JESSICA: Java-Enabled Single-System-Image Computing Architecture*

Matchy J. M. Ma Cho-Li Wang Francis C. M. Lau

Department of Computer Science and Information Systems
The University of Hong Kong, Hong Kong

April 1999

Abstract

JESSICA stands for 'Java-Enabled Single-System-Image Computing Architecture'. It is a middle-ware that makes a cluster of computers to appear as a single, multi-processor machine to Java applications. The single system encapsulation allows resources that are distributed around the cluster to be readily shared by applications independent of their physical locations. It supports load distribution and balancing by offering a parallel execution environment where threads can freely migrate across machine boundaries to exploit the maximal parallelism obtainable from the cluster. Consequently, resource utilization around the cluster can also be optimized. Preemptive thread migration is supported by a novel approach called Delta Execution, where only the machine-independent part of a thread's execution context is migrated strategically. In addition, Location transparency is upheld by employing a redirection mechanism which forward any location-dependent operations back to the home machine for execution. A working prototype has been implemented on a 12-node cluster and considerable speedup has been achieved on all the experimental applications we tested.

1 Introduction

The subject of cluster computing has been under active research in recent years. A cluster of computers is a federation of computers linked by an interconnection network where the computers are running integration software for performing collaborative computation. Many promising results have been reported on using clusters of computers for load sharing and parallel computing [12, 16]. With the advent of high-speed networking and microprocessor technologies, cluster computing has emerged as a favorable alternative to massively parallel machines for high performance and large-scale computing.

In order to span multiple machines and achieve true parallelism, Java programmers currently have to tackle the coordination between processing nodes at the application level, through some IPC mechanisms such as socket. Since the introduction of JDK version 1.1,

*This research was supported by the Hong Kong RGC Grant HKU 7032/98E and HKU Equipment Grant 10003.01991001.00000.14200.100.01.

the burden on the programmers is alleviated by the provision of Object Serialization [9], Remote Method Invocation (RMI) [10] and the Object Request Broker (ORB) [8] supports. They allow coordination and cooperation of processes at the function call level, through some remote procedure call (RPC) like mechanisms as illustrated in Fig. 1a. Nonetheless, programmers still have to worry about the availability of the processing nodes involved, as the usability of a distributed application depends on the nodes' availability. Moreover, parallel programming in this paradigm is still not as straightforward as one could do when using a multi-threaded model.

This paper introduces JESSICA, our solution to the issue mentioned above by providing a parallel execution environment for multi-threaded Java applications over a cluster of computers. JESSICA stands for 'Java-Enabled Single-System-Image Computing Architecture'; it is an integration-software that allows multi-threaded applications to exploit the maximal parallelism obtainable from the cluster. JESSICA acts as a middle-ware that encapsulates the distributed nature of the cluster hardware and provides applications with an illusion that they are running on a single multi-processor computer.

The Single-System-Image (SSI) is realized through the provision of a *Global Thread Space*. When an application is instantiated, the JESSICA system creates the logical thread space that spans the cluster, for the containment and execution of threads to be created by the application, as shown in Fig. 1b. The Global Thread Space hides the physical boundaries between machines, Java threads can then move freely around the cluster from one machine to another. This movement is supported by a preemptive thread migration mechanism called Delta Execution. Migration is transparent to threads residing in the Global Thread Space because a redirection mechanism forwards location dependent operations back to the home machine for execution. Moreover, JESSICA provides a single and contiguous *Global Object Space* that allows memory objects to remain accessible by threads independent of their physical locations, even if the threads have been migrated to other machines. JESSICA supports the Global Object Space by deploying a distributed-shared-memory (DSM) system over the cluster, so that memory consistency against concurrent accesses can be ensured. Consequently, the free movement of threads in the Global Thread Space provides an opportunity for optimizing utilization of shared resources in the cluster.

JESSICA supports the *Serial-Program-Parallel-Subsystem (SPPS)* programming paradigm such that application programmers no longer need to worry about the number of processors available, they just create as many threads as needed, as in a single execution environment. The system can migrate the threads around the cluster automatically to maximize the parallelism. It handles thread-to-processor mapping transparently and maintains the integrity and consistency of shared resources, which are distributed around the cluster. Since JESSICA conforms to the standard Java Virtual Machine Specification [13], it is compatible to the vast number of existing multi-threaded Java applications. By providing the favorable multi-threaded model for writing parallel applications in Java, it can promote the Java programming language to be the language of choice for parallel applications developement in the future.

In Summary, the main characteristics of JESSICA include:

- Provide a Single-System-Image — JESSICA transforms a collection of interconnected computers into a single multi-processor computer when it is viewed from the application level. All the system level details such as mapping of active Java threads to processors and the redirection of communication data after a thread is migrated are transparently

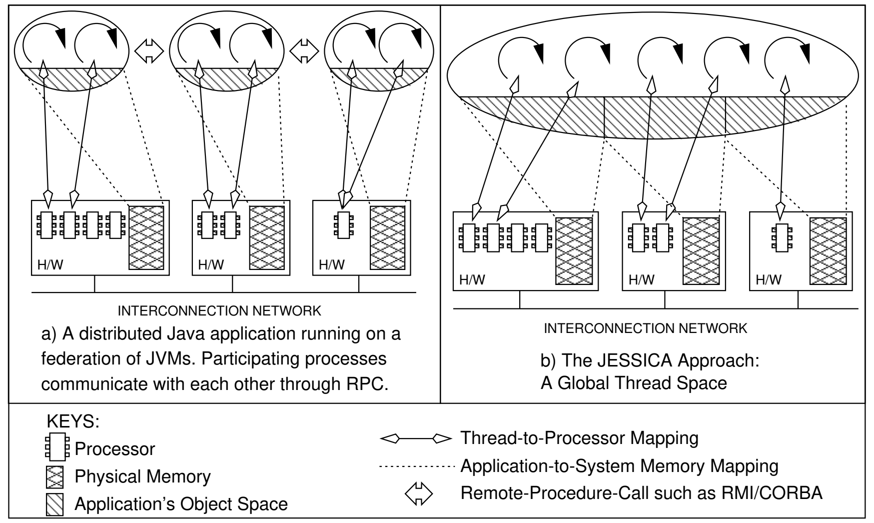

Figure 1: JESSICA encapsulates a cluster of computers into a single multi-processor machine

handled by JESSICA. With a preemptive thread migration mechanism installed, Java threads can be dynamically redistributed over the cluster whenever a load imbalance is detected. This kind of Single-System-Image illusion simplifies application program development by providing a single entry point, a single point for resource control and a single memory space for programmers. Parallel execution can be achieved simply by creating as many threads as needed, the thread migration mechanism can help to exploit the maximal parallelism and to optimize utilization of shared resources.

- Support Preemptive Migration of Java Threads — Delta Execution is a novel approach that we have devised for supporting preemptive thread migration. A Java thread can be preempted at any time and be migrated to another processor and to resume execution there. Delta Execution requires no knowledge from the application when the migration took place and is therefore transparent. In addition, Delta Execution does not introduce any migration-specific operations to be embedded into the application code, this ensures the greatest compatibility to the existing applications.

- Support Location Transparency — Location transparency is upheld in JESSICA by leaving helper threads (also known as master) at the home node. The helper threads are responsible for performing and any location dependent operations on behalf of the migrated threads (also known as slave). In addition, threads continue to be able to synchronize with one another even though some of them have been migrated to other nodes. This is achieved by using *Cooperative Semaphore*.

- A Portable UNIX Program — JESSICA executes Java applications in the form of a distributed Java Virtual Machine running on top of the UNIX operating system. JESSICA is, after all, just a UNIX program implemented entirely in the high-level language C/C++, hence it is portable across hardware platforms. The idea of running JESSICA as a middle-ware on top of an operating system also makes the goal of a Single-System-Image easier to achieve. Otherwise it would involve modifying the kernel such as the case in MOSIX [2] and SolarisMC [42], which can be much more tedious and time consuming.

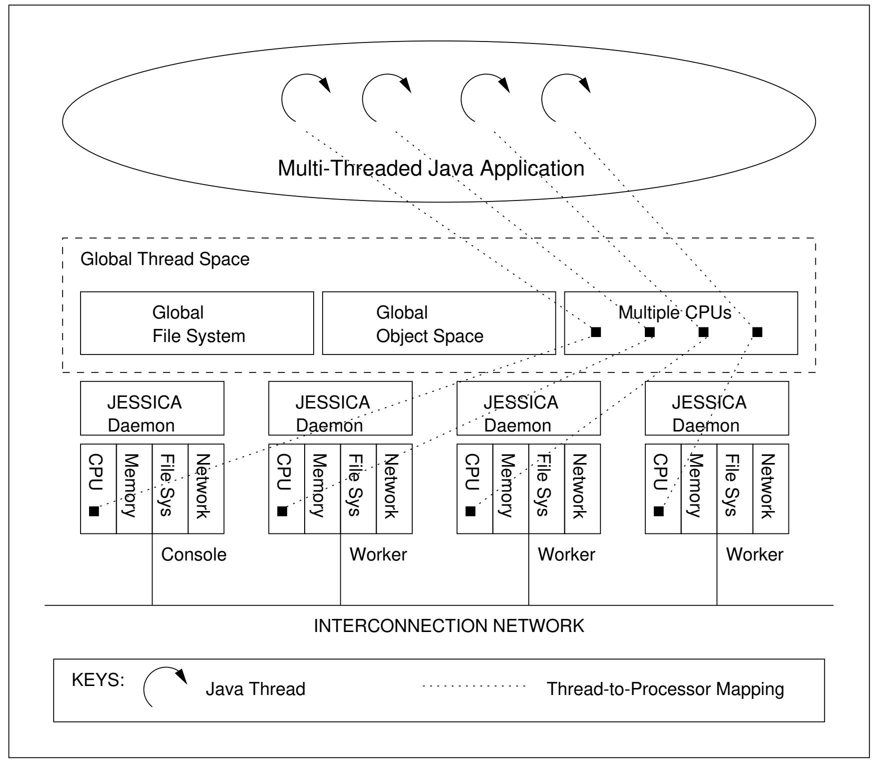

Figure 2: Entities in a JESSICA system

The rest of the paper is organized as follows. Section 2 presents the system architecture of JESSICA in details. Section 3 introduces Delta Execution, our solution to preemptive Java thread migration. Section 4 discusses how location transparency is supported. Section 5 evaluates the performance of our JESSICA prototype. Finally, section 6 concludes the paper and suggests further work on JESSICA.

