1 High-Performance Computing on Clusters : The Distributed JVM Approach [†]

Wenzhang Zhu, Weijian Fang, Cho-Li Wang, and Francis C. M. Lau
The Department of Computer Science and Information Systems
The University of Hong Kong

A Distributed Java Virtual Machine (DJVM) is a cluster-wide virtual machine that supports parallel execution of a multithreaded Java application on clusters, as if it was executed on a single machine but with improved computation power. The DJVM hides the physical boundaries between the cluster nodes and allows parallelly executed Java threads to access all cluster resources through a unified interface. It is a more user-friendly parallel environment than many other existing parallel languages [8], or libraries for parallel programming such as MPI [13], CORBA [16], and Java RMI [7]. The DJVM research is valuable for high-performance computing as Java has become the dominant language for building the server-side applications, such as enterprise information systems, Web services, and large-scale Grid computing systems, due to its platform independency and built-in multithreading support at language level.

This chapter addresses the realization of a distributed Java virtual machine, named JESSICA2, on clusters. Section 1.1 describes Java, Java Virtual Machine, and the main programming paradigms using Java for high-performance computing. We then focus our study on the newly emerging distributed JVM research in Section 1.2. In Section 1.3, we introduce our JESSICA2 Distributed JVM. Section 1.4 gives the performance analysis of JESSICA2. Related work is given in Section 1.5. Section 1.6 concludes this chapter.

[†]**This research is supported by Hong Kong RGC Grant HKU-7030/01E and HKU Large Equipment Grant 01021001.**

1.1 BACKGROUND

1.1.1 Java

Java [11] is a popular general object-oriented programming language. Java supports concurrency through its multithreading framework. A Java program can have simultaneous control flows, i.e., threads, and all the threads share a common memory space. To avoid the race conditions among threads, Java includes a set of synchronization primitives based on the classic *monitor* paradigm. The *synchronized* keyword is used for declaring a critical section in Java source code.

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 block, and is released on exit. The lock is also associated with a wait queue. The class java.lang.Object provides three additional methods to control the wait queue within the synchronized methods or blocks, i.e., *wait*, *notify*, and *notifyAll*. The method *wait* causes current thread to wait in the queue until another thread invokes the *notify* method or the *notifyAll* method which wake up a single thread or all the threads waiting for this object respectively.

Each Java object consists of data and methods. The object has associated a pointer to a virtual method table. The table stores the pointers to their 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.

1.1.2 Java Virtual Machine

Unlike most of other programming languages, usually a Java source program is not directly compiled into native code running on the specific hardware platform. Instead, the Java compiler will 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 Java Virtual Machine (JVM) is then used to load and execute the compiled Java bytecode. The bytecode is then interpreted or translated by the JVM execution engine. The JVM provides 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 of source code.

The JVM is a stack-oriented and multithreaded virtual machine. Figure 1.1 illustrates the architecture of JVM. Inside a JVM, each thread has a runtime data structure called Java stack to hold the *program counter* (PC) and the local variables. The threads create Java objects in the centralized garbage-collected heap and refer

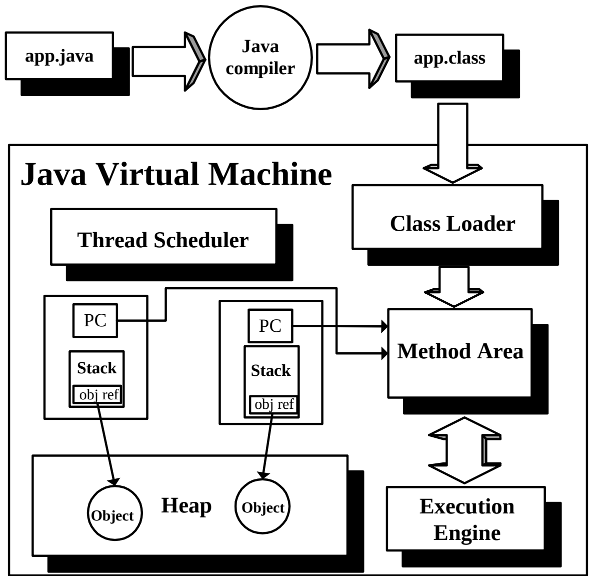


Fig. 1.1 The architecture of Java Virtual Machine.

to the objects using object references in the Java stack. All the created objects are visible to all the threads.

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 the call to the interpreter with the method id as the argument. The interpreter will create the data structures for the Java method and try to simulate the semantics of bytecode instructions by operating on these data structures in a big loop.

The interpreter is simple to implement. However, such interpretation is slow because it can not efficiently use the machine registers for computation. And it can not cache the previous computed results such as the constant offset to an object. To boost the Java execution performance, the concept of Just-in-Time (JIT) compilation is introduced.

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 each 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 time for JIT compilation is charged to the execution time of Java program, the JIT compiler requires the lightweight compilation techniques. Usually a JIT compiler can improve the performance of Java application by a factor.

1.1.3 Programming Paradigms for Parallel Java Computing

Several programming paradigms exist in the parallelization of applications. Generally we have three major paradigms, namely, *data parallel*, *message passing*, and *shared memory*. To support these paradigms in Java, many libraries and runtime systems have been proposed since Java was born in 1995.

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

One example to support the data parallel programming paradigm is the HPJava language [8]. It extends ordinary Java with some shorthand syntax for describing how arrays are distributed across processes. HPJava has no explicit interfaces for communication among processes. The communication among processes is handled transparently by the HPJava compiler.

The shortage 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. However, 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.

