copy of an object from the home node, the home node will decide that some other objects which are associated with the requested object be pushed to the requester. On the other hand, to avoid the overheads by pushing valid cached copies in remote node which is redundant, we let the requester post a filter list of valid objects in its own cache. The home node when pushing objects, will prevent such valid objects from being packed in the message.

To hide the communication latency, we use the threaded-IO interface inside the JESSICA2 to transfer the object data. When one thread is blocked in sending object data, the thread will yield the CPU and let other thread in the local JVM continue the execution. It is also superior to the approach of simply adopting an existing object-based DSM without multithreading support.

6.4.3 TCP connection caching and adaptive communication compression

In the GOS communication we use the reliable TCP connection for transferring data back and forth. The communication is multithreaded so that when one Java thread is waiting for the communication, the other thread can continue its computation.

As TCP is connection-oriented, it needs a startup time to create the connection by handshaking between the source node and the destination node.

If a thread opens a connection once it requests a remote object and closes it once the data have been received, the startup time for all the connection will introduce many overheads. In our GOS implementation, we introduce a socket cache for thread communication. Once a connection is established between one thread in local node to another thread in the remote node, the connection will not be closed after their first communication. Instead, the socket file handle will be cached respectively in each thread's socket cache list. Later communication between the two threads will re-use the socket file handle. Thus the startup cost for TCP communication is eliminated in later communication. The socket cache will be closed when one party exits.

Another optimization to exploit the TCP communication is to use data compression. When we pack the object data in the buffer, we will adaptively compress the packed data before sending and uncompress them when receiving. The compression uses the LZ compression algorithm [46]. Only when the length of a message is between two thresholds will it be sent in compressed form. The threshold selection takes advantages of the *Maximum Transfer Unit* (MTU) of the underlying network. For example, it is usually profitable to compress a message whose size is between 1500 bytes to 2000 bytes on fast Ethernet. The reason is that the LZ compression usually gives about 60% compression ratio and such compression will usually reduce two Ethernet frames into one frame. Thus the communication cost is reduced by half in such cases.

6.5 I/O redirection

The SSI view of Java threads needs to perform I/O operations as if they were running on a single JVM. Java realizes the I/O in its libraries. The basic Java libraries include the *java.lang*, *java.math*, *java.net*, *java.io*, etc. To fulfil the SSI requirement, we need to extend Java's I/O libraries to make the I/O operations appear to be performed on a single JVM. The extension is done at the native methods of the Java libraries, which can directly access the VM kernel data structures.

The modifications include the I/O redirections for file I/O and the network I/O. The I/O redirections will provide mechanism for an I/O operation on a worker JVM to be passed to the master JVM when necessary. Some I/O operations, such as the connectionless UDP messages, that can be done without affecting the SSI view will be performed locally. Also the access to I/O objects will be directed to the GOS service routine.

Basically all I/O libraries including the AWT or Swing GUI libraries should be extended in order to support a wide range of GUI applications. In our research we aim to prove the concept of I/O redirections only and do not change all such libraries. This restricts some GUI applications from running our DJVM.

The system clock interface for Java is *System.CurrentTimeMillis()*, which returns the current time in the unit of millisecond to the user. To provide a single clock, we use the clock at the master JVM as the standard clock and introduce an adjustment for all other remote worker JVMs. When the worker JVM registers to the master JVM, the adjustment for the clock is piggybacked in the acknowledge message. The adjustment value received, after subtracting the single-trip network latency from itself, is added to the worker JVM's clock. As the single-trip network latency of normal Fast Ethernet is about 100 microseconds in Linux that is far below the unit of the clock interface, it provides an accurate single clock for all the worker JVMs.

For the file I/O and network I/O, we take over the higher half word of a 32-bit file handle to represent the id of the host that first opens the file. Later read/write operations will be redirected to a host with the id extracted from the file handle. A read-only open() operation of file system I/O first checks the local disk before redirecting the request to the master node. Other file I/O operations will always be directed to master node. For network I/O operations, the connectionless open() also will be done by the local node without further forwarding as it does not need to keep the network connection status inside kernel. The other operations such as TCP open() will be redirected to the master JVM.

The master JVM creates a daemon thread to handle such I/O operations. In case of long I/O operation such as reading long data or accept() which needs to wait for incoming connections, the master JVM will spawn another I/O thread to handle the request so that the services of master JVM will be multithreaded.

6.6 Class loading

All the JVM in each node needs to load the Java classes of the running application into their local method area. The initialization of classes on a DJVM needs to guarantee that the Java classes are loaded and initialized, so that the shared static fields of the class need to be consistent for all the JVMs.

In our system, there is no assumption of a shared file system in the implementation. The application class files can be duplicated in each node, or they can be stored only in the master node. In the latter case, when a worker JVM can not find a class file locally, it will request the class bytecode from the master JVM on demand through network communication.

The initialization of Java classes will be guaranteed to be done only once for all JVMs. When one worker JVM loads a class C, the modified JVM class loader will first query the master JVM to check if C has been loaded and initialized. All such queries from different JVMs will be sequentialized. If the initialization of C has been done, the worker JVM will fetch the static data of C, and copy them into local static data area.

6.7 Garbage collection

Currently we only have each JVM perform the garbage collection locally without exchanging garbage collection information. For the objects in the master heap area that has been exposed to remote nodes, they will be scanned by the original garbage collector and will not be garbage collected.

6.8 Summary

In this chapter, we introduce our design and implementation of GOS to support the virtually shared heap for Java threads distributed among cluster nodes. We propose our adaptive object home migration protocol based on the Java Memory Model to efficiently realize the GOS by exploiting the data access locality of Java objects. We also provide further optimization techniques to reduce the overheads of GOS in every aspect, e.g., the object checking in native code, the message number reduction, and TCP/IP communication reduction.

Chapter 7

Experiments

7.1 Environment setting

Our system JESSICA2 [76, 77, 78, 79] is based on the modification of an open-sourced JVM Kaffe 1.0.6 version [80], which is compatible with JDK1.1 API. Though Kaffe is not a high-performance JVM implementation compared to the commercial JVM products, Kaffe provides all the necessary JVM services. It includes the JIT compiler, the garbage collector, and the Java libraries. There are quite a few academic projects based on Kaffe JVM, such as Latte JVM [81] and Kaffe OS [10].

By using Kaffe, we do not mean to introduce new enhancements to VM kernel technologies such as JIT compilation or garbage collection, which are valuable for high-performance Java computing on a single machine. Instead our focus is on extending the scale of the execution environment for a multithreaded Java application from a single node to a distributed environment such as a cluster.

We use the HKU Gideon 300 Linux cluster [70] as our main experimental platform. It consists of 300 nodes connected by a single, high-port density Foundry Fastiron 1500 Fast-Ethernet switch. Each node is a standard PC consisting of an Intel 2GHz Pentium 4 processor, a 512 Mbytes (PC2100)

DDR SDRAM, and a 40GB IDE hard disk.

The Gideon cluster can be configured to run different Linux kernels. The kernel version ranges from 2.2 to 2.4. In our experiment, we choose Linux 2.4.22 kernel, with gcc 2.95.3 compiler.

The performance study will evaluate the various factors that affect the performance of a DJVM. Through the evaluation of thread migration, the GOS optimization effects, and the application speedup, we can have a better understanding of the efficiency of the DJVM, as well as its bottleneck.

7.2 Benchmarks

In our experiments, we use several sets of Java benchmarks. In this section, we provide their detailed information.