2 System Architecture

JESSICA is designed as a group of daemons running on all the nodes in a cluster of computers, as illustrated in Fig. 2. They execute as user level processes on top of the UNIX operating system and it is the collaboration and coordination among these JESSICA daemons that offer an SSI illusion to Java applications. In JESSICA, we classify cluster nodes as either console or worker as follows:

- Console Node — Java applications can be started on any node in the JESSICA cluster. They are running on the distributed virtual machine that follows the home model, which is similar to that in Sprite [7] and MOSIX [1]. Any node at which the Java application is initiated will become the home of that application and is known as the console node. The whole cluster will appear as an extension to this console node where system resources including processors, memory pages and file storage locating at other nodes are all becomes accessible by applications running on it. The access of remote processors is supported by thread migration where an active thread can move and bind to a remote processor and execute there. The access of remote memory pages is supported by a DSM system where remote pages can be paged and cached locally when necessary, the access of remote files is supported by NFS provided by the underlying operating system.

The console node is responsible to handle any system service requests made by a migrated thread that are location dependent. In general, when a thread is migrated from the console node to another node in the cluster, the system resource requests made by the migrated thread will now be served by the local daemon running on the destination node. However, when the daemon discovers a requested service call that is location dependent, the call will be redirected back to the console node and to be handled there.

- Worker Node — When an application is instantiated on the console node, the role of other nodes in the JESSICA cluster is to support the console to execute the application and they act as the slaves with respect to the console. They are responsible to stand by and serve any requests forwarded from the console, in order to help the console to execute the application to its completion.

  During its course of execution, the migrated thread will continue to make system service requests to the worker node as if it was running at the console. The worker node will have to differentiate whether the request is location independent or not. If it is independent the request will be served locally, otherwise, the request will be forwarded back to the console. The console, after receiving the request, will perform the necessary operations and return the result back to the thread. The whole redirection process is carried out transparently, and the thread is totally unaware of what goes on.

2.1 Components in a JESSICA Daemon

Each JESSICA daemon is composed of the following four components that provide bytecode execution, memory management, thread creation, scheduling and synchronization to Java applications the same way as a standard Java Virtual Machine (JVM) does.

- Bytecode Execution Engine (BEE) — It is responsible to bind to an active thread and to execute its method code. Parallel execution of a multi-threaded application is realized by having multiple BEEs running on multiple machines to execute multiple threads simultaneously.

- Distributed Object Manager (DOM) — It is responsible to manage the memory resource in its local node and to cooperate with DOMs running on other nodes in the cluster to create the Global Object Space. They allocate new objects within the shared-space where access of objects can be made independent of their locations.

- Thread Manager (TM) — It is responsible to create, schedule and destroy threads running on the local node. Moreover, it works with TMs running on other nodes to perform thread migration. They marshal, ship and demarshal the execution states of migrating threads to resume their execution. They support distributed synchronization between migrated threads by forwarding synchronization operations back to the console TM.

- Migration Manager (MM) — It is responsible to collect load information of the local node and to exchange the information with MMs running on other nodes, in order to

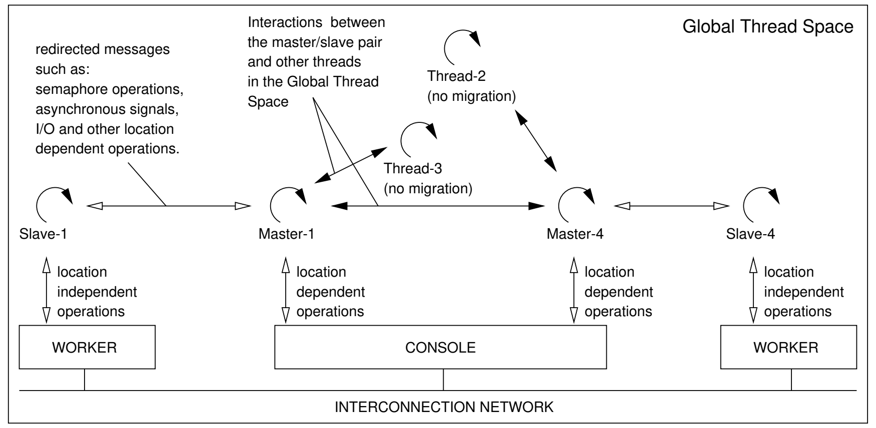

Figure 3: Interactions between the master and the slave threads that hide migration from the rest of the system

enforce a migration policy for load distribution around the cluster. When MM decides to perform thread migration, it notifies the TM about which thread should be migrated to where.

These four components are responsible to coordinate with their counterparts running on other nodes in the cluster to establish the SSI illusion in the form of a Global Thread Space. Together they provide a DSM layer for distributed-data sharing between threads, a preemptive thread migration mechanism for moving threads around the cluster for executing on different processors, and a message redirection mechanism for migrated threads to maintain the required location transparency.

2.2 A Master Slave Model for Migrating Threads

When a thread running on the console node migrates, it does not actually pack itself and move to the destination worker node. Instead, it is split into two cooperating entities, with one running at the original console, called the master; and the other running at the destination, called the slave. The slave thread is in fact created anew at the destination and acts as the migrated image to continue the execution of the original thread. The master thread remaining at the console is actually the original migrating thread, which is now responsible to perform any location dependent operations like I/O on behalf of the slave, in addition to forwarding messages between the slave and the rest of the system.

With this design we are able to create the Global Thread Space that maintains the same semantics and relationships between all the objects in the execution environment as if there was no migration, as shown in Fig. 3. For the rest of the threads that are running on the console node, only the master is visible to them, they are not aware of the existence of the slave as all the interactions between the slave and the rest of the threads have to go through the master. The redirections make the master appears to the rest of the threads that they are still interacting with the original thread as if there is no migration. Besides, for the slave who is now running at a worker node, it appears to be still running on the console because all the location dependent operations are redirected transparently back to the console to be performed by the master. As a result, the execution environment observed by a running thread in JESSICA is the same as that in a standard JVM, no matter whether the thread has been migrated or not.

3 Delta Execution

Delta Execution is a preemptive thread migration mechanism devised for JESSICA, which allows a cluster to move threads from the console node to the worker nodes. Delta Execution aims at providing a high-level and portable implementation for Java thread migration that does not deal with any low-level issues. Because of its high level approach, the Delta Execution mechanism is applicable to other languages which execute by means of virtual machine for supporting preemptive thread migration. The Delta Execution mechanism identifies and separates the machine independent states from the machine dependent states in the execution context of a running thread. When a thread migrates, only the machine independent states are migrated in a strategic manner. The manipulation of any machine dependent states is avoided by leaving them behind at the home node, where execution involving these states will still be performed. Active execution of the migrated thread is observed as switching back and forth between the home and the remote node. Because the migrated thread only incrementally advances its state of execution by a delta amount every time the execution control switches between the two nodes, we give this mechanism the name Delta Execution. The following is a summary of the characteristics of Delta Execution:

- Transparent migration — the Delta Execution mechanism is implemented entirely at the virtual machine level which is transparent to a migrating thread; together with a redirection mechanism for forwarding location-dependent operations and a distributed shared-memory subsystem that supports location-independent memory access, migration transparency can be guaranteed.
- Responsive to changes in system load — migration granularity is per bytecode-instruction where a thread can be stopped at any time and be migrated once the execution of the current bytecode instruction is finished. The system can swiftly response to any change in system load across the cluster for performing load balancing.
- Portable and manageable implementation — unlike traditional systems that implement migration at the kernel level, Delta Execution is implemented on top of the kernel which does not involve any low-level details of the operating system or the hardware. This approach provides better portability and the software effort required is more manageable.
- Compatible to existing Java applications — because the migration mechanism does not introduce any new migration-related instructions to a Java program, existing multi-threaded applications can therefore benefit from Delta Execution without modification.

3.1 The Java Method Frame (JMF) Structure

Observed that the life of an active Java thread begins with the calling of the `java.lang.Thread.start()` method or the equivalent if this thread is a sub-class of the `java.lang.Thread`. The bytecode instructions of this method will be executed by the execution engine, which may further call other methods, and so on. Finally, the thread terminates when this `java.lang.Thread.start()`

```
00 class Factorial {
01  static public int f(int n) {
02   if (n == 1) return 1;
03   else return n*f(n-1);
04  }
05  static public void main(String argv[]) {
06    int result = f(10);
07  }
08 }
```

Figure 4: `Factorial.java`

method or the equivalent returns. In other words, the execution states of a running thread can be represented by a method calling sequence and their respective execution context local to each method.

Here a structure called *Java Method Frame* (JMF) is defined to help represent the method-invocation sequence of an active thread. The purpose of JMF is to express the context of a Java method under execution. A JMF contains:

- a program counter (PC) pointing to the bytecode that is currently under execution;
- NPC, the location of next bytecode instruction to be executed;
- a stack pointer (SP);
- the set of local variables of this method;
- the method execution stack;
- other miscellaneous information such as that for Java exception handling.

Notice that information containing in a JMF is architecture-neutral.

A Simple Thread Migration Example

The above `Factorial.java` example, which recursively calculates the factorial of 10 (Fig. 4, line 06), illustrates how thread migration is carried out and how execution states are captured and transferred based on the JMF structure. The compiled bytecode instructions are shown in Fig. 5. The numbers following the line numbers in Fig. 5 represent the corresponding locations of the bytecode instructions to be stored in the program counter (PC).

Consider the console node triggering a migration when the `Factorial` program has executed for a number of iterations, and at this time `f()` has already recursed three times when the running thread is frozen at PC = 8 of method `f()` (Fig. 5, line 18). After the current instruction at PC = 8, i.e. `iload_0`, has been completed, the execution context of the *main* thread can be represented pictorially as in Fig. 6.