1.1.3.2 Message passing Message passing is probably the most common paradigm for parallel programming on the clusters. In this paradigm, the programmers write explicit code to send and receive messages for the communication and coordination among different 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), the Java *Remote Method Invocation* (RMI) [7] and *Common Object Request Broker Architecture* (COBRA) [16].

MPI is a widely accepted interface standard for communication. One implementation of MPI on Java is the mpiJava [4] library. It enables the communication of Java programs by introducing a new class called *MPI*. Using mpiJava, the programmers need to handle data communication explicitly that usually is a complicated and error-prone task.

Java RMI is similar to the 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. A RMI server application creates some objects, and publish them for the remote RMI clients to invoke methods on these objects. 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 Interface Description Language (IDL) to enable the interaction between Java programs and the CORBA-compliant distributed applications widely deployed on the Internet. The shortage of COBRA is that it is difficult for programmers to master.

1.1.3.3 Shared memory The shared memory paradigm assumes a shared memory space among the cooperative computation tasks. The multithreading feature of

Java fits this paradigm well in a single-node environment. However, the current standard JVM can only achieve limited parallelism of multithreaded programs even the machine on which it runs is an SMP machine.

1.2 DISTRIBUTED JVM

A Distributed Java Virtual Machine (DJVM) is a cluster-wide virtual machine that supports the parallel execution of threads inside a multithreaded Java application with *single-system image* (SSI) illusion on clusters. In this way, the multithreaded Java application runs on a cluster as if it ran on a single machine with improved computation power. As a result, DJVM supports the shared memory programming paradigm. Several approaches have been proposed for developing the distributed JVM [3, 14, 2, 25]. In this chapter, we focus our study on the research of distributed JVM in the following sections.

Such research is valuable for high-performance computing. Java provides a highly portable language environment and a simple thread model, thus a DJVM can provide a more portable and more user-friendly parallel environment than many other existing parallel languages, or libraries for parallel programming, as discussed in the Section 1.1.3.

1.2.1 Design issues

Building a DJVM on cluster poses a number of challenges.

- *Java thread scheduling.* The thread scheduler basically decides which thread to grasp the CPU and switches thread contexts inside the virtual machine. The scheduling of Java threads on clusters requires a nontrivial design of the virtual machine's kernel, so that the Java threads can be transparently deployed on different cluster nodes and run in parallel efficiently. Inefficient scheduling can lead the system to an imbalanced state which results in poor execution performance. It is therefore desirable to provide mechanisms for load balancing in DJVM.

- *Distributed shared heap.* The heap is the shared memory space for Java threads to store the created objects. The separation of memory spaces among cluster nodes is conflicting with the shared memory view of Java threads. Such 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.

- *Execution engine.* The JVM execution engine is the processor of Java bytecode. To make a high-performance DJVM, it is necessary to have a fast execution engine. Therefore the execution of Java threads in native code is a must. As the threads and heap are distributed, the execution engine needs to be extended to be “cluster-aware”, i.e., it should be able to choose appropriate actions for local and remote objects.

1.2.2 Solutions

In our design for distributed thread scheduling, we do not choose the simple initial thread placement, which simply spawns a thread remotely upon thread creation as used in related projects [3, 19, 2]. Thought it can achieve the efficient parallel execution for threads with balanced workload, it is not enough for a wide range of applications that exhibit significant imbalanced computation workload among threads. To provide advanced support for load balancing, we seek for a lightweight and transparent Java thread migration mechanism.

The design of heap actually provides distributed shared object services like what a multithreaded software DSM does. And there do exist a number of DJVMs [14, 25, 2] that are directly built on top of an unmodified software Distributed Shared Memory (DSM). This approach simplifies the design and implementation of a DJVM as it only needs to call the APIs of the DSM to realize the heap. However, it is far from an efficient solution since such a layered design will pose significant overheads in the interactions between the JVM and the DSM due to the mismatch of the memory model of Java and that of the underlying DSM. Moreover, the runtime information at the JVM level such as the object type information cannot be easily channelled to the DSM. Also the off-the-shelf DSM is difficult to be extended for supporting other services like the SSI view of I/O objects. In our system, we instead go for a built-in distributed shared object technique that realizes the JMM. This approach can make use of the runtime information inside the DJVM to reduce the object access overheads as it is tightly coupled with the DJVM kernel.

For the execution engine, we adopt the JIT compiler as the execution engine. A Java bytecode interpreter is relatively simple. 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. Static compilers, as used in Hyperion [2] and Jackal [19], although can achieve high-performance native execution of Java threads, usually miss the dynamic JVM functionalities such as loading new Java classes from remote machine during runtime. The mixed-mode execution engine, which is first introduced in Sun’s hotspot compiler, is much more complex to be adopted in the DJVM design. Dynamic or JIT compilers are rarely considered or exploited in previous DJVM projects. The JIT compiler is relatively simpler than the full-fledged static compilers, but it can still achieve high execution performance. Therefore we believe that it is the best choice for DJVM.

1.3 JESSICA2 DISTRIBUTED JVM

1.3.1 Overview

The overall architecture of our DJVM JESSICA2 is shown in Figure 1.2. It runs in a cluster environment and consists of a collection of modified JVMs that run in different cluster nodes and communicate with each other using TCP connections.

The class files of the compiled multithreaded Java program can be directly run on JESSICA2 without any preprocessing. We call the node that starts the Java program *master node* and the JVM running on it *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 Java thread can be dynamically scheduled by the thread scheduler to migrate during runtime in order to help achieve a balanced system load throughout. Being transparent, the migration operation is done without explicit migration instructions inserted in the source program.