7.2.1 SPECjvm98

SPECjvm98 [27] benchmark is a popular benchmark for measuring the performance of JVM. Table 7.1 shows the brief description of each benchmark in SPECjvm98 suit. To use the SPECjvm98 benchmarks, the JVM used needs to be compatible with JDK 1.1 API or later. SPECjvm98 is used in our experiments for measuring the thread migration overheads.

The benchmark *compress* implements a high-performance compression algorithm, i.e., the modified Lempel-Ziv method (LZW). It finds common substrings and replaces them with a code of variable size during the compression.

The program *jess* stands for the Java Expert Shell System, which is based on NASA's CLIPS expert shell system. The system continuously applies a set of rules to a set of fact lists in order to solve a set of puzzles commonly used with CLIPS.

The program *db* simulates the operations on a memory resident database. Its input includes a 1 MB file that consists of contact records, and a 19KB file that contains a stream of operations to perform on the input records. The operations include adding, deleting, searching, and sorting.

The program *javac* is the Java compiler adopted from the JDK 1.0.2. The program *mpegaudio* decompresses the ISO MPEG Layer-3 audio files. The workload consists of about 4MB of audio data. Both *raytrace* and *mtrt* realize a raytracer that renders a scene depicting a dinosaur. The raytrace program is single-threaded, while the mtrt is a multithreaded program. The program *jack* is a Java parser generator based on the Purdue Compiler Construction Tool Set (PCCTS).

Benchmarks	**Description**
compress	Lempel-Ziv compression
jess	Java expert shell system
db	Simple memory resident database
raytrace	Ray-tracing program
javac	Java compiler from the JDK 1.0.2
mpegaudio	Audio file decompression
mtrt	Two-thread ray-tracing
jack	Java parser generator

Table 7.1: Description of SPEC JVM98 benchmarks.

7.2.2 Multithreaded Java benchmarks

Another set of multithreaded Java benchmarking programs is collected from different sources and it is used in our experiments to evaluate JESSICA2's performance in different aspects.

The π *calculation* (CPI) is a multithreaded Java program that calculates an approximation of π by evaluating the integral. In CPI, the main thread creates a number of worker threads, each accumulating the value of a part of the integral. The sum of all the results from each thread will be the approximation value of π.

The *Successive Over-Relaxation* (SOR) benchmark is a multithreaded

Java program that does red-black successive over-relaxation on a 2-D matrix. It is commonly used in scientific applications, such as finite difference computation. Each thread is assigned with a sub-block of the 2-D matrix for computation.

The *All-pair Shortest Path* (ASP) program is a multithreaded Java benchmark that calculates the shortest path between any pair of nodes in a graph using a parallel version of Floyd's algorithm. The algorithm uses a connectivity matrix and keeps updating it to reflect the shortest known path in a loop. Each thread in the program is responsible for computing the shortest pathe from a subset of nodes assigned to it.

The *N-Body simulation* (NBody) program realizes the algorithm of Barnes & Hut [16] to simulate the motion of particles in a 2D space due to gravitational forces over a fixed amount of time steps. The Barnes-Hut method uses a hierarchical tree to efficiently calculate the inter-particle distances among the particles. In the multithreaded Java benchmark, the master thread creates a number of worker threads, and assigns each with a chunk of particles for the force calculation. The iteration begins with the master thread building the Barnes Hut tree. The forces among the particles will be calculated by the worker threads and the positions of the particles will be updated. In next iteration, the master thread will rebuild the tree based the updated data.

The *Traveling Salesman Problem* (TSP) program is a multithreaded Java application that finds the shortest route among a number of cities by visiting all the cities and returning to the starting city. It is a famous NP-complete problem. The multithreaded benchmark uses a parallel branch-and-bound algorithms to solve the problem. Each thread is assigned with a different starting city to search the shortest route. The shortest route found so far will be used by all the threads in pruning the search trees.

The *Parallel Adaptive Mesh Refinement* (PAMR) simulation program is a multithreaded Java application that simulates the computation on grid points using Adaptive Mesh Refinement (AMR) method. The basic idea of AMR is to refine the more interesting regions at higher resolutions, while leaving the less interesting parts of the domain at lower resolutions. The quality of the approximate solution can be preserved while keeping the total number of grid points small through AMR. In the multithreaded version, the PAMR simulation, each thread is assigned with a part of the grid points and the workloads of all the threads vary due to the different resolution requirements on different grid points.

The *N-Queen* benchmark solves the problem of putting N queens in the $N \times N$ chess board in a safe state. The state is safe if no two queens attack each other, i.e., no two queens are placed on the same row, the same column, or the same diagonal. The parallel program uses a branch-and-bound recursive algorithm. The search space is divided by fixing some queens in fixed columns. Each thread is responsible for searching in some sub-spaces.

The *Raytracer* is a multithreaded 3-D ray tracing program adopted from the *Java Grande Forum Multithreaded Benchmark* (JGFMB) suit [30]. The outermost loop of the benchmark is parallelized over rows of pixels. The scene rendered contains 64 spheres, and is rendered at a resolution of $N \times N$ pixels.

The *Series* is another multithreaded Java benchmark from JGFMB, which computes the first N fourier coefficients of the function $f(x) = (x+1)^x$ on the interval [0..2]. Transcendental and trigonometric functions are heavily used in the benchmark. The benchmark loops over the Fourier coefficients, with each iteration being independent of every other loop. The work of the loop is divided in blocks. Each thread is responsible for updating the elements of the block assigned to it.

The *Molecular Dynamics* simulation (Moldyn) is another N-Body program from JGFMB, which models particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. It calculates the force on a particle in a pairwise manner. The benchmark loops over all particles in the system. The loop is parallelized by dividing the range of the iterations between the threads in a cyclic manner to avoid load imbalance.

7.3 Evaluation of thread migration

7.3.1 Thread migration overheads

In the experiment, our goal is to evaluate the effect of enabling thread migration on the thread execution performance. In our system, the thread execution will be in two modes. One is the normal execution, in which the thread executes its task according to the requirements of the program. The other mode is the thread migration operation, which is introduced by our DJVM for load balancing.

We measure the cost introduced by enabling the thread migration support in JESSICA2 during the normal execution of the thread. In the DNCI approach, the time overheads of thread migration are mainly due to checking at the migration points. In the M-points, the overheads include the spilling of register values and variable types into the stack slots, and the checking of migration flag. In the B-points, the overheads include the checking of migration flag only. The space overheads are mainly due to the instrumented native code for both M-points and B-points. Such overheads are eliminated in our JITR approach.

In the experiment, we disable the object checking in the GOS support because the checking overhead is not relevant to the thread migration. We use two sets of benchmarks in the measurement. One set is the SPECjvm98 benchmarks, while the other set includes some multithreaded Java programs discussed in previous section.

Using SPECjvm98 benchmarks

Though most of the applications in SPECjvm98 are not multithreaded, the test can still show the overheads of thread migration on typical applications. In the single-threaded SPECjvm98 benchmarks, the master thread can still be migrated through DNCI. The overheads caused by the migration points inserted in the master thread will be measured.

The initial heap size for JESSICA2 is set to 48MB and the benchmarks are running on single node. We compared the differences in time and space costs between enabling and disabling the migration checking at migration points. The measurements on all the benchmarks in SPECjvm98 were carried out 10 times and the values were then averaged on one single node in the cluster.