Suppose at this moment this *main* thread migrates, the above five JMFs which constitute the entire execution context of *main*, are packed and shipped to a worker node. After the execution context arrives at the destination worker node, they are unpacked and reconstructed, and eventually bound to the execution context of a newly created *slave* thread. When the *slave* thread resumes its execution, it continues from the tail-JMF, and updates

```
Method void main(java.lang.String[])
  0 bipush 10
  2 invokestatic #4 <Method int f(int)>
  5 pop
  6 return

Method Factorial()
  0 aload_0
  1 invokespecial #3 <Method Object()>
  4 return

Method int f(int)
  0 iload_0
  1 iconst_1
  5 iconst_1
  2 if_icmpne 7
  6 ireturn
  7 iload_0
  8 iload_0
  9 iconst_1
 10 isub
 11 invokestatic #4 <Method int f(int)>
 14 imul
 15 ireturn
```

Figure 5: Disassembled bytecode of the class `Factorial`

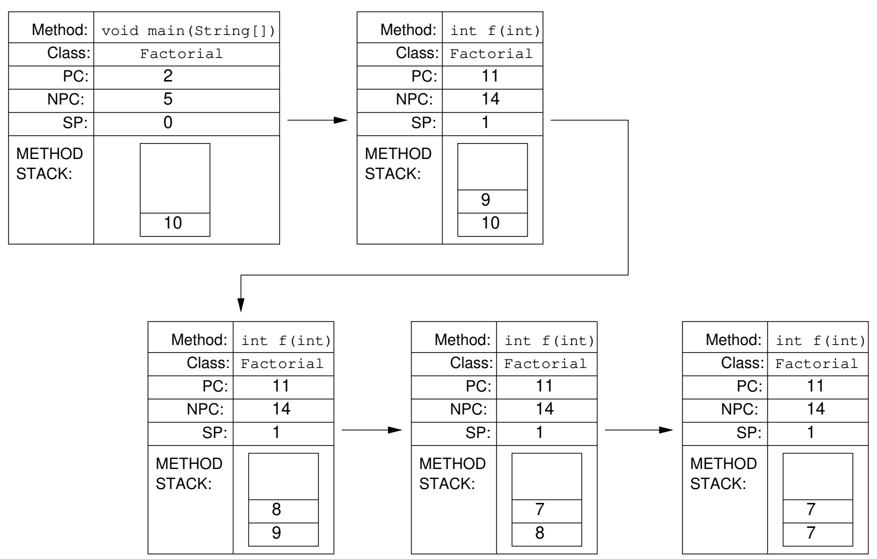

Figure 6: Execution context of thread *main* after a number of iterations

```
00 class Bar {
01   static int count;
02   int id;
03   static { count = 100; }
04   Bar() { id = count--; }
05 }
06
07 class Foo {
08   public static void main(String argv[]) {
09     Bar b = new Bar();
10   }
11 }
```

Figure 7: `Bar.java` and `Foo.java`

the value of PC to 9, i.e. `iconst_1` (Fig. 5, line 19), according to the content of the NPC; the result is an integer constant 1 being pushed onto the tail-JMF's method stack. From this point onwards, the *slave* thread continues its execution correctly from the point where the *master* was previously stopped. When the recursive method `f()` at this level (n = 7) returns, the JMF is popped and the context of the previous JMF (n = 8) is restored. The method stack now should contain two integer elements: 8, and 5040, i.e. the value of n and the evaluated value of `f(7)` respectively. From this point onwards, the execution context of the *slave* thread further unwinds itself and the JMFs of the method `f()` are popped from the thread runtime stack one after another as the recursive calls return. Eventually when all the JMFs are exhausted the *slave* thread will have completed its execution, and the result is the return value of `f(10)` stored at the top of the stack.

3.2 How Machine-dependent States Come About

Although the JMF structure discussed in the last section appeared to be comprehensive enough to capture the execution context of a running thread, it turns out that such solution is only good for part of the cases. If a thread invokes methods directly from within the execution engine through one of the `invokeMethod`[1] instructions, then the scheme of using a sequence of JMFs as described can sufficiently capture and represent the state of this running thread at any point of its execution. On the other hand, if this thread calls a native method or if the execution engine tries to execute an instruction that within itself will invoke another method, this will generate machine dependent states where a sequence of JMFs alone cannot sufficiently capture the entire execution context. This can be illustrated by the following `Bar` and `Foo` class as shown in Fig. 7. Their corresponding disassembled bytecode are shown in Fig. 8 and Fig. 9 respectively.

An Illustrating Example

In this example, the `static` block in the class `Bar` defining the class initialization routine (Fig. 7, line 03), internally denoted as method ‘‘<clinit>’’ (Fig. 8, line 12), is to be executed once when the class is first loaded by the virtual machine. According to this example, the `static` block will initialize the class variable `count` to 100 when `Bar` is loaded (Fig. 7, line 03).

[1] 1 A method can be invoked from within the execution engine by executing one of the `invokevirtual`, `invokespecial`, `invokeinterface`, or `invokestatic` bytecode instructions.

```
Method Bar()
  0 aload_0
  1 invokespecial #4 <Method java.lang.Object()>
  4 aload_0
  5 getstatic #5 <Field int count>
  8 dup
  9 iconst_1
 10 isub
 11 putstatic #5 <Field int count>
 14 putfield #6 <Field int id>
 17 return

Method <clinit>
  0 bipush 100
  2 putstatic #5 <Field int count>
  5 return
```

Figure 8: Disassembled bytecode of the class `Bar`

```
Method Foo()
  0 aload_0
  1 invokespecial #5 <Method java.lang.Object()>
  4 return

Method void main(java.lang.String[])
  0 new #1 <Class Bar>
  3 invokespecial #4 <Method Bar()>
  6 return
```

Figure 9: Disassembled bytecode of the class `Foo`

```
00 // A simplified internal implementation of the
01 // bytecode instruction new in the execution engine
02 //
03 void instruction_new(char *className) {
04   // if the class is not loaded, lookupClass
05   // will load it also
06   javaClass* class = lookupClass(className);
07   // if the class has a class initialization
08   // method to execute, invoke the method too
09   if (class->needInitialize) {
10     vmExecuteJavaMethod(class, "<clinit>");
11     // the initialization method is only
12     // required to execute once
13     class->needInitialize = false;
14   }
15   // ask MemoryManager to allocate a memory block for
16   // the new object
17   javaObject* object = MemMgr->alloc(class->objectSize);
18   // push the object handle onto the method stack
19   *SP++ = object;
20   return;
21 }
```

Figure 10: Simplified implementation for the new bytecode instruction in C/C++

After the execution of `Foo` is initiated, the *main* thread of the virtual machine enters the `main()` method of the class `Foo` (Fig. 7, line 08) and executes the first instruction at PC = 0, i.e. `new` (Fig. 9, line 06). This instruction creates a new object instance of class `Bar` after a JMF for the `Foo.main()` method has been pushed onto the runtime thread stack. At this point, the execution engine looks up and loads the class `Bar`, as it is given in the argument to the instruction (Fig. 9, line 06). After the class is loaded, the execution engine invokes the class initialization routine, if there is any. Finally, the execution engine obtains a free memory block to create a new `Bar` object. The C/C++ code segment shown in Fig. 10 summarizes the above actions. It is a simplified implementation for the new bytecode instruction.

When the *main* thread invokes the ``<clinit>'' method at line 10 of the `instruction_new` function in Fig. 10, it will push another JMF onto the runtime thread stack to record the entering of a new method context. If at this moment the console initiates a migration as the thread executes the first instruction at PC = 0, i.e. `bipush` (Fig. 8, line 13), the *main* thread will be frozen after the `bipush` instruction has been completed successfully. Consequently, the execution context of the thread can be represented by the sequence of JMFs as illustrated in Fig. 11.

When the *slave* thread resumes execution at the destination worker node, it continues within the method context of ``<clinit>'' at PC = 2, which stores the value 100 found from the top of the method stack to the class variable `count` and then returns (Fig. 8, line 14). The JMF of this context is therefore popped from the runtime thread stack, and the method context of the `Foo.main()` is restored. Now if we follow exactly the scheme as described in section 2, the execution engine will continue the execution within the method context of `Foo.main()` and execute the next instruction at PC = 3, i.e. inovkestatic (Fig. 9, line 07). But this will lead to incorrect result, as the *slave* thread has "forgotten" to allocate memory for the new `Bar` object.

Note that if there is no migration, the control flow of the *main* thread should con-

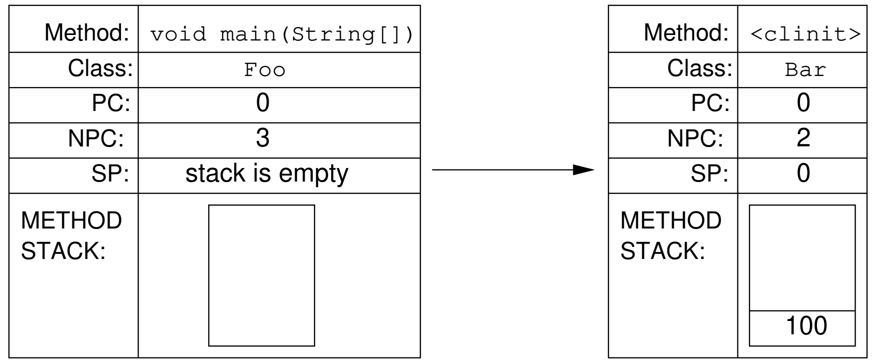

Figure 11: Execution context of thread *main* after entering the class initializer ‘‘<clinit>’’

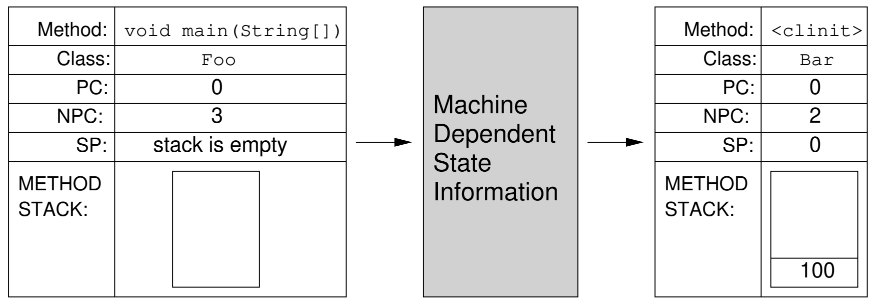

Figure 12: Revised execution context of thread *main* after entering the class initializer ‘‘<clinit>’’

tinue at line 13 of the `instruction_new()` implementation as shown in Fig. 10, after the ‘‘<clinit>’’ method returns; i.e. to set the `needInitialize` flag to `false`, to obtain a memory block from the memory manager, and to push the memory handle onto the method stack (Fig. 10, line 13 to line 19) before returning to the method context of `Foo.main()` at PC = 3, i.e. `invokestatic` (Fig. 9, line 07). In other words, there is certain state information of the execution context that failed to be represented between the two JMFs, as shown in Fig. 12.