In a distributed JVM, the shared memory nature of Java threads calls for a *global object space* (GOS) that “virtualizes” a single Java object heap spanning the entire cluster to facilitate transparent object accesses. In the distributed JVM, each node can dynamically create new objects, and manage them independently. The objects created at the remote nodes can be replicated to improve the access locality. Concurrent accesses on the copies of shared objects are also allowed, thus raising the issue of memory consistency. The memory consistency semantics of the GOS are defined based on the Java memory model (Chapter 8 of the JVM specification [12]).

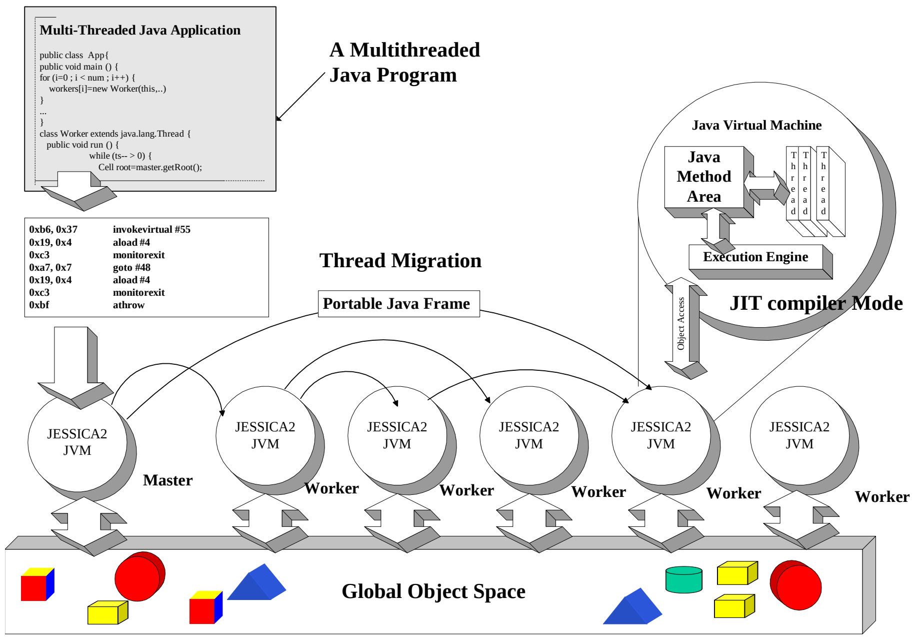
There is no assumption of a shared file system in the implementation. The application class files can be duplicated in each node, or they can be stored only in the master node. In the latter case, when a worker JVM can not find a class file locally, it will request the class bytecode from the master 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, the modified JVM class loader will first query the master JVM to check if it has been loaded and initialized. All such queries from different JVMs will be sequentialized. If the initialization of the class has been done, the worker JVM will fetch its static data , and copy them into local static data area.

Our system does not rely on a single-shared distributed file systems such as NFS, nor does it need to restrict a single IP address for all the nodes in the running cluster. The system has built the I/O redirection functionalities inside to enable the SSI view of the file I/O operations and the networking I/O operations.

The following sections discuss our implementation of JESSICA2 [22, 23, 20].

1.3.2 Global object space

We follow the Java memory model (JMM) to solve the memory consistency issue in GOS. We also incorporate several optimization techniques to further improve the performance of GOS by exploiting the runtime information inside JVM.



```
public class  App{
public void main () {
for (i=0 ; i < num ; i++) {
   workers[i]=new Worker(this,..)
}
...
}
class Worker extends java.lang.Thread {
  public void run () {
                while (ts-- > 0) {
                    Cell root=master.getRoot();
```

```
0xb6, 0x37      invokevirtual #55
0x19, 0x4       aload #4
0xc3            monitorexit
0xa7, 0x7       goto #48
0x19, 0x4       aload #4
0xc3            monitorexit
0xbf            athrow
```

Fig. 1.2 The JESSICA2 system overview.

1.3.2.1 Implementing Java memory model According to the Java memory model, the synchronization operations in Java are used not only to guarantee exclusive access in the critical section, but also to maintain the consistency of objects among all threads that have performed synchronization operations on the same lock.

We follow the operations defined in the JVM specification to implement this memory model. Before a thread releases a lock, it must copy all assigned values in its private working memory back to the heap which is shared by all threads. Before a thread acquires a lock, it must flush (invalidate) all variables in its working memory; and later uses will load the values from the heap.

Figure 1.3 shows all the memory access operations in the GOS. In the JVM, the *connectivity* exists between two Java objects if one object contains a reference to another. Based on the connectivity, we divide Java objects in the GOS into two categories: *distributed-shared objects* (DSOs) and *node-local objects* (NLOs). Distributed-shared objects are reachable from at least two threads in different cluster nodes in the distributed JVM, while node-local objects are reachable from only one cluster node. Distributed-shared objects can be distinguished from node-local objects (by default) at runtime [21].

We adopt a home-based multiple writer cache coherence protocol to implement Java memory model. Each shared object has a home from which all copies are derived