Benchmarks	Time (seconds)		Space (native code / bytecode)	
	No migration	Migration	No Migration	Migration
compress	11.31	11.39(+0.71%)	6.89	7.58(+10.01%)
jess	30.48	30.96(+1.57%)	6.82	8.34(+22.29%)
raytrace	24.47	24.68(+0.86%)	7.47	8.49(+13.65%)
db	35.49	36.69(+3.38%)	7.01	7.63(+8.84%)
javac	38.66	40.96(+5.95%)	6.74	8.72(+29.38%)
mpegaudio	28.07	29.28(+4.31%)	7.97	8.53(+7.03%)
mtrt	24.91	25.05(+0.56%)	7.47	8.49(+13.65%)
jack	37.78	37.90(+0.32%)	6.95	8.38(+20.58%)
Average		(+2.21%)		(+15.68%)

Table 7.2: The execution overheads of DNCI using SPECjvm98 benchmarks on single node.

Table 7.2 shows the test results of DNCI approach on single node. The space overheads are in terms of the average size of native code per bytecode instruction, i.e., the blowup of the native code compiled from the Java bytecode.

From the table we can see that the average time overhead charged to the execution of Java thread with thread migration is about 2.21% and the space overhead due to the generated native code is 15.68%. Both the time and space overheads are much smaller than the reported results from other static bytecode instrumentation approaches. For example, JavaGoX [65] reported that for four benchmarks (Fibo, qsort, nqueen and compress in SPECjvm98), the additional time overhead ranges from 14% to 56%, while the additional space cost ranges from 30% to 220%.

Though the overhead is much reduced in the DNCI approach, it can be further improved by our JITR approach. The result of using JITR does not introduce any overhead since there is no code instrumentation during the normal thread execution.

Using multithreaded benchmarks

In the second benchmark set, we include four more multithreaded Java applications, namely, CPI, SOR, ASP, and NBody. The test is running on one single node just like the SpecJVM98 benchmarks.

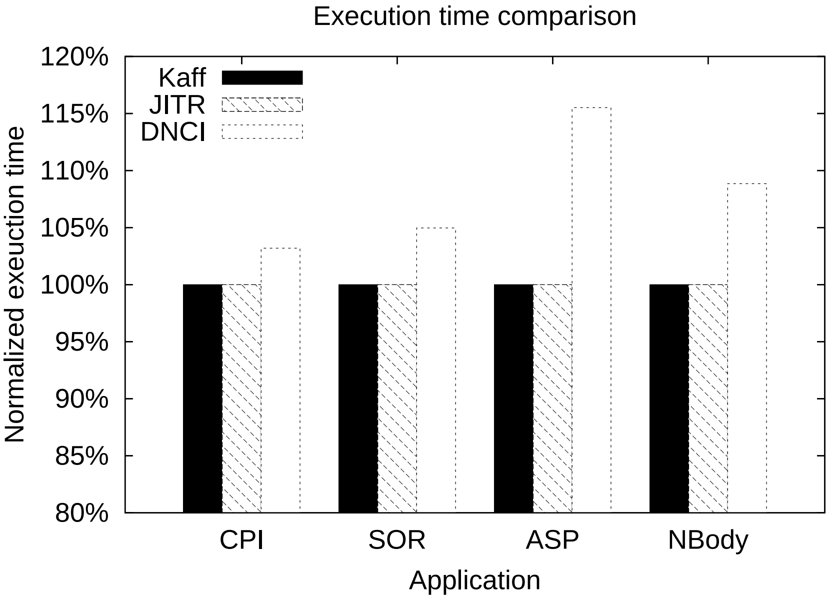


Figure 7.1: Normalized execution time comparison during normal execution.

We first measure the time and space overheads caused by the two approaches in realizing the thread migration. Figure 7.1 shows the normalized

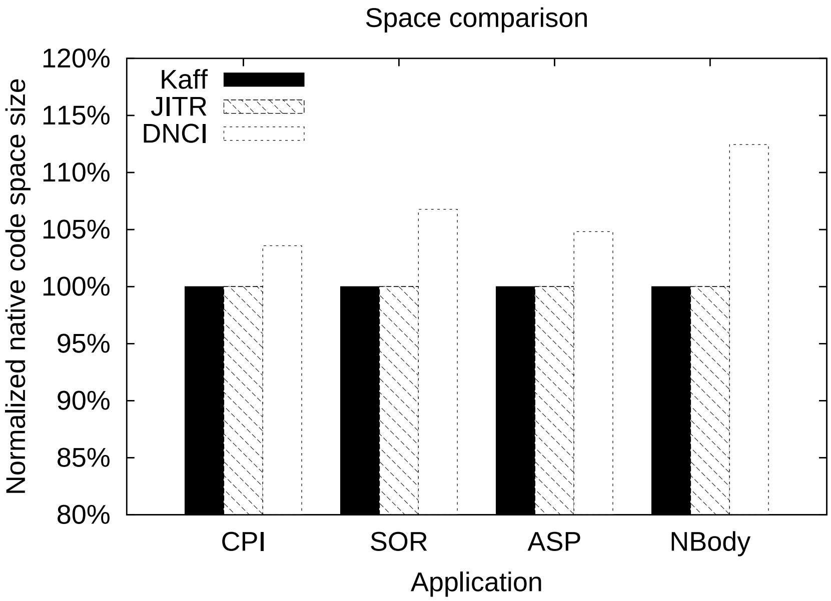


Figure 7.2: Normalized space comparison during normal execution.

execution time comparison, and Figure 7.2 shows the normalized space cost comparison. The execution time overheads and space overheads are measured during the normal thread execution without migration. In the figures, "Kaffe" means the original unmodified JVM Kaffe version 1.0.6, "JITR" means our JITR approach, and "DNCI" means the DNCI approach. The Kaffe's time and space are set to 100 in the y-axis. The results of the others are shown as ratios over Kaffe's values.

As expected, JITR does not introduce overheads in both time and space during normal thread execution. Its execution time and space are the same as those of the unmodified Kaffe JVM. The largest time overhead of DNCI reaches about 16% in the SOR program. The space overhead in the NBody program reaches about 13%. The result shows that JITR totally eliminates the overhead introduced in the case of DNCI during normal thread execution time.

7.3.2 Migration latency and breakdown

We proceed to evaluate the cost of the migration operation using two nodes running JESSICA2. It is measured in terms of migration latency. The latency includes the time from the point of stack capturing in the source node to the time when the thread has finished its stack restoration on the remote node and has sent back the acknowledgement. In the experiment, we compare the results of the DNCI approach and the JITR approach with/without the latency hiding technique. We adopt the four multithreaded Java benchmarks used in the previous overhead experiment, namely, CPI, SOR, ASP, and NBody. The migration point is selected at the first Java frames in the thread stack.

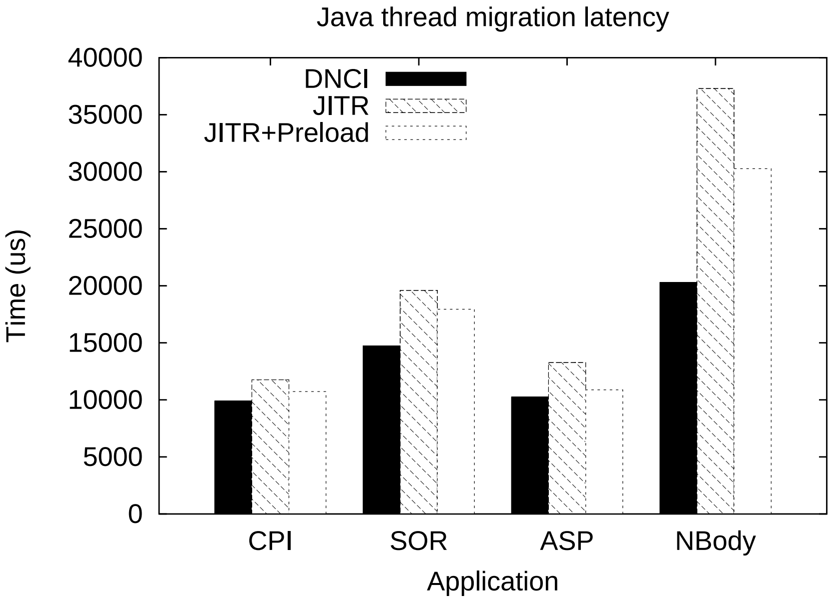


Figure 7.3: The Java thread migration latency.

