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

Ma Jin Ming

A thesis submitted in partial fulfillment
of the requirements for
the degree of Master of Philosophy
at the University of Hong Kong.

February 1999

Abstract of thesis entitled

"JESSICA: Java-Enabled Single-System-Image Computing Architecture"
submitted by

Ma Jin Ming

for the degree of Master of Philosophy

at the University of Hong Kong

in February 1999

The Java programming language has taken the entire computing community by storm since its introduction in late 1994. Apart from its object-oriented and network-centric characteristics, Java also supports the multi-thread model for concurrent programming as a built-in feature of the language. However, not until recently with the release of Java 2 could multi-threaded applications achieve true parallel execution as the implementation of the previous virtual machines did not support the feature, even if the underlying hardware is a symmetric multi-processor machine.

On the other hand, cluster computing is gaining overwhelming acceptance in recent years because it offers a viable and inexpensive alternative to supercomputers for parallel execution of applications. While popular communication libraries like MPI are available for programming parallel applications in a cluster, they follow the message-passing programming model which is considered more difficult to handle than the shared-memory model.

This thesis presents JESSICA which combines the merits of both technologies; namely the parallel execution capability of clusters and the simple thread model of Java for concurrent programming. JESSICA stands for 'Java-Enabled Single-System-Image Computing Architecture'. It is a middle-ware that runs on top of the standard UNIX operating system and transforms a cluster of computers into a single system with multiple processors, a single and contiguous memory space and other unified resources. It provides a Global Thread Space that hides the physical boundaries between machines, and makes a cluster appear as a single computer to applications. 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, migration transparency is achieved by redirecting any location-dependent operations to the appropriate machines for execution. The result is a parallel execution environment where threads can freely move across machine boundaries.

JESSICA supports the Serial-Program-Parallel-Subsystem programming paradigm such that application programmers no longer need to worry about the number of processors available. The programmers create as many threads as needed and the system will automatically migrate the threads within the cluster to maximize parallelism. Since JESSICA conforms to the Java Virtual Machine Specification, the vast number of existing Java applications can run on the system immediately and gain speedup. By providing the favorable shared-memory model for writing parallel applications in Java, JESSICA can promote the Java programming language to become the language of choice for parallel application development in the future.

To prove the JESSICA concept, a working prototype has been implemented on a 12-node cluster, along with a number of multi-threaded applications that are commonly found in the parallel computing literature. Experiments show that considerable speedup is achievable in all the applications, with efficiency ranging from about 95% to 50% with 2 to 12 nodes. Furthermore, Delta Execution offers fast migration capability with a migration latency of only about 28 milliseconds.

Declaration

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma, or other qualification.

_____

Ma Jin Ming

February 1999

Acknowledgements

I would like to take this opportunity to express my gratitude towards the following people. First of all, I would like to thank my supervisors, Dr Francis Lau and Dr Cho-Li Wang for their invaluable guidelines and perpetual patience. I would also like to thank the early members of the System Research Group, Anthony Loong and Carl Yau, for providing me with many stimulating ideas for doing research. My vote of thanks also goes to Doug Kwan, Tim Yeung, Barry Will, Thomas Li and Paul Chow for their kindest help and support. Last but not least, I would like to thank Katherine Lau and Roderick Ma for their assistance in the preparation of the thesis draft.

Contents

List of Tables and Figures

Chapter 1

Introduction

1.1 Cluster Computing

Topics in Cluster Computing have been under active research in recent years. A cluster of computers is a federation of computers linked by an interconnection network that are running an integration software for performing collaborative computation. The integration software allows computers to coordinate their activities and to share resources within the system, such as CPU cycles, data residing in memory, file storage, et cetera. Many promising results have been reported on using clusters of computers for load sharing and parallel computing [23, 29, 30 and 41]. With the advent of high-speed networking and microprocessor technologies, Cluster Computing has emerged as a favorable alternative to Massively Parallel Machines like the Cray T3D and the IBM SP2 for high performance and large-scale computing. Clusters are scalable and they are constructed from affordable, off-the-shelf component hardware.

In general, computers belonging to a cluster are loosely coupled and they do not share memory. As a result, parallel programs developed to run on a cluster usually follow the message-passing programming model. In this model, each computer can only access data that are stored in its local memory, non-local data are obtained as messages being sent from remote nodes. As a result, parallel computation proceeds as the participating nodes exchange messages between them. While the message-passing model matches the No-Remote-Memory-Access (NORMA) characteristic of a cluster, it is generally agreed that programming in this model is more difficult than the shared-memory model as the latter is closer to the Von Neumann model for sequential programming, a model which is well understood by programmers. In the shared-memory model, activities between the processors are coordinated through updating a region of memory that is shared by all. Mechanisms to enforce mutual exclusion are provided for accessing the shared memory to ensure data consistency. Despite of the favorable shared-memory model, most of the programming libraries available today for a cluster are designed for message-passing, such as MPI [38].

1.2 Java and Parallel Computing

The Java Programming Language [16] has been receiving unprecedented acceptance and support since its introduction in late 1994. It is being studied and deployed by a very broad range of users and application developers. Many colleges have already adopted Java as the first language to teach in their beginners' courses for computer programming. We envision Java as the language of choice for most kinds of application development. In addition, the following features of Java also make itself a favorable tool for distributed computing:

- Multi-threaded programming is directly supported and is part of the language.
- A comprehensive set of BSD socket API for inter-process communication is included in the network extension of the language.
- The object-oriented nature of the language allows incorporation of new communication libraries by implementing their corresponding classes. Existing library implementations can be reused immediately by linking them to any Java applications using the Java Native Interface (JNI).