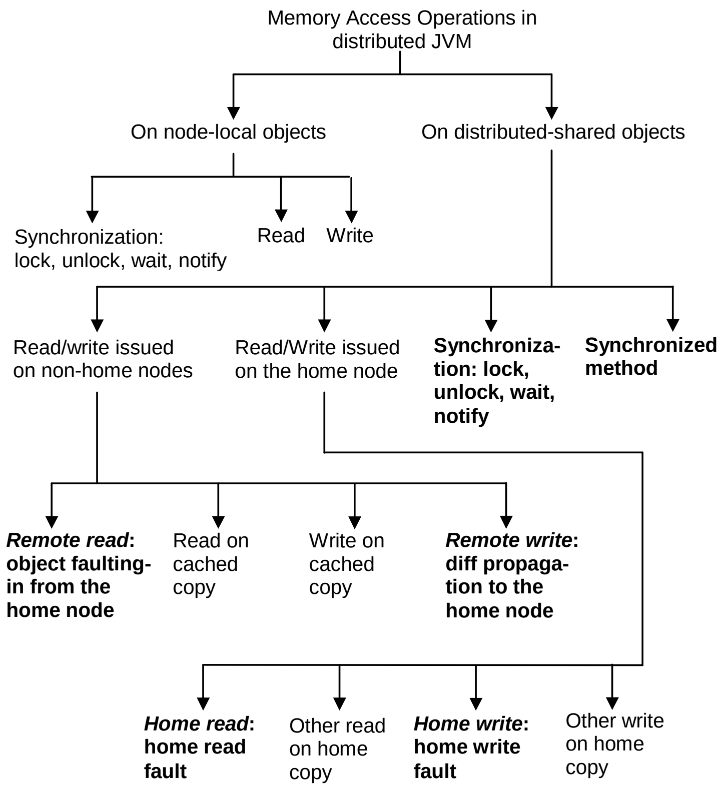

Memory Access Operations in distributed JVM
On node-local objects
On distributed-shared objects
Synchronization: lock, unlock, wait, notify
Read
Write
Read/write issued on non-home nodes
Read/Write issued on the home node
Synchronization: lock, unlock, wait, notify
Synchronized method
Remote read: object faulting-in from the home node
Read on cached copy
Write on cached copy
Remote write: diff propagation to the home node
Home read: home read fault
Other read on home copy
Home write: home write fault
Other write on home copy

Fig. 1.3 Memory access operations in GOS

and to which all writes (diffs) are propagated. The access events of a distributed-shared object comprise those on non-home nodes and those on the home node. On non-home nodes, after acquiring a lock, the first read should fault in the object from its home. All the subsequent reads or writes can be performed on the local copy. Before acquiring or releasing a lock, the locally performed writes should be identified using *twin* and *diff* techniques and sent to the home node. We call the object faulting-in *remote read*, and the diff propagation *remote write*.

In order to facilitate some optimizations such as object home migration that will be discussed later, we also monitor the access operations on the home node. On the home node, the access state of the home copy will be set to invalid on acquiring a lock and to read-only on releasing a lock. *Home read fault* and *home write fault* will be trapped. For both types of fault, the GOS does nothing more than to set the object to the proper access state. We call the home read fault *home read*, and the home write fault *home write*.

All the synchronization operations performed on a distributed-shared object, such as lock, unlock, wait, and notify, influence the object access behavior, and are thus considered access events too. The synchronized method is treated as a special access event.

1.3.2.2 Optimizations Since our GOS is built-in component inside the distributed JVM, we are able to effectively calibrate the runtime memory access patterns and dynamically apply optimized cache coherence protocols to minimize consistency maintenance overhead. The optimization devices include an *object home migration* method that optimizes the single-writer access pattern, *synchronized method migration* that allows the execution of a synchronized method to take place remotely at the home node of its locked object, and *connectivity-based object pushing* that uses object connectivity information to perform pre-fetching.

Object home migration In a home-based cache coherence protocol, the home node of a DSO plays a special role among all nodes holding a copy. Accesses happening in the non-home nodes will incur communication with the home node, while accesses in the home node can proceed in full speed.

Our GOS is able to determine a better location of the home of an object and perform object home migration accordingly. we choose to only apply object home migration to those DSOs exhibiting the single-writer access pattern, where the object is only written by one node, to reduce home migration notices that are used to notify of the new home. In the situation of multiple-writer pattern where the object is written by multiple nodes, it does not matter which is the home node as long as the home node is one of the writing nodes.

In order to detect the single-writer access pattern, the GOS monitors all home accesses as well as non-home accesses at the home node. To minimize the overhead in detecting the single-writer pattern, the GOS records consecutive writes that are from the same remote node and that are not interleaved by the writes from other nodes. We follow a heuristic that an object is in the single-writer pattern if the number of consecutive writes exceeds a predefined threshold.

Synchronized method migration The execution of a synchronized method of a DSO not at its home will trigger multiple synchronization requests to the home node. For example, on entering and exiting the synchronized method, the invoking node will acquire and then release the lock of the synchronized object. Memory consistency maintenances are also involved according to Java memory model. Migrating a synchronized method of a DSO to its home node for execution will combine multiple round-trip messages into one and reduce the overhead for maintaining memory consistency. While object shipping is the default behavior in the GOS, we apply method shipping particularly to the execution of synchronized methods of DSOs.

Connectivity-based object pushing Object pushing is a pre-fetching strategy which takes advantage of the object connectivity information to more accurately pre-store the objects to be accessed by a remote thread, therefore minimizing the network delay in subsequent remote object accesses. Connectivity-based object pushing actually improves the reference locality. The *producer-consumer* pattern is one of the patterns that can be optimized by connectivity-based object pushing [21].

Object pushing is better than pull-based pre-fetching which relies on the requesting node to specify explicitly which objects to be pulled according to the object connectivity information. A fatal drawback of pull-based pre-fetching is that the connectivity information contained in an invalid object may be obsolete. Therefore, the pre-fetching accuracy is not guaranteed. Some unneeded objects, even garbage objects, may be pre-fetched, which will end up wasting communication bandwidth. On the contrary, object pushing gives more accurate pre-fetching results since the home node has the up-to-date copies of the objects and the connectivity information in the home node is always valid.

In our implementation, we rely on an optimal message length, which is the preferred aggregate size of objects to be delivered to the requesting node. Reachable objects from the requested object will be copied to the message buffer until the current message length is larger than the optimal message length. We use a breadth-first search algorithm to select the objects to be pushed.