Figure 7.3 shows the execution time for the migration operations in three cases. The "DNCI" means the DNCI approach. The "JITR" means JITR approach. The "JITR + Preload" means the latency hiding technique is enabled in the JITR approach. The latency hiding proves to be effective in all the four benchmarks. From the figure, the “JITR + Preload” can cut the extra overhead caused by the recompilation to about a half compared to that of the DNCI approach on average.

JITR(μs)	CPI(1 frame)	SOR(2 frames)	ASP(1 frame)	NBody(8 frames)
stack walk	7.35	11.73	12.94	13.73
segmentation	228.97	250.71	309.66	333.9
positioning	1002.46	3576.89	1466.72	5237.09
breakpoint	0.52	1	0.35	2.9
type derivation	5.2	8.11	3.64	28.51
translation	1047.36	3702.19	1485.17	6275.27
patching	0.07	0.1	0.1	0.12
parsing	141.6	141.6	138.59	338.46
thread creation	125.53	125.53	135.97	194.39
register patching	942.33	964.89	3000	11965.47
frame rebuilding	52.63	52.63	72.39	57.36
misc. IO	7173.66	9069.66	4255.32	5820.29

Table 7.3: Latency breakdown of JITR.

DNIC(μs)	CPI(1 frame)	SOR(2 frames)	ASP(1 frame)	NBody(8 frames)
capturing	74.47	168.88	86.11	356.37
parsing	122.46	150.07	124.42	362.3
thread creation	112.85	119.6	116.65	118.67
translation	949.51	2941.6	3193.41	5932.6
resolution	34.6	110.78	150.71	43.79
setup frame	16.13	107.6	92.33	22.14
misc. IO	8588.89	11138.21	6484.62	13460.13

Table 7.4: Latency breakdown of DNIC.

Table 7.3 and Table 7.4 show the detailed breakdown of thread migration using JITR and DNCI respectively. In JITR part, the upper half of the steps corresponds to the thread context capturing using the JITR approach. The lower half is result of the restoration phase. The type derivation is much lightweight that it needs only a few microseconds because it just linearly scans the bytecode in Java methods. The most costly part in the capturing phase is the positioning of bytecode PC and the translation. Because they need to invoke the JIT compiler to recompile the methods in the captured frames. In the restoration phase, the register patching needs to compile the Java methods. Thus it is costly in the restoration phase. The time for data fetching of Java objects and the communication time for transferring the thread context, is shown in last row. It is the dominant cost in the whole latency. All the other steps are lightweight. Their time ranges from a few microseconds to a few hundreds microseconds in the test.

In DNCI, the RTC-BTC transformation time is shown in the row named "Capturing". It does not have the other detailed steps like those in JITR because it only needs to scan the memory and collect the context data. The capturing time is proportional to the number of frames and the number of variables inside the frames. But the time is much smaller than JITR, because DNCI does not need to recompile the Java methods in its RTC-BTC transformation. The more frames are included in the migrated thread, the more time will be saved in DNCI compared to JITR.

There are slight time differences on the destination node between DNCI and JITR in the steps of translation, resolution, and frame setup. This is because in DNCI, when compiling the migrated frames, the native code instrumentation is also invoked. The additional efforts in instrument the native code in DNCI add some slight overheads to the restoration time.

7.4 Speedup of scientific applications

We use seven multithreaded scientific applications as our benchmarks to measure the speedup using JESSICA2. They are CPI, TSP, NBody, Raytracer, Series, N-Queen, and Moldyn. In the test, we use up to 32 nodes on Gideon cluster.

We run CPI with 9,000,000,000 iterations, TSP with 14 cities, NBody with 640 particles in 10 iterations, Raytracer with a 150x150 scene containing 64 spheres, Series with data size 100,000, N-Queen with 16×16 board size, and Moldyn with $4 * 8^3$ particles. Since the scientific applications have the characteristics of balanced regular iteration, not many migration operations happen in the execution. Therefore we use JITR thread migration scheme to test the speedup.

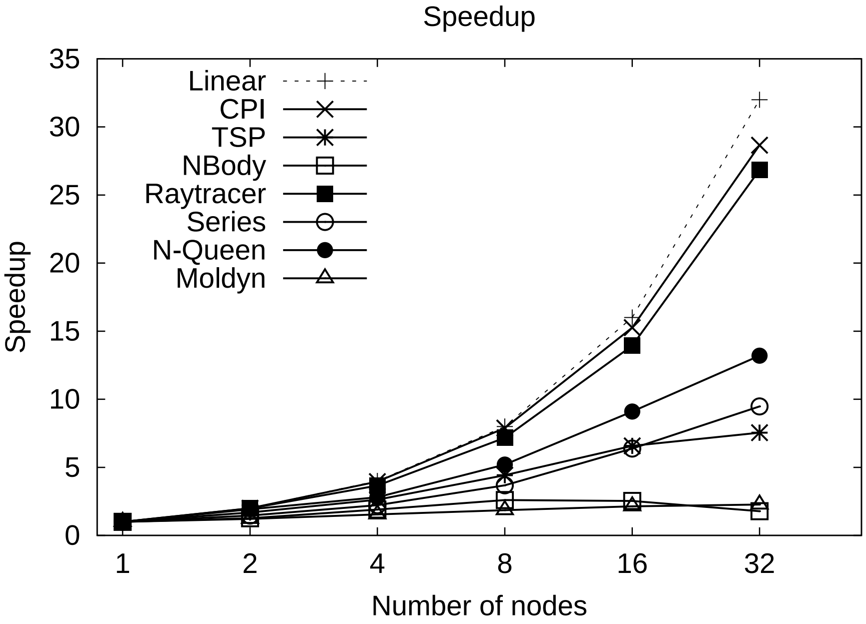


Figure 7.4: Speedup measurement of Java applications

Figure 7.4 shows the speedup by comparing the execution time of JESSICA2 and that of Kaffe 1.0.6 (in a single-node) under JIT compilation mode. From the figure, we can see nearly linear speedup in JESSICA2 for CPI and Raytracer. The is due to the heavy computation/communication ratio in these two applications. They are typical embarrassingly-parallel applications, with each thread computing independently with few interactions. In the CPI, all threads are computing their partial result for π independently. Only when the threads finish their jobs, will they enter a synchronization for updating the shared data holding the value of π. In Raytracer, after receiving the messages of the scene data, all the threads can begin to render the scene without communication with others.

For TSP, N-Queen, and Series, the speedup curves show about 50% to 60% of efficiency on 8 nodes. But when the number of nodes increases to 32, the efficiency begins to drop by about 20%. The number of messages exchanged between nodes in TSP has been increased because the migrated threads have to access the shared job queue and to update the best route during the parallel execution, which will result in flushing of working memory in the worker threads. In N-Queen, the inner loop contains the checking of the safety of queens in the 2-D array that represents the chess board. The access to array has caused checking overheads which leads to a little slow down.

The other class of applications exhibits small computation/communication ratio. The examples are NBody and Moldyn in our test. These two applications are two variants of N-Body computation. The poor speedup is expected, which is due to the frequent communications between the worker threads and the master thread in updating the forces of the moving bodies, and computing the Barnes-Hut Tree in each iteration.