The missing information, including the return address at line 13 of the `instruction_new()` code in Fig. 10, is in fact stored in the *runtime C-stack* of the daemon process running at the console, which is implementation and hardware dependent. The C-stack here refers to the runtime execution stack of a UNIX process. We call it a C-stack to differentiate from the runtime thread stack of the bytecode execution engine.

As the shaded block in Fig. 12 is machine dependent, it cannot be migrated directly. Here we propose an alternative approach to thread migration that is independent of the lower level implementation of the operating system. Concluding from the two examples discussed above, we can categorize JMFs into two types:

- *B Frame* — is the JMF that is pushed to the runtime thread stack as a result when a method is invoked from within another method. The invocation is called from within bytecode by the execution of either `invokevirtual`, `invokespecial`, `invokeinterface`, or the `invokestatic` instruction. For example, the calling of method `f()` at PC = 2 of `Factorial.main()` (Fig. 5) as well as the recursive invocation in `f()` itself at PC =

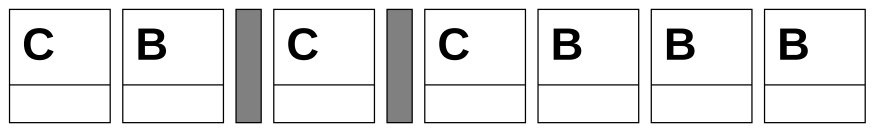

Figure 13: Revised thread execution context representation, shaded blocks are the sets of machine dependent states not captured by JMFs

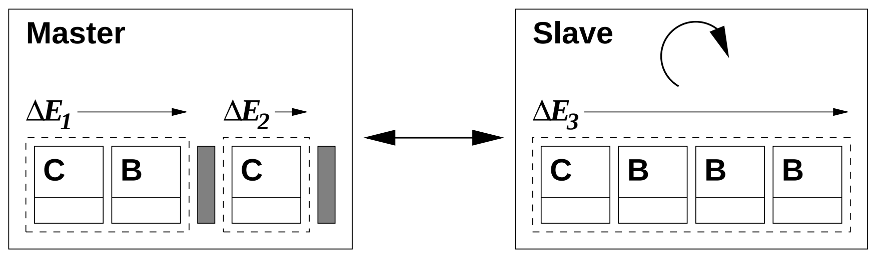

Figure 14: Delta sets are sent to the *slave* thread one by one for execution

11 will both push a B frame onto the stack.

- *C Frame* — is the JMF that is pushed to the runtime thread stack when the implementation code of the execution engine or a native method invokes explicitly a method through the `vmExecuteJavaMethod()` function. It is the machine dependent state introduced by the implementation code before pushing this C frame onto the thread stack that cannot be captured directly. For example, the JMF being pushed at line 10 of the `instruction_new()` code in Fig. 10 is a C Frame.

Consequently, for any thread execution context, the correct representation should be expressed as a sequence of interleaving B frames and C frames, with a set of machine dependent states attached before each C frame. Note that every sequence must begin with a C frame, since any thread execution must be initiated by the virtual machine implementation itself (Fig. 13).

Our approach is to partition a thread execution context into chunks of JMFs, known as *delta sets*, ΔE_i (Fig. 14). Each *delta set* begins with a C frame and followed by zero or more B frames. Now, instead of shipping the whole execution context to the destination as done in the traditional way of thread migration, we ship them in an incremental manner, one *delta set* at a time.

When the console node performs a migration operation, it marshals and ships the tail *delta set* in the execution context of the migrating thread. The *slave* thread at the destination worker node will bind to this *delta set* and to continue execution until the JMFs in this set is exhausted. After which the execution control will switch back to the *master* thread, to let it finish whatever machine dependent context whose states cannot be shipped to the destination, such as that from line 13 to line 19 in the `instruction_new()` implementation as shown in Fig. 10. Immediately after the machine dependent execution is completed, control flow will switch back to the *slave* with the next *delta set* so as to continue execution at the worker node.

With this arrangement of *Delta Execution*, every time the worker node only incrementally advances the execution states of the migrating thread by *delta*. Besides, we are able to isolate those non-migratable machine dependent contexts and to have them executed locally at the console, while machine independent contexts are migrated across machines and be executed remotely.

4 Support for Location Transparency

Location transparency support in JESSICA is based on a master-slave design for thread migration, where the cooperation between the master thread running on the console node and the slave thread running on the worker node together produce the required transparency. With this master-slave design we are able to implement location-transparent services such as network communication, file operations, distributed thread synchronization and remote exception.

4.1 Cooperative Semaphore

The essence of multi-threaded computing is that threads can share resources and execute concurrently to multiplex multiple computations, or even communications, at the same time. In such a multi-threaded runtime environment like JESSICA, mechanism for providing mutual exclusion is necessary to ensure coordinated access to shared resources so as to maintain their integrity. The Java Programming Language utilizes semaphore to implement mutual exclusion control. A semaphore is associated with every shared object so that it is possible for application programs to avoid any race condition by first acquiring the associated semaphore before updating a shared object. Thread synchronization is a direct consequence of using semaphore. Because if a thread that is trying to acquire a semaphore of a given object, but the semaphore is already acquired by another thread, then the thread will be blocked. This is in effect having the Thread Manager (TM) to schedule out the thread and place it at the waiting queue. This blocked thread will be re-scheduled back to the ready queue by the TM when all the other threads that have previously acquired the semaphore before the blocked thread did later release it. As a result, the implementation of semaphore has to couple to that of the TM, so that semaphore operations are able to trigger thread rescheduling when necessary.

In JESSICA, as a thread is possible to migrate to other nodes and continue to execute there, the implementation of semaphore and the TM have to be augmented in order to maintain the same semantics of semaphore operations even when a thread is migrated, as described in the last paragraph. This is because when a migrated thread tries to acquire a semaphore of a shared object, the semaphore may be located back at the console node, hence a mechanism for remote semaphore operations is required. In addition, when re-scheduling a thread that is previously blocked by an already acquired semaphore, the thread can be a migrated one and a mechanism for notifying the TM of the node where the migrated thread is located is also necessary in order to schedule in the migrated thread.

JESSICA takes a de-centralized approach to implements distributed semaphore. The approach makes use of the master-slave thread pair to forward semaphore operations between the console node and the worker nodes. The TMs are not required to be aware of the existence of other TMs and work together, each TM can perform its own local thread scheduling without affecting the other TMs. In addition, all the semaphore operations are performed at the console node by the master threads, so the same semaphore semantics is enforced as if there was no migration. The de-centralized approach is therefore more scalable and reliable than the centralized approach. The implementation of distributed semaphore in JESSICA is called Cooperative Semaphore. It is the cooperation between the master thread at the console and the slave at the worker node that provides a transparent, scalable and reliable implementation.

Observed that a running thread T will be blocked and scheduled out from the ready queue if:

- It tries to acquire a semaphore S where S is previously acquired by another thread T', or
- It tries to perform an I/O operation in blocking mode and the I/O channel is not ready, or
- It explicitly performs a *wait* operation on a given object O.

And a blocked thread T will be rescheduled back to the ready queue when:

- The semaphore S that T has previously requested is released by another thread T' and it is now T's turn to acquire S, or
- The I/O channel that T previously tried to operate on is now ready, or
- Another thread T' has issued a *notify* operation on an object O which T has previously waited upon.

We take advantage of the property that a thread will block when trying to read from an I/O channel until data has arrived. When a slave thread tries to acquire a semaphore S, instead of directly operating on the semaphore S, it sends a message to its master, asking the master to acquire S at the console on its behalf. After that the slave thread will be blocked waiting for the master reply. Now at the console node, when the master thread received the semaphore acquire request on S from its slave, it will try to acquire the semaphore S accordingly. Until eventually the master has successfully acquired the semaphore S, it will then send a successful message back to its slave so that the slave can continue its execution at the worker node as if it has successfully acquired the semaphore itself. Similarly, when the slave thread later tries to release the semaphore S, it again sends a message back to the master and asks it to release the semaphore S on its behalf. After the master has received the message and released the semaphore accordingly, the local TM at the console can then re-schedule any threads that have previously issued a acquire operation on the semaphore. With this design the observed effect for the slave thread is that a semaphore acquire operation will block until the semaphore is acquired, and a semaphore release operation will causes other threads that are also trying to acquire on the semaphore to be rescheduled. Hence the fact of migration is transparently hidden from the slave. And for the rest of the threads that are running on the console node, what they observed is still the master thread who performs all the semaphore acquire and release operations, therefore the fact of migration is also transparently hidden from them. The code segments in Fig. 15 illustrate how the above mechanism is implemented at the master and the slave thread.

```
// Master waiting for
// redirection request
// at the console ...
while (true) {
  req = waitForRequest();
  switch (req.cmd) {
    case REMOTE_SEM_ACQUIRE:
    semAcquire(req.sem);
    replyResult(ACQUIRE_OK);
    break;
    case REMOTE_SEM_RELEASE:
    semRelease(req.sem);
    replyResult(RELEASE_OK);
    break;
    // ...
    // ...
    // ...
  } // end switch
} // end while
```

```
// Slave performs the following
// remote semaphore operations
// to redirect the operations back
// to the master
void remoteSemAcquire(Sem s) {
  Request req;
  req.cmd = REMOTE_SEM_ACQUIRE;
  req.sem = s;
  sendRequest(req);
  Reply rep = waitForReply();
  assert(rep == ACQUIRE_OK);
}

void remoteSemRelease(Sem s) {
  Request req;
  req.cmd = REMOTE_SEM_RELEASE;
  req.sem = s;
  sendRequest(req);
  Reply rep = waitForReply();
  assert(rep == RELEASE_OK);
}
```

Figure 15: Pseudo-code for implementing Cooperative Semaphore in JESSICA

4.1.1 Distributed Thread Synchronization

Note that by similar design, we are able to implement remote thread signaling where the master is responsible to transparently forward any *wait* and *notify* signals between its slave and the rest of the threads running on the console node. With the Cooperative Semaphore and the remote signaling mechanism installed, we are able to implement distributed thread synchronization in a decentralized manner that is reliable and scalable.