1.3.3 Transparent Java thread migration

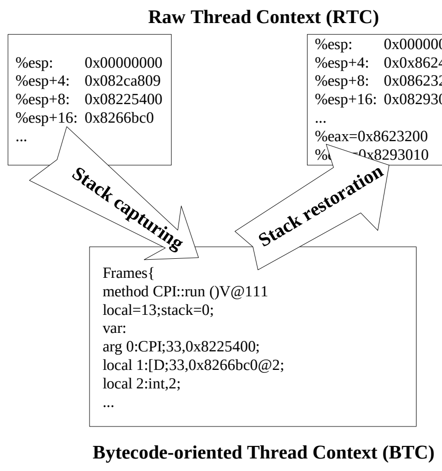
One of the unique features of our system is that we support the dynamic transparent migration of Java threads. This section describes our lightweight and efficient solution for thread migration in the context of JIT compiler.

Transparent thread migration has long been used as a load balancing mechanism to optimize the resource usage in distributed environments [10]. Such systems usually use the *raw thread context* (RTC) as the communication interface between the migration source node and target node. RTC usually includes the virtual memory space, thread execution stack and hardware machine registers.

We adopt the *bytecode-oriented thread context* (BTC) to make the system more portable. 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 program counter (PC), JVM operand stack pointer, operand stack variables, and the local variables, all encoded in a JVM-independent format.

In a JIT-enabled JVM, the JVM stack of a Java thread becomes native stack and no longer remains bytecode-oriented. We solve the transformation of the RTC into the BTC directly inside the JIT compiler. Our solution is built on two main functions, *stack capturing* and *stack restoration* (see Figure 1.4). Stack capturing is to take a snapshot of the RTC of a running Java thread and transforms the snapshot into an equivalent BTC. Stack restoration is to re-establish the RTC using the BTC. Such a process via an intermediate BTC takes advantage of the portability of the BTC.

1.3.3.1 Stack capturing To support the RTC-BTC transformation, we perform Just-in-Time native code instrumentation inside the JIT compiler. We insert additional optimized native code in a method when it is first compiled. The instrumented code enables the running thread to manage its own context in a reflexive way. During execution, the thread maintains the execution trace of the Java thread in some lightweight runtime data structures. Figure 1.5 shows the working flow of the JIT compiler in our system.



```
%esp:      0x00000000
%esp+4:    0x082ca809
%esp+8:    0x08225400
%esp+16:  0x8266bc0
...
```

```
%esp:      0x00000000
%esp+4:    0x0x86243c
%esp+8:    0x08623200
%esp+16:  0x08293010
...
%eax=0x8623200
%[illegible]0x8293010
```

```
Frames{
method CPI::run ()V@111
local=13;stack=0;
var:
arg 0:CPI;33,0x8225400;
local 1:[D;33,0x8266bc0@2;
local 2:int,2;
...
```

Fig. 1.4 The thread stack transformation. The frame for method run() of class CPI is encoded in a text format in the BTC box.

In the instrumentation of JIT compiler, we limit the migration to take place at some specific points called *migration points*. We choose two types of points in our system. The first type (referred as M-point) is the site that invokes a Java method. The second type (referred as B-point) is the beginning of a bytecode basic block pointed by a back edge, which is usually the header of a loop.

At such points, the instrumented native code is used to check the migration request and to spill the machine registers to the memory slots of the variables. For the variable types, we use *type spilling* to store the variable types at the migration points. The type information of stack variables will be gathered at the time of bytecode verification. We use one single type to encode the reference type of stack variable as we can identify the exact type of a Java object from the object reference. Therefore, we can compress one type into 4-bit data. Eight compressed types will be bound in a 32-bit machine word, and an instruction to store this word will be generated to spill the information to appropriate location in the current method frame. For typical Java methods, only a few instructions are needed to spill the type information of stack variables in a method.

The Java frames in the stack are linked by the generated native code. The code only needs a few instructions to spill the previous Java frame stack pointer and previous machine stack pointer. Such arrangement makes it possible to tell a Java frame from

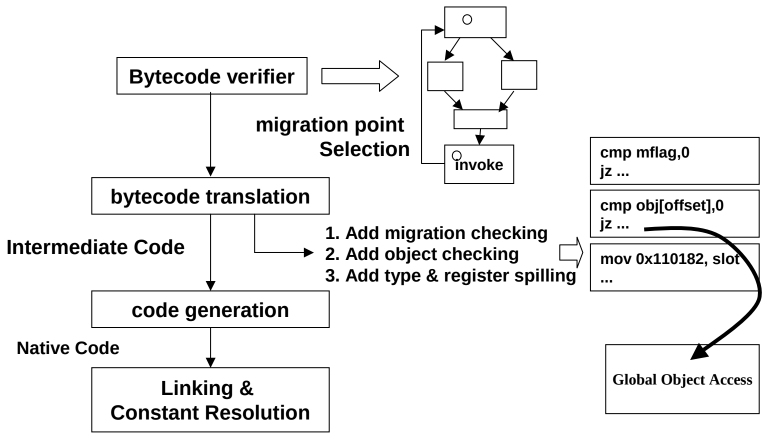


Fig. 1.5 The working flow of the JIT compiler. The native instruction "cmp mflag, 0" checks the migration flag. The instruction "cmp obj[offset],0" is for checking object status. The instruction "mov 0x110182, slot" stores the variable types in the memory slot. The "Global Object Access" is the GOS layer.

the internal JVM functions frames (we call it C frames). In our thread migration, we choose the consecutive Java frames to be migrated to the remote machine. Upon completion of such Java frames, the control will return back to the source machine to complete the C frame execution.