From the test, we can conclude that the computation/communication ratio is the dominant factor that limits the scale of multithreaded Java applications on DJVM. Our efforts in reducing communication cost in GOS design and increasing execution speed through JIT compilation and thread migration, therefore, address this problem so that more multithreaded Java applications can be run efficiently on a DJVM.

7.5 Evaluation of GOS

Figure 7.5 shows the result of the GOS checking overheads for SPECjvm98 benchmarks on single node. We use JITR for thread migration. Therefore there is no thread migration overhead involved in the test. The GOS checking overheads come from the native code generated to check if an object is a cached object before accessing its fields. From the table, we can see that the overheads in most of the benchmarks are below 10%. However, for the program compress, due to the heavy array accessing of the compressed data in the main compression loop, the slow down of compress reaches about 93.4%. This calls for more advanced compiler analysis technologies for reducing array checking time overheads.

Benchmark	Time overhead
compress	93.4%
jess	7.2%
raytrace	10.2%
db	10.6%
javac	6.3%
mpegaudio	30.8%
mtrt	6.8%
jack	7.4%
Average	21.6%

Table 7.5: The time overheads for GOS checking using SPECjvm98 benchmarks on single node.

We also evaluate the effect of GOS optimization techniques in reducing the communication messages. We use the multithreaded Java programs including TSP, NBody, SOR, and ASP.

We use 8 nodes in the test. The problem size for TSP is 14 cities. In N-Body we use 512 objects. For SOR, we use a 2-D array of with size 2048x2048. For ASP, we use 512 cities. Figure 7.5 shows normalized result of message reduction after applying the adaptive object home migration protocol (H), fast software checking (F), and object pushing (P) respectively. The "H+P+F" means that all optimizations are enabled. Figure 7.6 shows the corresponding execution time.

The adaptive object home migration protocol introduces significant message reduction in SOR and ASP. Because in these applications, the object accessing exhibits single-writer pattern, i.e., most of the objects in the shared arrays are accessed only by one thread. After GOS discovers the pattern, the homes of these objects will be dynamically changed to the nodes where the accessing threads reside. As a result, the overall execution time also improves

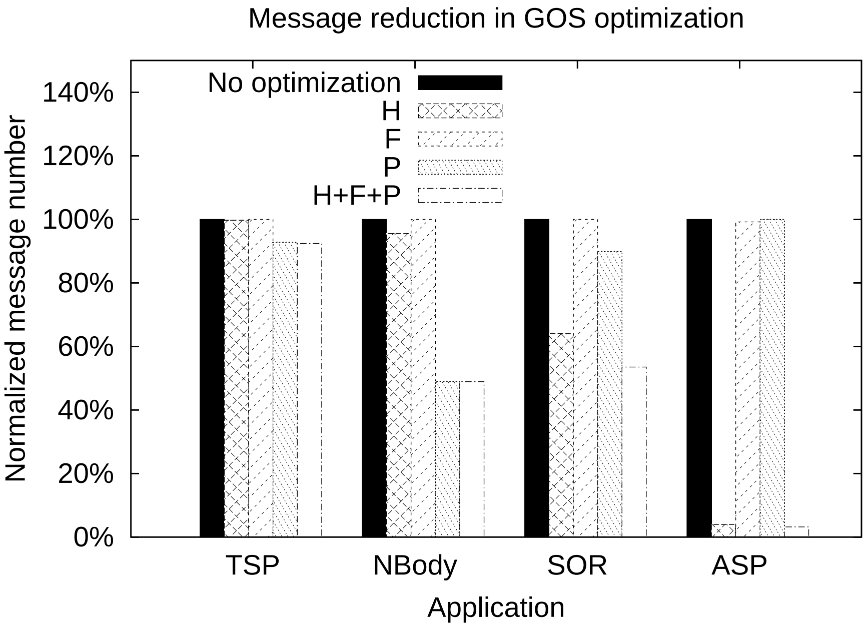


Figure 7.5: GOS optimization results in message reduction.

in both applications. For NBody and TSP, the effect of adaptive home migration is not obvious. In NBody, new objects are created in rebuilding the Barnes & Hut tree. Therefore there are few objects that are repeatedly accessed by one single thread. In TSP, because it is computation intensive, only limited objects such as the shortest path array are updated during the execution. Therefore all the optimizations of GOS do not reduce the message number. And it does not speedup the application.

In all the applications, the fast software checking doesn't reduce the total number of message size. But it reduces the execution time in NBody, SOR, and ASP. In NBody, there are many accesses to the particle objects in each iteration. In both SOR and ASP, arrays are extensively used in the computation. Therefore the checking optimization improves the performance for SOR and ASP much more than NBody.

The object pushing improves NBody due to the creation of large number

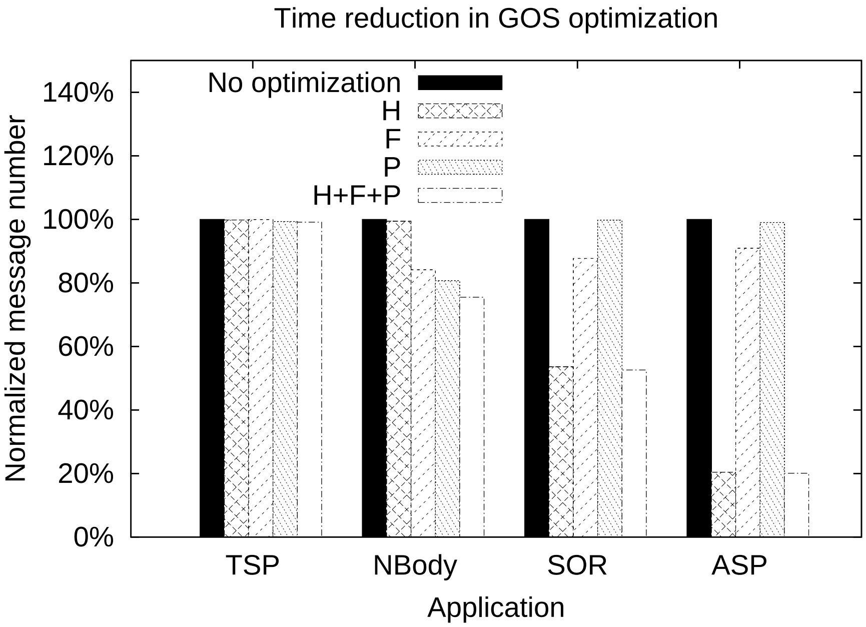


Figure 7.6: GOS optimization results in execution time.

of new objects in the master thread. Though adaptive object home migration protocol cannot improve this case, the object pushing can improve the situation by aggregating the object data in one single message. As a result, the object pushing can reduce the message number in NBody by nearly 60%.

7.6 Dynamic load balancing experiment

In this section, we introduce several applications of thread migration in our distributed JVM and evaluate the effect of thread migration.

7.6.1 Irregular multithreaded Java applications

We use two irregular multithreaded Java applications, TSP and PAMR. In our test, the multiple threads will be spawned among the running nodes when the application starts. We compare the difference between enabling and

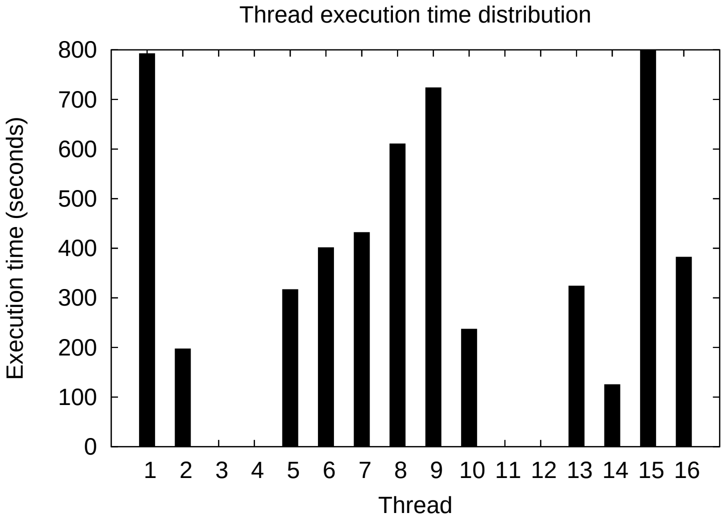


Figure 7.7: Thread execution time distribution of TSP.