4.2 I/O Redirection

An important issue in providing migration transparency is how to ensure all the opened files and network communication links to remain functional the same way as they are before migration. They are the primary requirements for implementing location transparency and network transparency. They have been a non-trivial problem for traditional process migration systems implemented at the kernel level such as Sprite [12] because the control information relating to the open files and network communication end-points are all stored within the kernel and migrating them is difficult. On the other hand, for higher level implementation like Condor [25] and JESSICA, it is possible to achieve location and network transparency by augmenting the subsystems implementation that provides the files and network communication services to the applications running on top of them.

As in the case of JESSICA, location and network transparency are achieved by having the master and the slave thread to forward the corresponding messages and operations between the console and the worker node. File and network communication operations are location dependent operations and hence they are all redirected back at the console node where it is the master thread to perform the operations. The redirection code are implemented within the `java.io` and the `java.net` class libraries for files operations and socket communications respectively. Their interface definitions are kept unchanged so that existing classes that make use of their methods can still function correctly without modification. An important feature of JESSICA that has simplified the implementation of I/O redirection is its object-

```
00 // implementation of java.io.FileInputStream.read() method with
01 // redirection
02
03 jint java_io_FileInputStream_read(
04   struct Hjava_io_FileInputStream *this) {
05   if (currentThread->isSlave) {
06     Request req;
07     req.cmd = JAVA_IO_FILEINPUTSTREAM_READ;
08     req.obj = this;
09     sendRequest(req);
10     Reply rep = waitForReply();
11     return rep.result;
12   } else {
13     int fd = unhand(unhand(this)->fd)->fd;
14     char byte;
15     int r = read(fd, &byte, 1);
16     if (r > 0)
17       return byte;
18     else return r;
19   }
20 }
```

Figure 16: Code segment for the implementation of `java.io.FileInputStream.read()` with redirection

oriented nature. The class hierarchies of both the `java.io` and the `java.net` class libraries are well organized. There are base classes located towards the top of the hierarchies who are responsible for performing the raw I/O operations on the underlying operating system. All of their children classes who are specialized to perform specific I/O operations inherit the functionality offered by these base classes and will directly invoke the methods provided by them when trying to access the raw I/O channels. As a result, when these base classes in the hierarchies are augmented to support the required migration transparency, the rest of the children classes who inherit them will automatically support migration transparency also. This is also one of the favorable feature of object oriented programming.

When file I/O is concerned,

- the `java.io.File`,
- the `java.io.FileInputStream`,
- the `java.io.FileOutputStream`,
- and the `java.io.RandomAccessFile` base classes

in the `java.io` class libraries are identified as necessary to incorporate the redirection code for supporting location transparency. The code segment in Fig. 16 demonstrates how the `java.io.FileInputStream.read()` (Fig. 16) is augmented. Note that because the method needs direct access to the underlying operating system, it is a native implementation.

Notice that at line 04, the system check if the current thread is a slave. If it is a slave then instead of performing the read operation locally, the operation is sent back to the master thread and ask it to read the file channel on its behalf, the `FileInputStream` object is also transported for the master to operate upon (line08).

Simiarly, when socket I/O is concerned, the following are the classes from the `java.net` library that need to be augmented:

```
// implementation of java.net.DatagramSocket-
// .datagramSocketCreate() method with redirection

void java_net_DatagramSocket_dataSocketCreate (
  struct Hjava_net_PlainDatagramSocketImpl *this) {
  if (currentThread->isSlave) {
     Request req;
     req.cmd = JAVA_NET_DATAGRAMSOCKET_DATAGRAMSOCKETCREATE;
     req.obj = this;
     sendRequest(req);
     Reply rep = waitForReply();
     assert(rep.result == SUCCESS);
  } else {
     int fd = socket(AF_INET, SOCK_DGRAM, 0);
     unhand(unhand(this)->fd)->fd = fd;
     return;
  }
}
```

Figure 17: Code segment for the implementation of `java.net.DatagramSocket.datagramSocketCreate()` with redirection

- `java.net.SocketInputStream`
- `java.net.SocketOutputStream`
- `java.net.PlainSocketImpl`
- `java.net.PlainDatagramSocketImpl`

The code segment in Fig. 17 demonstrate how the to modify the method for creating a datagram socket.

Now for the master thread to be able to handle the redirection request correctly, its service loop has to be augmented as shown in Fig. 18.

The master thread after receiving the I/O redirection requests from slave at line 13 and line 18, it has to perform the request locally (line 15 and line 20) and reply the slave with the result of the operations (line 16 and line 22).

4.3 Remote Exception

The Java Programming Language supports a language exception construct where a method can defined a block of code for handling any specified exceptions that may be generated as the method executes (Fig. 19). In the example, the code segment that can generate a *divide-by-zero* exception is enclosed in the *try* block from line 03 to 04, and the code segment to handle the exception is enclosed in the *catch* block from line 05 to 08 that follows. If the virtual machine can execute the *try* block without generating any exception, the program continues to execute from line 09, otherwise the *catch* block will be executed before resuming execution from line 09.

The idea of language exception is that when the current execution has generated an anticipated error, the execution can be aborted and then be rewind back to a point where the program has pre-defined a specific handler to deal with the anticipated error. As a result, to implement language exception in the Bytecode Execute Engine (BEE), the sequence of

```
// Master waiting for redirection request at the console ...
while (true) {
  req = waitForRequest();
  switch (req.cmd) {
    case REMOTE_SEM_ACQUIRE:
    semAcquire(req.sem);
    replyResult(ACQUIRE_OK);
    break;
    case REMOTE_SEM_RELEASE:
    semRelease(req.sem);
    replyResult(RELEASE_OK);
    break;
    // ...
    // ...
    case JAVA_IO_FILEINPUTSTREAM_READ:
    Reply rep;
    rep.result = java_io_FileInputStream_read(req.obj);
    replyResult(rep);
    break;
    case JAVA_NET_DATAGRAMSOCKET_DATAGRAMSOCKETCREATE:
    Reply rep;
    java_net_DatagramSocket_dataSocketCreate(req.obj);
    rep.result = SUCCESS;
    replyResult(rep);
    break;
  } // end switch
} // end while
```

Figure 18: Service loop of the master thread that handles redirection request of location dependent operations from Slave

```
class Foo {
  public float divide(int a, int b) {
    float c;
    try {
      c = a / b;
    } catch (ArithmeticException e) {
    System.out.println("divide-by-zero!");
    c = float.NaN;
  } // end try-catch
  return c;
} // end divide
```

Figure 19: A simple try-and-catch example of exception

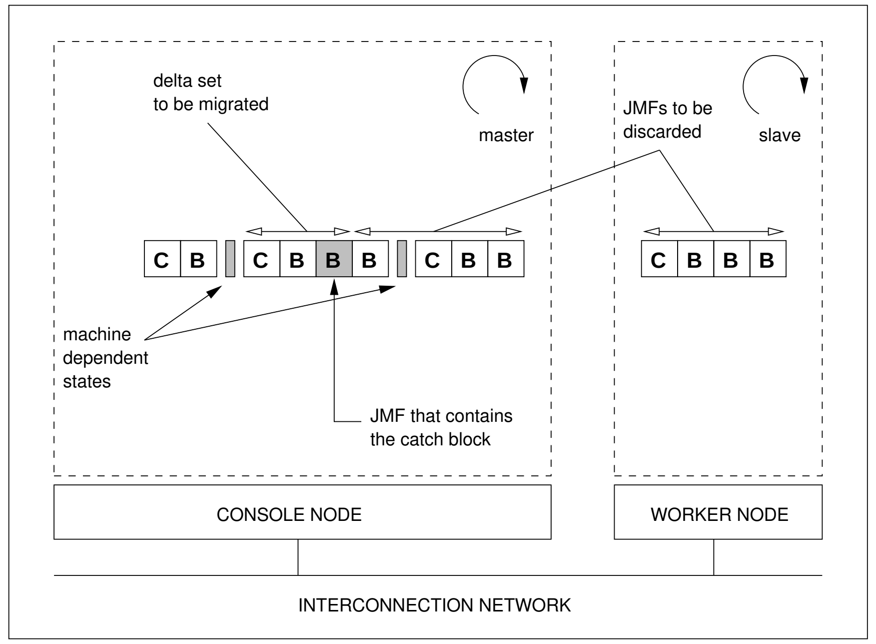

Figure 20: The mechanism for handling Remote Exception

Java Method Frames (JMFs) are to be scanned from the tail to the head to locate the nearest called method which has implemented a *catch* block that can handle the exception. When this method is located, the execution context of the current thread is rewind up to this method so that execution can continue from the exception handling block of this method.

Notice that when a thread is migrated, according the Delta Execution implementation, only the tail-most delta set is sent to the worker node for the slave thread to execute. Now if this slave thread generates a remote exception, it is possible that the worker node is unable to locate the JMF from the delta set that can handle the remote exception. This is because the JMF of the method that contains the *catch* block for this remote exception is still located at the console node. Hence, instead of generating an uncaught exception error, the system should discard the slave thread and forward the exception back to the master thread at the console, so that the searching for the right JMF to handle the exception can continue (Fig. 20). Once the JMF has been located, the corresponding delta set that contains this JMF can be sent to the worker node and a new slave thread can be instantiated to continue execution of the *catch* block that handle the remote exception.

5 Performance Evaluation

We have implemented a JESSICA prototype that runs on a cluster of 12 SUN Ultra-1 machines interconnected by a Fore ASX1000 ATM switch. The implementation is based on Wilkinson's work on the KAFFE virtual machine [14] and the Treadmarks package [5] for distributed shared-memory support.

```
00 class Foo extends Thread {
01 int j;
02 public void run() {
03   long begin = System.currentTimeMillis();
04   for (int i = 0; i < 1000000; i++)
05     j = 1;
06     long total_time = System.currentTimeMillis() - begin;
07     System.out.prinln("time for 1000000 iteration = "+total_time);
08   }
09 }
```

Figure 21: class Foo measures the time it takes to update an integer object variable 1,000,000 times

5.1 Primitive Operations Overhead

This section studies the extra overheads that are incurred as a result of allowing threads to distribute around the cluster and to execute in parallel on multiple nodes. These overheads come from distributed object access and distributed thread synchronization. Note that they are not presented in the traditional case where there is no migration as all the threads are running on the same machine.