1.3.3.2 Stack restoring 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 create a new native thread. The newly created thread becomes the clone of the migrated thread in current JVM. The clone thread then brings back the calling sequence as described by the input context. In our system, we build a sequence of stack frames with the return addresses and the frame pointers properly linked together to simulate the method invocation. The local variable inside the frames will be initialized to the values according to the input thread context.

The stack restoring needs to recover the machine registers in the migration target node. Most previous approaches supporting thread stack restoring often build the stack by simulating the method invocation and use additional status variables to distinguish the restoring execution flow and the normal execution flow inside the methods [18]. This will results in large overheads because it needs to add such branching codes in all the methods. Rather we directly build the thread stack and use recompilation techniques to get the mapping between the thread variables and the machine registers at all restoration points. Each mapping is then used by a generated code stub that will be executed before the restoration point to recover the machine registers. In our implementation, we allocate the code stubs inside the restored thread stack so that they will be freed automatically after execution.

1.3.3.3 Load balancing policy We have integrated the load balancing policy in our current DJVM so that it is responsive for thread scheduling. The policy adopts a scheme similar to the work stealing [6]. A lightly loaded JVM will try to acquire computation threads from other heavily loaded nodes periodically. The load information such as the CPU and memory usages is stored on the master node. 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. Subsequent thread migration operation 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. It will switch the state to heavy load after it stays at the state of heavy load for a time period. Then the discharging begins by migrating threads to lightly loaded nodes.

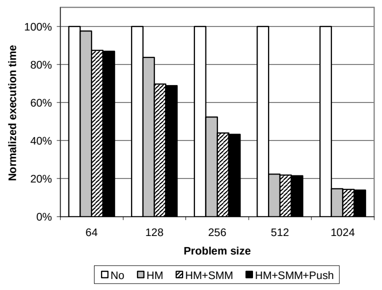
The master node will not be a bottleneck caused by the load information because only those worker nodes that have radical load changes (from heavy load state to light load state or vice versa) will send the messages to it.

1.4 PERFORMANCE ANALYSIS

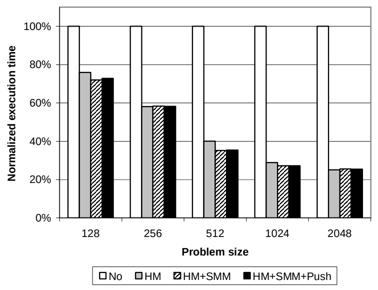
Our distributed JVM, JESSICA2, is developed based on Kaffe open JVM 1.0.6 [24]. We run JESSICA2 on the HKU Gideon 300 Linux cluster to evaluate the performance. Each cluster node consists of 2GHz Pentium 4 CPU, 512M RAM and runs Linux kernel 2.4.14. The cluster is connected by a Foundry Fastiron 1500 Fast Ethernet switch.

The following benchmarks are used in our experiments.

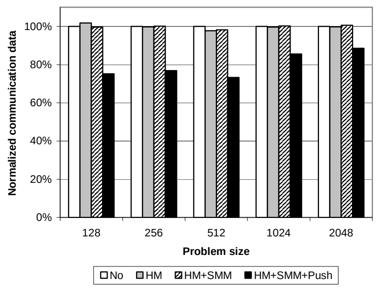
- CPI calculates an approximation of π by evaluating the integral.
- ASP (All-Shortest Path) computes the shortest paths between any pair of nodes in a graph using a parallel version of Floyd's algorithm.
- TSP (Travel Salesman Problem) finds the shortest route among a number of cities using a parallel branch-and-bound algorithm.
- Raytracer renders a 3-D scene by using the raytracing method.
- SOR (Successive Over-Relaxation) performs red-black successive over-relaxation on a 2-D matrix for a number of iterations.



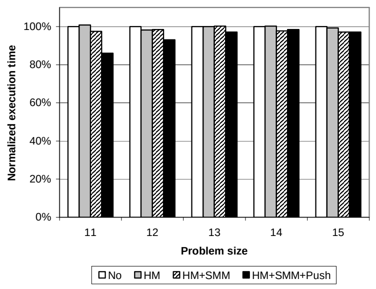
(a) ASP



(b) SOR



(c) Nbody



(d) TSP

Fig. 1.6 Effects of adaptations w.r.t. execution time

- Nbody simulates the motion of particles due to gravitational forces between each other over a number of simulation steps using the algorithm of Barnes & Hut.

- SPECjvm98 [9] is the benchmark suite used for testing JVM's performance.

1.4.1 Effects of optimizations in GOS

In the experiments, all adaptations are disabled initially; and then we would enable the planned adaptations incrementally. Fig. 1.6 shows the effects of adaptations on the execution time. We present the normalized execution time against different problem sizes.

Our application suite consists of four multi-threaded Java programs, namely ASP, SOR, Nbody, and TSP. In ASP, we scale the size of the graph; in SOR, we scale the size of the 2-D matrix; in Nbody, we scale the number of the bodies; in TSP, we scale the number of the cities. All data are normalized to that when none of the adaptations are enabled. All tests run on 16 processors. In the legend, "No" denotes no adaptive protocol enabled, "HM" denotes object home migration, "SMM" denotes synchronized method migration, and "Push" denotes object pushing. All the application are running in the interpreter mode of JVM.