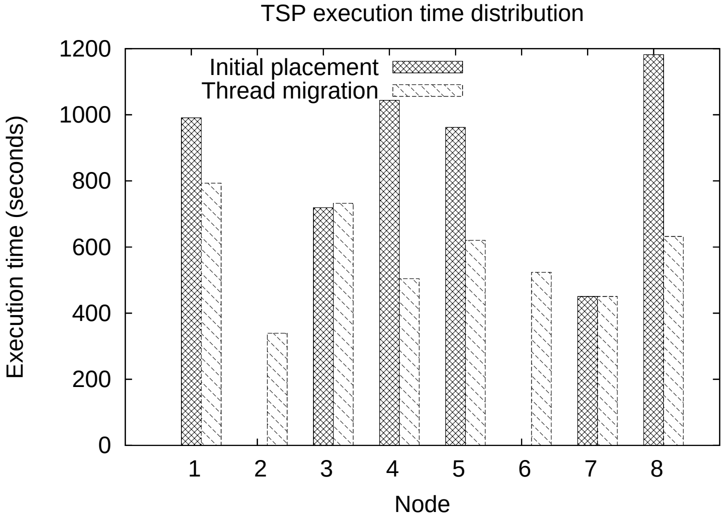


Figure 7.8: Execution time distribution of cluster nodes.

disabling thread migration after the initial spawning. The thread migration is driven by a load balancing module that monitors the CPU and memory usage of different nodes participating in the execution. The module will migrate threads from heavily loaded nodes to lightly loaded nodes when there is imbalance among the nodes.

In TSP, we use 16 threads to compute the shortest path to cover the 13 cities. The execution time distribution of the 16 threads are shown in Figure 7.7. We run the multithreaded TSP on 8 nodes and compare the time distribution of initial placement and thread migration. Figure 7.8 shows the result. From the figure, we can see that thread migration has made the total execution time of different nodes more balanced than that of initial placement. The standard deviation of initial placement is 438 seconds. By thread migration, the deviation drops to 152 seconds. The total execution time is therefore saved from 1203 seconds using initial placement to 793 seconds (saving about 33.6%) using thread migration.

In the PAMR simulation test, we use 64 threads to perform floating point operations in an 8x8 mesh, each concentrating on one cell of the mesh with few communication to each other. The required resolution for each cell of the mesh is assigned with random number. The workload of each thread, which is in terms of the number of floating point operations, varies dramatically, as shown in Figure 7.9.

Figure 7.10 shows the speedup comparison of initial placement and thread migration using up to 16 nodes. From the figure, we can see that thread migration outperforms initial placement. The reason is that the initial placement scheme has grouped some computation threads that happen to have heavy workloads in the later computation while the thread migration scheme can dynamically recover from such a situation to find some lightly loaded nodes to share the heavy workload.

7.6.2 Non-dedicated environment

We run our test in a non-dedicated cluster environment, where different jobs are submitted to time-share the resources in the cluster nodes. In the test, we use 8 nodes from the Gideon cluster in our experiments. According to

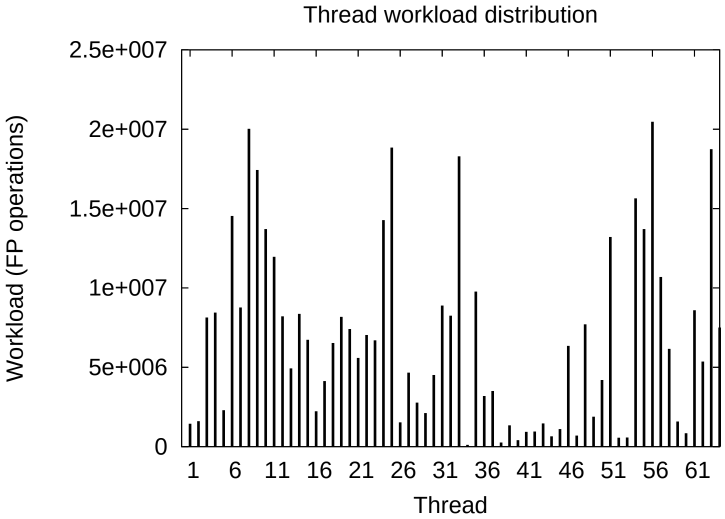


Figure 7.9: Thread workload distribution of PAMR simulation.

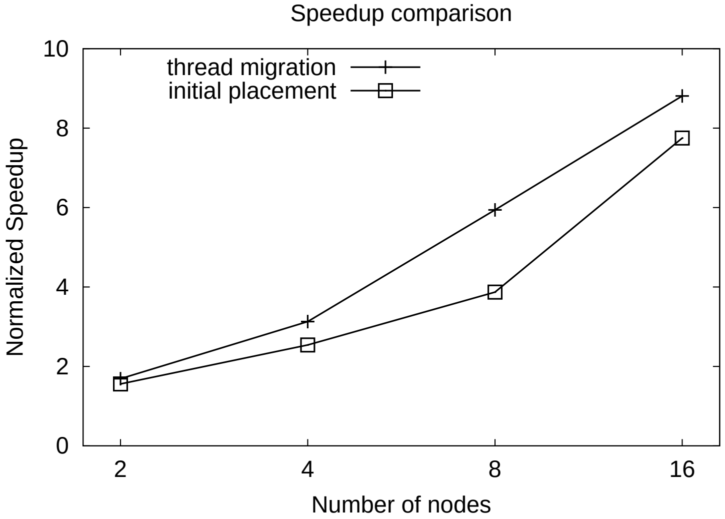


Figure 7.10: Speedup comparison of PAMR.

the cluster workload monitored, we choose two nodes that are heavily loaded, while the others are lightly loaded. The 5-minute load average of the heavily loaded nodes is larger than 3.0, while that of the lightly loaded nodes is below

0.1. We then run our applications on such a non-dedicated environment and observe its speedup with thread migration.

Four multithreaded Java applications are used in the test. They are CPI, Raytracer, Series, and N-Queen. Except for N-Queen, the first three benchmarks all have balanced workload. However, when all are running in a non-dedicated environment, their speedups are affected by the varied system workload in different nodes. The slowest node will slow down the execution of the whole application. Figure 7.11 shows the speedup comparison between using thread migration and disabling thread migration.

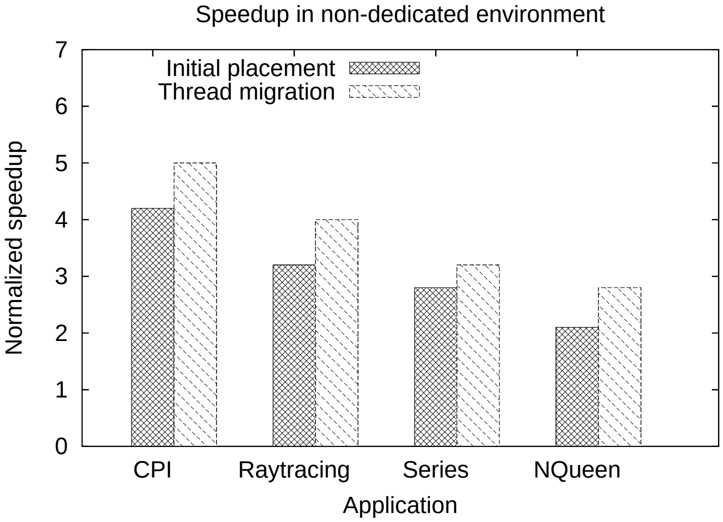


Figure 7.11: Evaluation of thread migration in a non-dedicated environment.