5.1.1 Overhead from Accessing Distributed Objects

In order to ensure objects to be continuously accessible by any threads even after they have migrated to another machine, objects in the JESSICA system are allocated from the distributed-shared memory (DSM). The use of a DSM system can simplify the implementation of the distributed virtual machine substantially. On the other hand, since it is possible to have two or more threads to update a given object at the same time, extra operations have to be executed in order to prevent such race condition from occurring. They are the semaphore acquire and release procedures available from the DSM subsystem for maintaining the consistency of the shared memory. However, these operations introduce additional overhead for object accesses. Furthermore, when a thread tries to access an object that is either not cached locally or located in a dirty memory page, this will also cause the DSM subsystem to fetch the page in which object is residing from a remote node, and thus introducing extra access penalty.

To study the performance difference for object accesses, we have designed the following simple class Foo (Fig. 21), which measures the duration for updating an object variable j 1,000,000 times. To obtain the time for the case of distributed object access, we simply create an instance of the Foo thread and migrate it to a remote node before the <run()> method is invoked.

Notice that an experimental error is introduced by the loop counting variable i in the Foo class because extra time is taken for incrementing it 1,000,000 times (Fig. 21, line 04). However, the fact is for local variables like i which are defined locally in a method and are of simple data type, they are allocated from the local method stack which do not belong to the DSM. To eliminate the error we need to run another test program that measures the time to update a local integer variable i 1,000,000 times, the corresponding duration is found to be about 3562 milliseconds. The adjusted result for updating an object variable both with or without migration support is tabulated in Table 1. From the result shown, it is found that

	Object allocated in DSM for supporting migration (msec)	Object allocated in local memory when not supporting migration (msec)
Adjusted time for 1,000,000 updates	35343	3929

Table 1: Adjusted time for updating an integer object variable 1,000,000 times

the access penalty for distributed object accesses is about 9 times more than the case when objects are allocated locally.

Another point to note is that when updating an object variable, the virtual machine has to execute an additional bytecode instruction that loads the address of the object onto the method stack first, before it can store value to the object using the `PUTFIELD` instruction. On the other hand, a local variable can always be directly stored in a slot of the local stack, whose location is already determined at compile time. Therefore the time to update a local variable which is of simple data type will always be shorter than that for an object variable of the same data type, no matter the implementation supports migration or not. In light of this, we have also measured the time to update a local integer variable 1,000,000 times by defining `j` in class `Foo` as a local variable instead. As a result, we found that the ratios of memory access overhead between object variable that is allocated from the DSM, object variable that is allocated in the local heap, and local variable that is allocated from the method stack are:

$$\begin{array}{lllllll} \text{Distributed Object Variable} & & \text{Local Object Variable} & & \text{Local Stack Variable} & & \\ \text{Access Penalty} & : & \text{Access Penalty} & : & \text{Access Penalty} & & = 19.54 : 2.17 : 1 \\ \text{(simple data type)} & & \text{(simple data type)} & & \text{(simple data type)} & & \end{array}$$

5.1.2 Overhead from Cooperative Semaphore

As explained in section 4.1, the Standard Java Virtual Machine [36] utilizes semaphore for implementing mutual exclusion control between threads. For the case of JESSICA, mutual exclusion control is implemented as Cooperative Semaphore where the same semaphore semantics is maintained even after threads are migrated to other nodes. Notice that Cooperative Semaphore is independent from the DSM semaphore that is mentioned in the last section. When a thread tries to enter a critical section in the DSM and is blocked, the whole JESSICA Daemon running on the node will be blocked, too. On the other hand, Cooperative Semaphore works in conjunction with the Thread Manager (TM); when a thread tries to acquire a Cooperative Semaphore and is blocked, the TM will schedule in another ready thread to continue execution so as to keep the JESSICA Daemon alive. Since the Cooperative Semaphore is built on top of the TCP/IP protocol, a migrated thread has to send message requests back to its master in order to have the master to perform the semaphore operations on its behalf, therefore it will take longer time for a migrated thread to acquire and release a semaphore. This section compares this Cooperative Semaphore overhead with that when there is no migration.

Consider when a migrated thread tries to acquire a Cooperative Semaphore (Fig. 22), it will send a semaphore-acquire message back to its master thread (T0). When the message arrives at the console node, it will trigger a `SIGIO` signal and the corresponding signal handler of JESSICA will notify TM that data is ready for the master thread, and the master thread is then rescheduled back to the ready queue. However, the master thread may not be able to resume its execution immediately because there may be other threads which have higher execution priorities. Even if they are of the same priority, they may be queuing in front of the master thread at the ready queue. Anyhow after a while (T1) it will come to the

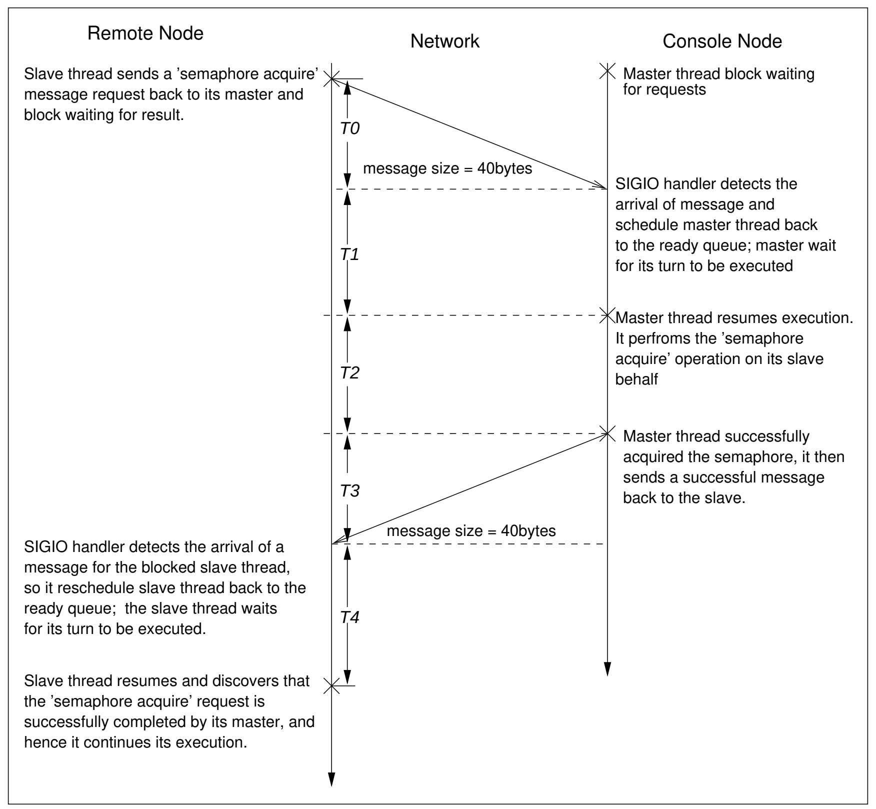

Figure 22: Cooperative Semaphore in action: a migrated thread performs an acquire operation

master's turn to execute, and as a result the master can acquire the semaphore accordingly (T2). Finally, when the semaphore is successfully acquired, the master then sends a successful message back to the slave. Once the message arrives and the slave is resumed (T3 + T4), it will notice that its master has successfully acquired the semaphore and therefore it can continue its execution. From the diagram illustrated in Fig. 22, it can be seen that the total time for a slave thread to acquire a Cooperative Semaphore equals to (T0 + T1 + T2 + T3 + T4). While for a local thread when there is no migration, the time for a thread to acquire a semaphore will be simply T2 only. Hence, the extra overhead in this case is (T0 + T1 + T3 + T4).

We have conducted a series of experiments to measure the time it takes for a migrated thread to acquire a free Cooperative Semaphore remotely and that it takes for a local thread to acquire a free semaphore locally when there is no migration. A free semaphore means the semaphore is not currently hold by anyone else and a thread can acquire it immediately. In this case, the value for T2 will be the smallest possible. In addition, both the master and the slave thread are set to be the only active threads runnable on their respective node, therefore the time to wait before they can resume execution after data was arrived (T1, T4) will also be zero. It is found that the time it takes to acquire a remote Cooperative Semaphore for a slave thread in this way is about 1.20 milliseconds. And for the case for a local thread where it does not migrate, the time is about 16.29 microseconds. In other words, the ratio of the time to acquire a free semaphore remotely to that locally is about 74 : 1. By similar arrangement, we are able to determine the time for releasing a semaphore both remotely and locally. And the result shows that the corresponding times are about the same: it takes about 1.18 milliseconds to remotely release a Cooperative Semaphore and again 16.29 microseconds to release a semaphore locally. The time it takes to transmit the message between the nodes, the time for the local operating system to notify the JESSICA daemon that data is ready with `SIGIO` and the time to execute the `SIGIO` handler account for a major portion of the Cooperative Semaphore overhead. A point to note is that the times determined here are minimum values. In general, it will take some time for a thread to resume execution after it is rescheduled, since there can be other threads, with the same or higher execution priority, already running on the same node. Moreover, a semaphore may not be always available immediately when a thread tries to acquire it. Hence, T1 and T2 will be much longer. For example, in our recursive ray-tracing example to be discussed later, where threads are tightly synchronized, it is found that the time to acquire a Cooperative Semaphore increases from 27 to 84 milliseconds when more migrated threads are introduced into the system.

5.2 Analysis of Migration Latency

This section examines the migration latency.

Fig. 23 illustrates in details the interactions between the console and the worker node when a thread is migrated from the former to the latter. The migration latency is the time between the migrating thread is first frozen by the console and when it is restarted later as a slave thread at the destination, i.e. T0 + T1. Now T0 is the time taken to notify the destination node and to have the destination node prepare itself for the migration; the value of T0 is relatively constant. T1 is the time taken to marshal a delta set at the console node, to send the marshaled data across the network, and eventually to be de-marshaled at the destination node. The value of T1 is therefore proportional to the size of the transferring delta set. The migration latencies for different sizes of the delta sets are measured.