As can be seen in the figures, object home migration greatly improves the performance of ASP and SOR. In ASP and SOR, the data are in the 2-D matrices that are shared by all threads. In Java, a 2-D matrix is implemented as an array object whose elements are also array objects. Many of these array objects exhibit the single-writer access pattern after they are initialized. However, their original homes are not the writing nodes. Object home migration automatically makes the writing node the home node in order to reduce communication. We can see that object home migration dramatically reduces the execution time. Also the effect of object home migration is amplified when the problem size is scaled up in ASP and SOR. In Nbody and TSP, the single-writer access pattern is insignificant, and therefore the effect of object home migration cannot be obviously observed.

Synchronized method migration optimizes the execution of a synchronized method of a non-home DSO. Although it does not reduce the communication volume, it reduces the number of messages significantly. We also observe in Fig. 1.6(a) and (b) that synchronized method migration improves ASP and SOR's overall performance to some extent, particularly when the problem size is small. ASP requires n barriers for all the threads in order to solve an n-node graph. The synchronization is quite heavy in ASP. So synchronized method migration has more positive effect on ASP. When the problem size is scaled up, the communication-to-computation ratio decreases, thus the adaptation effect becomes not so evident. The synchronization overhead comprises not only the processing and transmission time, but also the waiting time. Sometimes the synchronization overhead is dominated by the waiting time, which cancels out the benefit from synchronized method migration. Nbody's synchronization uses synchronized block instead of synchronized method, and so synchronized method migration has no effect here. TSP's situation is similar to Nbody's.

Connectivity-based object pushing is a pre-fetching strategy which takes advantage of the object connectivity information to improve reference locality. Particularly, it improves the producer-consumer pattern greatly. Nbody is a typical application of the producer-consumer pattern. In Nbody, a quadtree is constructed by one thread and then accessed by all other threads in each iteration. The quadtree consists of a lot of small-sized objects. We can see that object pushing greatly reduces the execution time in Nbody, as seen from Fig. 1.6(c). However, when the problem size is scaled up, the communication-to-computation ratio decreases, thus the effect of object pushing decreases. Notice that communication is relatively little in TSP, the improvement on the total execution time due to this optimization is limited. Compared with Nbody and TSP, most DSOs in ASP and SOR are array objects, and object pushing is not performed on them to reduce the impact of pushing unneeded objects.

1.4.2 Thread migration overheads

We first test the space and time overheads charged to the execution of Java threads by the JIT compiler when enabling the migration mechanism. Then we measure the latency of one migration operation.

The time overheads are mainly due to the checking at the migration points; and the space overheads are mainly due to the instrumented native code. We do not require the benchmarks to be multithreaded in the test since the dynamic native code instrumentation will still function even on single-threaded Java applications.

We use SPECjvm98 benchmark in the test. The initial heap size was set to 48MB. 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.

Table 1.1 shows the test results. The space overheads are in terms of the average size of native code per bytecode instruction, i.e., it is the blowup of the native code compiled from the Java bytecode.

From the table we can see that the average time overheads charged to the execution of Java thread with thread migration are about 2.21% and the overheads of generated native code are 15.68%. Both the time and space overheads are much smaller than the reported results from other static bytecode instrumentation approaches. For example, JavaGoX [17] reported that for four benchmark programs (Fibo, qsort, nqueen and compress in SPECjvm98), the additional time overheads range from 14% to 56%, while the additional space costs range from 30% to 220%.

Table 1.1 The execution overheads using SPECjvm98 benchmarks.

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%)

We also measured the overall latency of a migration operation using different multithreaded Java applications including a latency test (LT) program, CPI, ASP, Nbody, and SOR. The latency measured includes the time from the point of stack capturing to the time when the thread has finished its stack restoration on the remote node and has sent back the acknowledgement. CPI only needs 2.68 ms to migrate and restore thread execution because it only needs to load one single frame and one Java class during the restoration. LT and ASP need about 5 ms to migrate a thread context consisting of one single frame and restore the context. Although they only have one single frame to restore, they both need to load two classes inside their frame contexts. For SOR which migrates two frames, the time is about 8.5 ms. For NBody, which needs to load four classes in 8 frames, it takes about 10.8 ms.

1.4.3 Application benchmark

In this section, we report the performance of four multi-threaded Java applications on JESSICA2. The applications are CPI, TSP, Raytracer, and Nbody.

We run TSP with 14 cities, Raytracer within a 150x150 scene containing 64 spheres, Nbody with 640 particles in 10 iterations. We show the speedups of CPI, TSP, Raytracer and Nbody in Figure 1.7 by comparing the execution time of JESSICA2 against that of Kaffe 1.0.6 (in a single-node) under JIT compiler mode. From the figure, we can see nearly linear speedup in JESSICA2 for CPI, despite the fact that all the threads needed to run in the master JVM for 4% of the overall time at the very beginning. For the TSP and Raytracer, the speedup curves show about 50% to 60% of efficiency. Compared to the CPI program, 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 Raytracer the number of messages is small, as it only needs to transfer the scene data to the worker thread in the initial phase. The slowdown comes from the object checking in the modified JVM as the application accesses the object fields extensively in the inner loop to render the scene. But for the Nbody program, the speedup is only 1.5 for 8 nodes. The poor speedup is expected, which is due to the frequent communications between the worker threads and the master thread in computing the Barnes-Hut Tree.