From the figure, we can see that thread migration in DJVM can help to reduce the execution time of the running multithreaded applications by migrating threads from overloaded nodes to underloaded nodes, even if the workload in original threads are balanced.

7.7 Summary

In this chapter, we present the performance evaluation of JESSICA2 on Linux clusters. We analyze the system overhead of providing the key feature of JESSICA2, i.e., the thread migration. Then we give the performance evaluation of the GOS sub-system. We also compare the performance of JESSICA running in interpretive mode and that of JESSICA2 that runs in JIT compilation mode to justify the use of JIT compilers in DJVM. Next we give the speedup results of JESSICA2 on multithreaded Java applications to show the advantages of a Distributed JVM. Finally we conduct experiments to show how load balancing using the thread migration mechanism in JESSICA2 improves the Java applications' performance.

The performance evaluation proves that JESSICA2 is a high-performance computing platform for multithreaded Java applications. Its JIT compiler-assisted thread migration mechanism together with the optimization in distributed object sharing are the key factors to help improve the performance of multithreaded Java applications.

Chapter 8

Conclusions and future work

8.1 Conclusions

The goal of our research on DJVM is to exploit the power of PC clusters for high-performance Java computing. In the JESSICA2 project, we research on thread migration and distributed object sharing to support a high-performance SSI cluster middleware. The system is built at the JVM level, and it can transparently execute multithreaded Java applications in parallel without requiring any modification to the Java source code or Java bytecode.

The novelty of our research lies in the application of a JIT compiler to support the above two features. The following components of the original single-node JVM have been modified or extended: the JIT compiler, the class loader, the heap and the I/O class libraries. Our experimental results confirm that it is feasible to deliver high performance for multithreaded execution using Java through the support of a DJVM in a cluster environment.

The research will have strong impacts in the following areas.

- High-performance Java servers. Using message passing for implementing distributed applications, such as the servers, is hard, since it needs many programming efforts. The programmer needs to remember different APIs, partition the work load, and maintain the data consistency among the distributed nodes. The bad programmability in the message passing tools has made the development of high-performance distributed servers tricky and error-prone.

Today multithreaded programming using Java is frequently adopted in server side programming with the increased popularity of Java, due to its high portability, as well as the good programmability in supporting OO programming. There is a need to provide a high-performance platform for running such applications on a larger scale system than the SMP machine. A DJVM answers this need. With the introduction of the DJVM, the server programs can immediately benefit from such a high-performance computing platform on low-cost clusters. The DJVM automatically hides the machine boundaries in the distributed environment from the programmers, and provides support for automatic load balancing among the running machines. Therefore, the effort to tune the performance on clusters at the application level by the programmers is released with the help of the DJVM.

- SSI cluster computing. SSI is the ultimate goal of a cluster, which allows the user to control and program the cluster as a single machine. Though SSI cluster can be realized at the hardware level, the operating system level, or the application level, the middleware level is proved to be more reliable and portable. It does not need to modify the underlying operating systems.

  The DJVM realizes SSI at the middleware level. It provides the Java programmers the ability to program cluster in the shared-memory paradigm. The whole cluster is now abstracted as a single JVM that provides extreme computation power, huge memory space, and strong I/O capabilities. Such a friendly environment will make clusters more attractive for high-performance computing in both sciences and business.

- Computation mobility. Computation mobility [35] mainly has two types, i.e., strong mobility and weak mobility. Strong mobility allows the program context to move during execution. It is an attractive service, based on which many useful functions can be realized, such as load sharing, reconfiguration, and high availability.

Much attention has been paid on Java in supporting code mobility. Our work provides a breakthrough in supporting the strong mobility of Java threads in JIT-enabled JVM. In our work, the native code instrumentation approach and the JIT recompilation approach introduce new methods in the transformation of a native thread context into a portable thread context using JIT compilers. The technologies can be adopted in Java mobile agent systems for achieving high-performance native execution in JIT-enabled JVMs. Our work can also be used in checkpointing Java threads for fault tolerance computing.

- Distributed object sharing. The usual software DSM to provide distributed object sharing normally requires that the programmers explicitly point out the shared objects using additional annotations or using special DSM APIs. Our research provides the location transparency for accessing Java objects on clusters. All the object sharing and synchronization support is embedded seamlessly at the JVM level without the involvement of the programmers.

  The tight integration of distributed object sharing inside the DJVM also provides a new direction for the research in DSM. Our research shows that distributed object sharing tightly coupled with the language semantics could provide more opportunities for reducing communication overheads, and developing new consistency models. The optimization techniques used in our work can be examples for such a direction.

8.2 Future directions

JESSICA2 has been developed for high-performance multithreaded Java computing on clusters. It covers a lot of issues like distributed thread scheduling, distributed object sharing, SSI and JIT compilation. Nevertheless, some issues have not been addressed, which can become areas for our future study.

8.2.1 Distributed garbage collection (DGC)

Current JESSICA2 pins down the shared objects that are accessed by the threads on multiple nodes to prevent them from being collected. Each local JVM in JESSICA2 does garbage collection locally. This results in a waste of memory to store the unused shared objects.

To support the garbage collection on the shared objects, efficient DGC algorithms are needed. The requirement on DGC is that it should be accurate and fast. DGC should be incremental and should not stop the whole JESSICA2 when it does the garbage collection. This needs efficient data structures and algorithms to scan and exchange the shared object data.

8.2.2 Advanced JIT compilation support

Current JESSICA2 uses the simple Kaffe JIT compiler as its execution engine. The speedup of the JIT compiler versus the interpreter is significant in Kaffe. However, compared to the advanced JVM JIT compiler, Kaffe's JIT compiler is less efficient.

Advanced JIT compilers can bring higher local execution performance for DJVM, which will in turn benefit the execution performance of the multithreaded Java applications running on top of DJVM. To support the thread migration in native mode, such advanced JIT compilers need to be modified, which is more difficult than what we did in Kaffe's JIT compiler.

8.2.3 SSI GUI support

To provide the complete SSI illusion for a wider range of Java applications such as the GUI applications, more supports are needed in Java I/O libraries (such as the AWT or the Swing library). Also, in a large cluster environment, the master node could easily turn into a bottleneck for I/O requests. New techniques for duplicating the master node can be explored.

8.2.4 Optimization on I/O operations

Pure Java applications usually rely on thread pools to handle large-volume of the external TCP connections. Current DJVM is restricted in supporting the handling of the connections in parallel in each participating node. Most of the TCP connection needs to be forwarded to the master node in order to preserve the SSI. Changes in SSI illusion for multithreaded Java server applications on clusters can be explored in a DJVM, in order to efficiently handle the I/O in parallel. And the DJVM should be able to support the execution of multiple different Java applications simultaneously, and protect their shared data from each other.

8.2.5 SSI middleware for other languages

The practice of JESSICA2 can provide the insight in designing the SSI middleware for other languages such as C# (pronounced “C sharp”) [39]. C# is a Java-like language that is an object-oriented and type-safed. C# includes multithreading at its language level like Java. The architecture of C# resembles Java in its portable intermediate code, Virtual machine architecture and JIT compilers. As a result, the idea of DJVM can be applied to build a distributed C# virtual machine for high-performance threaded computation on clusters running Microsoft’s Windows operating systems.