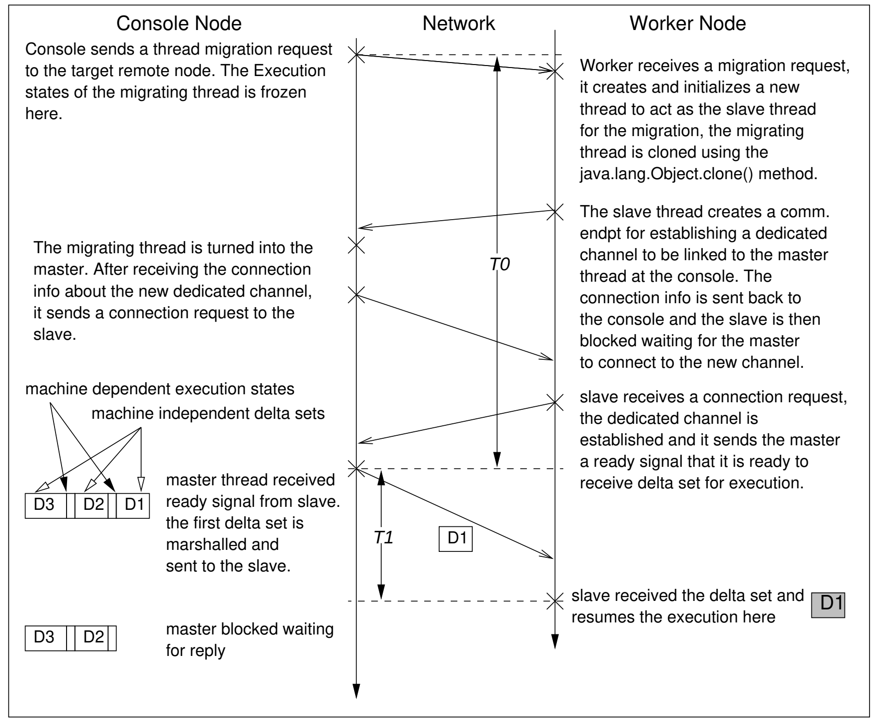

Figure 23: Interactions between the console and the worker during migration

According to the data collected, when the size of delta set is zero, the migration latency is about 24.55 milliseconds (T0). T0 is the time taken to execute the `java.lang.Object.clone()` method at the slave as well as that for sending the four handshake messages between the master and the slave, as shown in Fig. 23. A further break-down of this T0 value reveals that the time required to invoke the `java.lang.Object.clone()` method is about 17 milliseconds and that for a handshake message to be sent between the master and the slave is about 2 milliseconds. Now consider the case when a thread is migrated just before it starts executing the first instruction; the delta set contains only one JMF and the corresponding method stack is empty. If there is no local variable defined in the method, the size of this minimal JMF is 208 bytes. Hence, the minimum migration latency is about 28.12 milliseconds.

It can be seen that the minimum migration latency is dominated by the time for invoking the `java.lang.Object.clone()` method, which accounts for over 60% of the migration time. It is relatively expensive for the virtual machine in JESSICA to invoke a method like the `java.lang.Object.clone()`, when compared to classical migrating mechanisms which execute in native machine code. It is the intrinsic characteristic of JESSICA that a thread object cannot be duplicated by simply copying its memory content but a particular method has to be called to clone the object. Although this cloning method is expensive, it is only to be invoked once for migrating a thread, and there is no addition overhead in the later part of the Delta Execution process. Besides, despite this overhead we are still able to achieve good speedup by migrating threads to different nodes in the cluster for executing them in parallel, as demonstrated in [6].

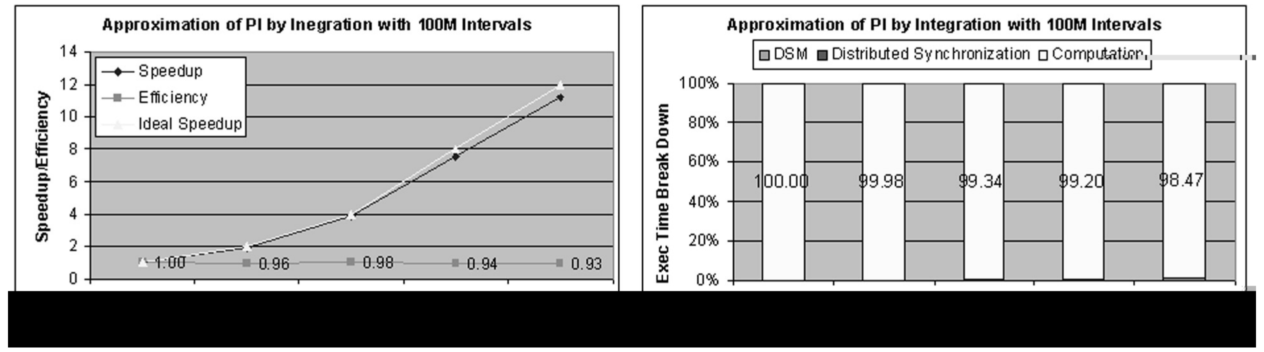

Figure 24: Performance results of the approximation of the value π with 100M intervals

5.3 Applications Performance

The following three multi-threaded Java applications have been devised to evaluate the performance of the system:

- π Calculation — This application approximates π by evaluating an integral. The area under the corresponding graph is divided into multiple regions and multiple threads are deployed to find the sub-areas. The value π is obtained by summing up all the sub-areas once all the threads have finished. This application shows the raw parallel performance of JESSICA since there is no interaction between the worker threads until all the computations are done; the extra overhead due to migration is minimal.

- Recursive Ray-Tracing — We have implemented a multi-threaded recursive ray-tracer in Java where the worker threads render the pixels of a projected 2D image by shooting rays into a given 3D scene. The threads obtain the next line of pixels to compute from a globally shared job queue, which provides a load-balancing effect at the application level. Since threads are tightly synchronized between themselves when they access the job queue in order to maintain its consistency, this application demonstrates how distributed thread synchronization affects the performance for the worker threads that are distributed around the cluster.

- Red-Black Successive Over-Relaxation (R/B-SOR) on a Grid — The R/B-SOR program creates multiple threads to compute matrix elements in parallel. A huge matrix is divided into two sub-matrices, the Red and the Black matrix, which are in turn divided into roughly equal size bands of rows, with each band assigned to a different thread. The threads repeatedly retrieve values from one matrix, compute the average, and write the result back to the other matrix. Since the huge matrix is allocated from the Global Object Space, the execution imposes a significant amount of loading onto the DSM subsystem. Hence, it is a good candidate for studying how the DSM overhead contributes to the overall execution time as a result of migration.

Each of the applications was tested on the 12-node cluster using 1, 2, 4, 8 and 12 processors with 1, 2, 4, 8 and 12 worker threads running respectively, and the results are presented as in the following figures.

According to Fig. 24, it can be seen that almost ideal speedup and efficiency are achieved in the π approximation application, since there is no communication or coordination between

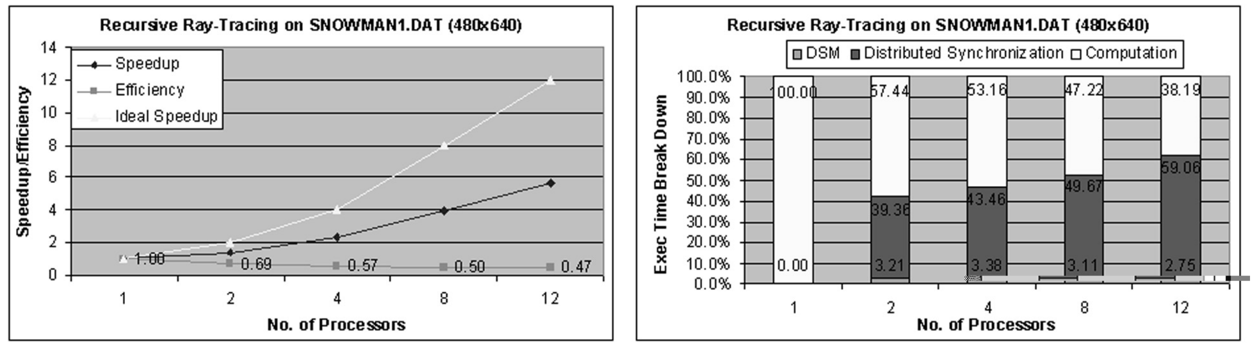

Figure 25: Performance results of the recursive ray-tracer to produce a 480x640 image

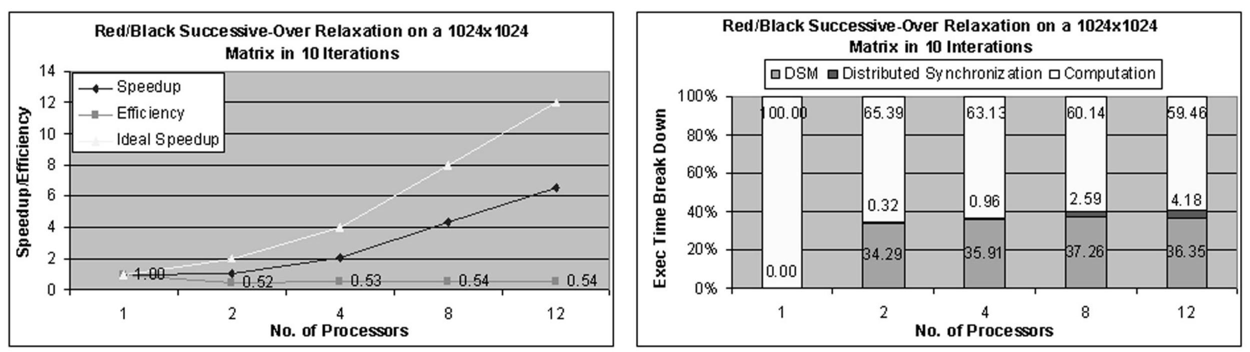

Figure 26: Performance results of the red/black successive-over-relaxation application on a 1024x1024 matrix

worker threads until all the computations are completed. The recursive ray-tracing experiment shows the efficiency is less than optimal and drops moderately as more processors are used. The efficiency decreases from 69% when using 2 processors to 47% when using 12 processors. This is because as indicated in Fig. 25, the percentage of distributed synchronization overhead contributes a significant amount to the total execution time. As shown in Fig. 26, the R/B-SOR application can gain moderate speedup when running with 4 or more processors. When running with 2 processors, the speed gained by overlapping the computation is offset by the extra overhead incurred due to remote memory access. The efficiency stays at around 53% and improves slightly when the number of processors is progressively increased from 2 to 12. This can be due to the fact that the amount of data shared, i.e. the sizes of the Red and the Black matrices, remain the same when executed by any number of processors, therefore the DSM overheads constitute roughly the same percentage of the execution time.