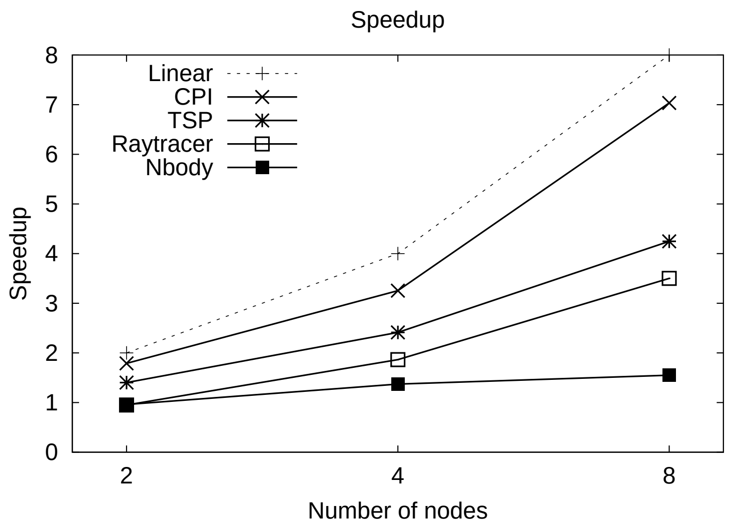


Fig. 1.7 Speedup Measurement of Java applications

1.5 RELATED WORK

Our system was motivated by recent work on distributed JVM, particularly cJVM [3] and JESSICA [14]. However these systems mainly base on slow Java interpreters. Our work distinguishes itself from these projects in that we use JIT compilers for the construction of the DJVM. Besides the relationship to the distributed JVM, our system also relates to techniques of software Distributed Shared Memory and computation migration. These techniques are exploited in our system in the context of JIT compilers.

1.5.1 Software Distributed Shared Memory

The software Distributed Shared Memory (DSM) has been studied extensively during the past decade. Orca [5] is one object-based DSM that uses a combination of compile-time and runtime techniques to determine the placement of objects. Our GOS differs from Orca in that we provide the shared object abstraction supports all at runtime through the JVM JIT compiler.

TreadMarks is a page-based DSM [1] 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 overheads of software checking on object status. One of the drawbacks, however, is that the page-based DSM will have the problem of false sharing if directly applied to an object-based language such as Java.

1.5.2 Computation migration

The research on computation migration has been studied for many years. Process migration can be regarded as the ancestor of thread migration. The paper [15] reviews the field of process migration till 1999. It provides detail analysis on the benefits and drawbacks of process migration. The systems included in the paper range from user-level migration systems to kernel-level migration ones. Compared to the existing computation migration techniques, we try to solve the computation migration from the new perspective by introducing the Just-in-Time compilation.

There are systems developed to support thread migration. Arachne [10] is one of such systems. It provides a portable user-level programming library that supports thread migration over a heterogeneous cluster. However the thread migration is not transparent to the user as it required that programs be written using special thread library or APIs.

There are related systems in the mobile computing area that support the mobility of Java threads. For example, JavaGoX [17] and and Brakes [18] all use the static preprocessor to instrument Java bytecodes to support the migration of Java thread. These systems do not address the distributed shared object issues.

1.5.3 Distributed JVM

cJVM [3] is a cluster-aware JVM that provides SSI of a traditional JVM running on cluster environments. The cJVM prototype was implemented by modifying the Sun JDK1.2 interpreter. cJVM does not support thread migration. It distributes the Java threads at the time of thread creation.

There are other DSM-based DJVM prototypes, for example, JESSICA [14] and Java/DSM [25]. Both systems are based on interpreters. JESSICA supports thread migration by modifying the Java interpreters. Java/DSM lacks supports for the location transparency of Java threads. It needs programmers' manual coding to place the threads on different cluster nodes.

Jackal [19] and Hyperion [2] adopt the static compilation approaches to compile the Java source code directly into native parallel code. The parallel code is linked to some object-based DSM library packages.

1.6 SUMMARY

Java is becoming an important vehicle for writing parallel programs due to its built-in concurrency support and high portability. To support the high-performance Java computing on clusters, there exist three main paradigms: data parallel, message passing and shared memory. The support for shared memory for Java inspires the new research on distributed JVM (DJVM) which aims to extend the single-node JVM to clusters for achieving high-performance multithreaded Java computing without the need for introducing new APIs. The multithreaded Java programs running on a DJVM is written in usual manners. The underlying DJVM tries to hide all the physical machine boundaries for the Java application transparently.

This chapter focuses on the study of supporting shared memory paradigm on clusters using DJVM. We use our prototype system JESSICA2 as an example to discuss the design, implementation, and performance analysis of a DJVM. The design of a DJVM needs to provide a virtually shared heap for the Java threads inside one single application. Due to Java's memory model constraints, the existing consistency protocols of software Distributed Shared Memory can not match well with that of Java. Incorporating the distributed shared object support inside DJVM is worthy of the efforts for achieving the improved communication performance. Another important aspect of a DJVM is the scheduling of threads. We believe that a lightweight thread migration mechanism can help balance the workload among cluster nodes especially for irregular multithreaded applications where the workload can not be simply estimated and equally partitioned among the threads.

References

1. 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.

2. Gabriel Antoniu et al. The Hyperion System: Compiling Multi-threaded Java Bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

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

10. 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.

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

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

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

14. 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.

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

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

17. 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.

18. 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.

19. 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.

20. Weijian Fang, Cho-Li Wang, and Francis C.M. Lau. Efficient Global Object Space Support for Distributed JVM on Cluster. In *The 2002 International Conference on Parallel Processing (ICPP-2002)*, pages 371–378, British Columbia, Canada, August 2002.

21. Weijian Fang, Cho-Li Wang, and Francis C.M. Lau. On the Design of Global Object Space for Efficient Multi-threading Java Computing on Clusters. In Parallel Computing, Vol.29, pp. 1563-1587, 2003.

22. 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.

23. Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Lightweight Transparent Java Thread Migration for Distributed JVM. In *International Conference on Parallel Processing*, pages 465–472, Kaohsiung, Taiwan, October 2003.

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

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

26. Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape Analysis for Java. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1999.