Bibliography

[1] Jigsaw – w3c's server. http://www.w3.org/Jigsaw/, 2004.

[2] Ali-Reza Adl-Tabatabai, Micha&#322; Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 280–290. ACM Press, 1998.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] 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, February 1996.

[5] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations). *IEEE Micro*, 151:54–64, Feb. 1995.

[6] Gabriel Antoniu, Luc Boug&#233;, and Raymond Namyst. An efficient and transparent thread migration scheme in the pm2 runtime system. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 496–510. Springer-Verlag, 1999.

[7] Gabriel Antoniu et al. The Hyperion System: Compiling Multithreaded Java Bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

[8] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[9] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and Jose E. Moreira. High Performance Numerical Computing in Java: Language and Compiler Issues. In *Languages and Compilers for Parallel Computing*, pages 1–17, 1999.

[10] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000. USENIX.

[11] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California at Berkeley, December 1997.

[12] M. Baker, B. Carpenter, G. Fox, and S. H. Koo. mpiJava: An Object-Oriented Java interface to MPI. In *Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP* , 1999.

[13] Mark Baker and Rajkumar Buyya. Ieee computer society task force on cluster computing. http://www.ieeetfcc.org/, 2004.

[14] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Performance Evaluation of the Orca Shared-Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[15] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[16] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[17] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernels Internals*. Addison-Wesley, second edition, 1998.

[18] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.

[19] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, November 1994.

[20] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–955, 1998.

[21] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston Texas, April 1992.

[22] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[23] Michael G. Burke et al. The Jalapeno dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, 1999.

[24] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. HPJava: Data parallel extensions to Java. *Concurrency: Practice and Experience*, pages 873–877, 1998.

[25] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape Analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[26] Michal Cierniak and Wei Li. Briki: A flexible java compiler. Technical Report TR621, 1996.

[27] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/org/jvm98, 1998.

[28] Markus Dahm. Byte code engineering. In *JIT'99*, 1999.

[29] B. Dimitrov and V. Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5), 1998.

[30] EPCC. The java grande forum multi-threaded benchmarks. http://www.epcc.ed.ac.uk/javagrande/threads/contents.html.

[31] Michael Factor, Assaf Schuster, and Konstantin Shagin. JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogenous Collections of Comodity Workstations. In *IEEE Intl. Conference on Cluster Computing*, Hong Kong, December 2003.

[32] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software Practice and Experience*, 30(3):199–232, 2000.

[33] M. Fleury and F. Reverbel. The JBoss extensible server. In *Proceeding of International Middleware Conference*, 2003.

[34] The Apache Software Foundation. The jakarta site – apache tomcat. http://jakarta.apache.org/tomcat/, 2004.

[35] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[36] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. Pvm 3 user's guide and reference manual. Technical report, Oak Ridge Nat. Lab., Oakridge, Tennesee, 1994.

[37] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[38] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[39] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language.* Microsoft Corporation, 2004.

[40] U. Hlzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, USA, June 1992.

[41] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In *the 29th annual IEEE/ACM International Symposium on Microarchitecture*, 1996.

[42] K. Hwang, H. Jin, E. Chow, Cho-Li Wang, and Z. Xu. Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space. *IEEE Concurrency Magazine*, 7(1):60–69, January 1999.

[43] L. Iftode. *Home-based Shared Virtual Memory.* PhD thesis, Princeton University, August 1998.

[44] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Systems and Software*, 42(1):71–87, 1998.

[45] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computers: Technology, Architecture, Programming.* McGraw-Hill, New York, 1998.

[46] A Lempel and J Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22:75–81, 1976.

[47] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1986. ACM Press.

[48] T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification.* Addison Wesley, second edition, 1999.

[49] Snir M., Otto S.W., Huss-Lederman S., Walker D.W., and Dongarra J. *MPI –The Complete Reference.* The MIT Press, 1996.

[50] Carmine Mangione. Performance test show Java as fast as C++. Technical report, JavaWorld, 1998.

[51] Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. Delta Execution: A preemptive Java thread migration mechanism. *Cluster Computing*, 3(2):83–94, 2000.

[52] Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Parallel and Distributed Computing*, 60(10):1194–1222, October 2000.

[53] Satoshi Matsuoka, Hirotaka Ogawa, Kouya Shimura, Yasunori Kimura, Koichiro Hotta, and Hiromitsu Takagi. Openjit a reflective java jit compiler. In *OOPSLA '98*, 1998.

[54] Sun Microsystems. 100% pure java cookbook. http://java.sun.com/products/archive/100percent/, 2000.

[55] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[56] Nagendra Nagarajayya and J. Steven Mayer. Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1. Technical report, Sun Microsystems, 2002.

[57] OpenMP Architecture Review Board. OpenMP: A proposed industry standard api for shared memory programming. www.openmp.org, Oct. 1997.

[58] Robert Orfali and Dan Harkey. *Client/Server Programming with JAVA and CORBA*. John Wiley And Sons Inc., 2nd edition edition, 1998.

[59] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 129–140. ACM Press, 2002.

[60] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer Magazine of the*

Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314, 21(2), 1988.

[61] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *the 1st Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, Monterey, USA, April 2001.

[62] Michael Philippsen and Matthias Zenger. JavaParty — Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, Novermber 1997.

[63] Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, USA, 1997.

[64] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: User-level shared memory. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 325–337, 1994.

[65] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Joint Symposium on Agent Systems and Applications / Mobile Agents*, pages 16–28, 2000.

[66] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Operating Systems Design and Implementation*, pages 101–114, 1994.

[67] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39:175–193, 2000.

[68] Andrew S. Tanenbaum. *Distributed Operating Systems*. 1995.

[69] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[70] The System Research Group, The Department of Computer Science, University of Hong Kong. HKU Gideon Cluster. http://www.srg.csis.hku.hk/gideon/, 2004.

[71] TOP500.ORG. Top500 supercomputer sites. http://www.top500.org.

[72] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 29–43, 2000.

[73] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Jackal, A Compiler Based Implementation of Java for Clusters of Workstations. Technical report, University of Erlangen-Nurnberg, Germany, 2001.

[74] Ronald Veldema, Rutger F. H. Hofman, Raoul Bhoedjang, and Henri E. Bal. Runtime optimizations for a Java DSM implementation. In *Java Grande*, pages 153–162, 2001.

[75] Bruce Walker. Openssi (single system image) clusters for linux. http://openssi.org, 2005.

[76] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.

[77] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Lightweight Transparent Java Thread Migration for Distributed JVM. In *Inter-*

national Conference on Parallel Processing, pages 465–472, Kaohsiung, Taiwan, October 2003.

[78] Wenzhang Zhu, Cho-Li Wang, Weijian Fang, and Francis C. M. Lau. Jit-compiler-assisted distributed java virtual machine. In *The 10th Workshop on Compiler Techniques for High-Performance Computing*, Hsinchu Taiwan, Mar 2004.

[79] Wenzhang Zhu, Weijian Fang, Cho-Li Wang, and Francis C. M. Lau. A new transparent java thread migration system using just-in-time recompilation. In *The 16th IASTED International Conference on Parallel and Distributed Computing and System (PDCS2004)*, MIT Cambridge, MA, USA, Nov. 2004.

[80] T. Wilkinson. Kaffe - A Free Virtual Machine to run Java Code. http://www.kaffe.org/, 1998.

[81] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. Latte: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *International Conferenceon Parallel Architectures and Compilation Techniques*, pages 128–138, October 1999.

[82] Weimin Yu and Alan L. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.