5.4 Conclusions

From the experiments conducted we can conclude that considerable speedup is possible by migrating threads to other machines in a cluster transparently and let them execute in parallel. In addition, the system remains to be responsive since the migration latency is sufficiently small. On the other hand, experiments in section 5.1 also show that the additional overhead that is a direct consequence of migration is significant. The DSM memory access penalty is about 9 times more than that of local access and the distributed thread synchronization also takes at least 75 times longer to complete when compared to local thread synchronization. The relatively high cost of migration overhead is demonstrated in the recursive ray-tracing and the R/B-SOR experiments, where their efficiency of parallelism ranges from about 70% to 50% when number of processors used are gradually changed from 2 to 12. Therefore, in order to benefit from the parallel execution capability of JESSICA, applications should be implemented in a way such that the performance gained by being able to overlap computation in multiple processors can out-weight the extra overheads incurred due to distributed object accesses and distributed thread synchronization. Here we provide a number of techniques that can improve the parallel execution performance of an application.

5.4.1 Reduce the Number of Distributed Object Accesses

To reduce the penalty due to accessing data in the DSM, we should try to avoid using object or class variables when defining variables that are of simple data type, we should try to use local variables whenever possible. In addition, If an object/class variable is going to be accessed for repeated number of times, we should use a local variable to cache its value so that further reading will be provided by the local variable. This is demonstrated in Fig. 27.

In this example, method `foo()` is supposed to accumulate the values in the object variable array and stored the result in another object variable sum. In each iteration of the for loop (Fig. 27, line 09), the method will check if the index `i` has reached the end of the array yet. As the length of the array is first cached in a local variable n (Fig. 27, line 07), the checking of the condition at line 09 can be completed much faster then the case when using the array.length variable directly. By the same token, when an object/class variable is being updated for repeated number of times, it will be more efficient if we store the updates in a local variable and save the final value in the object/class variable when all the updates are completed. This is also demonstrated in the `Bar` example: instead of adding the array

```
00 class Bar {
01   int sum;
02   int array[];
03
04   public void foo() {
05     int temp_sum = 0;
06     int myArray[] = this.array;
07     int n = myArray.length;
08
09     for (int i = 0; i < n; i++) {
10       temp_sum += myArray[i];
11     }
12     this.sum = temp_sum;
13   }
14 }
```

Figure 27: Class `Bar` that can minimize the number of distributed object accesses when migrated

elements directly into the sum variable every time the for loop iterates, the result is stored in a local temp_sum variable first (Fig. 27, line 10). The final value is only written to the sum variable when the for loop exit (Fig. 27, line 12). Finally, the last point to note is about indirect object accesses, that is 'reference of a reference' like objectA.objectB.varC. To access the content of varC, the virtual machine has to first load the reference of objectA into the stack, from which it will then obtain the reference of objectB by a `GETFIELD` instruction. With the reference of objectB on the top of the stack, consequently a `GETFIELD` is executed again to obtain the content of varC. `GETFIELD` is an expensive operation because it accesses the DSM to obtain the content of an object variable. The situation is worse if there are multiple levels of indirect object references. To reduce the overhead, the last reference that is able to access the targeted object variable should be cached first. In our example, we can define a local variable like ref_varC and initialize it with the value objectA.objectB. With this arrangement any further access to varC can make use of the reference stored in ref_varC directly and performance can be improved. This idea is also demonstrated in the `Bar` example in Fig. 27, the local myArray variable is used to cached the reference to the object's array (Fig. 27, line 06), so that its elements can be immediately accessed in the for loop (Fig. 27, line 10).

5.4.2 Reduce the Amount of Distributed Thread Synchronization

Since distributed thread synchronization is expensive, we should try to avoid it as much as possible. One of the possible ways is to enlarge the granularity of parallelism so that threads synchronize less often and more computation can be done between synchronization. This is because a too small granularity will cause the overhead of distributed thread synchronization to dominate the overall execution time. On the other hand, having the granularity too large will affect load balancing. When granularity is too large there may be time when some processors have already completed their tasks while others are still working. These idle processors will reduce the overall efficiency. In our recursive ray-tracing experiment, setting the granularity to be a line of pixels is a good starting point to tune the performance. We have also tried to set the granularity to be one pixel and found that the execution speed is slowed down dramatically, the more processors are used, and the more slow down we get. This is because when more processors are used, the overhead of distributed thread synchronization also increases, as discussed in section 5.3. In general, there is no perfect solution to tackle the granularity problem, usually careful tuning is required to find the optimal size.

6 Related Work

MOSIX [2] is one of the landmark systems that supports transparent process migration and follows the monolithic kernel approach. NOW MOSIX [1] uses the home model whereby user processes are created to run seemingly at the user's home workstation. Location transparency is maintained for migrated processes by redirecting any system calls that are location-dependent. However, considerable modifications have been made to the kernel in order to support the transparency. On the other hand, since Delta Execution is implemented at the middle-ware level, it enables the migration mechanism to focus only on the machine independent states in a thread execution context, and to leave the remaining machine dependent states untouched. As a result, the implementation of Delta Execution does not involve any low-level details of the underlying operating system or the hardware. Consequently, the implementation requires much less effort than that of MOSIX and is more manageable.

Java/DSM [15] is a distributed Java Virtual Machine that runs on a cluster of heterogeneous computers. It provides an illusion to Java applications that they are running on a single JVM with multiple processors attached. Parallel execution of a multi-threaded application is possible by having multiple threads running on multiple nodes in the cluster.

Both Java/DSM and JESSICA follow similar approach by implementing a distributed virtual machine at the middle-ware level. They utilize DSM systems to simplify their implementations. However, in Java/DSM load distribution is achieved by remote invocations of Java threads alone, while JESSICA supports also transparent thread migration. Besides, The current Java/DSM prototype focuses mainly on supporting DSM in a heterogeneous environment, other issues such as location transparency are not addressed.

The Millipede system [2] supports transparent thread migration and dynamic load sharing over a network of computers running the Windows NT operating system. Millipede is implemented at the middle-ware level that does not require any alteration to the operating system. There are two major components in Millipede. One is the DSM system that provides a global memory space for application instances to run over the network and to keep concurrent accesses made by them consistent. The other is the Migration Server who is responsible for collecting load information and making migration decisions.

In Millipede, for a worker thread that is running in an application instance, all its counterparts that are occupying the same memory slots as it is in other application instances have to be reserved. They cannot be assigned any jobs to execute because at any time this running thread may be migrated to one of their locations and have the thread execution states replaced. As a result, if there are M application instances running with N worker threads initialized in each of them, they will be occupying altogether M(N amount of active thread resources while at any time at most N threads will be running. On the other hand, in JESSICA all the object references stored in the execution stack of a Java thread are pointing to the Global Object Space. Consequently, the execution stack of a migrating thread in JESSICA can be copied to the destination without any special restriction on its new location, and there is no reservation required. In addition, migration in Millipede is not transparent as any threads that are occupying system resources cannot migrate.

	Level of Approach	Method of Load Distribution	Implementation Techniques	Characteristics	Support Migration or SSI Transparency
JESSICA	Middle-ware	Thread migration	DSM + Message redirection by helper threads	Execution by means of a virtual machine	Full SSI & migration transparency
NOW MOSIX	Monolithic kernel	Process migration	Message redirection by shadow process	Process migration following the home model	Full migration transparency
Java/DSM	Middle-ware	Remote execution of thread	DSM + data translation	Execution by means of a virtual machine, heterogeneous	Not supported
Millipede	Middle-ware	Thread migration	DSM + Migration Server	A pool of worker threads is reserved on each node	Limited migration transparency
Arachne	Application programming interface (API)	Thread migration	Provide a set of migration related routines to be included into applications	Heterogeneous migration, no data sharing between migrated threads	Not supported

Table 2: Comparison of characteristics between JESSICA and the related work discussed

Arachne [4] is a portable user-level programming library that supports thread migration over a heterogeneous cluster. However, migration is not transparent to application programmers as they have to include the Arachne migration-related primitives into the code. It is the programmer's responsibility to decide when to migrate, which thread to migrate, and where to migrate. In addition, it introduces new keywords to the C++ programming language to facilitate the implementation of thread migration. Even though the primary objective of Arachne is to support efficient thread migration on a heterogeneous network, it lacks certain features such as thread synchronization that is fundamental to a thread package. Besides, migrated threads cannot share data.

JESSICA provides a more flexible, portable, efficient and useable thread package for application programmers than Arachne. Firstly, migration is entirely transparent to programmers in JESSICA, there is no migration primitive that programmers have to insert into their code. Secondly, JESSICA supports thread synchronization and inter-thread signaling even if the threads are residing on different machines. Migrated threads can share object of any data types. Thirdly, JESSICA is more portable as it does not introduce any new keyword to the Java programming language.

7 Conclusions and Further Work

Our implementation experience with the JESSICA prototype and the experiment results have shown that establishing an SSI illusion to support transparent Java thread migration in a cluster environment is not only feasible but also beneficial. The design of the Java programming language does not in particular impose any limitation that would hinder such a design and implementation. On the other hand, it is the characteristics of the language, such as bytecode execution, thread as the first class citizen, and simple model of inter-thread signaling and synchronization that have simplified the implementation of JESSICA. Consequently, we are able to arrive at an implementation that does not introduce any changes to the Java programming language and still is able to support thread migration, to achieve SSI, and to provide a parallel execution environment with good efficiency.

We are in the process of porting the JESSICA prototype to a Linux-based cluster which is equipped with the Direct-Point fast communication subsystem [6] and the JUMP DSM subsystem [3], which would enable us to investigate the effect of using different shared-memory consistency models on the performance of JESSICA.