However, whether to execute multi-threaded Java applications over a cluster or on a symmetric multi-processor (SMP) machine, neither case is in fact capable of reaping the maximum processing power delivered by the underlying system. Although the Java Programming Language has multi-threaded support, SUN only provides a green thread implementation in its Java Virtual Machine. Hence it is not possible to map multiple Java threads to multiple processors, and Java threads still run sequentially and are bounded to a single processor (Fig 1.1a). (This is the case for JDK releases prior to version 1.2 [44]. Version 1.2 supports native-thread on SMP machine (Fig 1.1b). It was released in December 1998 and renamed to Java 2.) To span across multiple machines and achieve real parallelism, Java programmers currently have to tackle the coordination between processing nodes at the application level, through some IPC mechanisms such as sockets. Since the introduction of JDK version 1.1, the burden on the programmers is alleviated by the provision of Object Serialization [21], Remote Method Invocation (RMI) [22] and the Object Request Broker (ORB) [20] support. They allow coordination and cooperation of processes at the function call level, through some remote procedure call (RPC) like mechanisms (Fig 1.1c). 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.

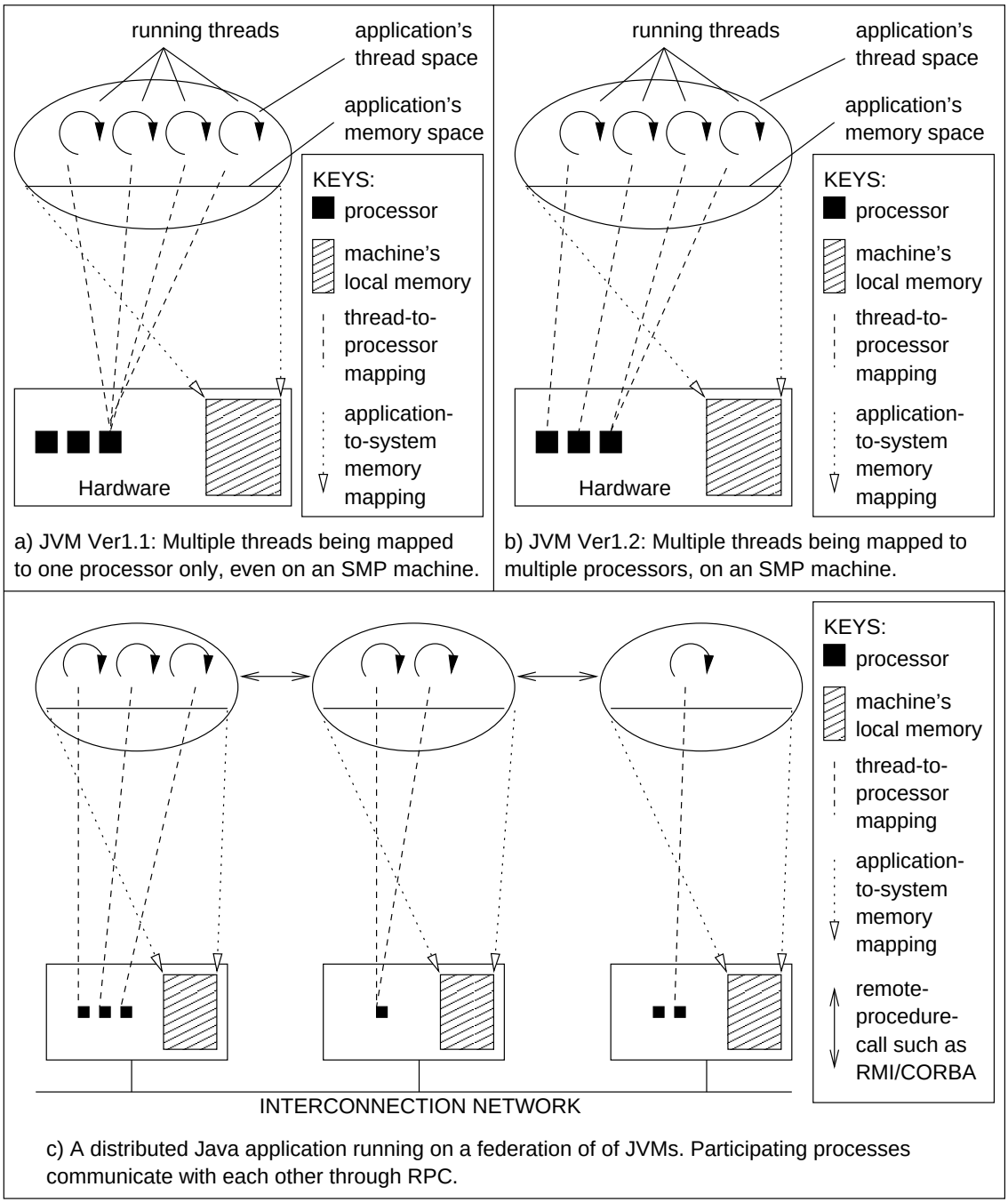

Fig 1.1: Limited parallel execution support in Java Ver1.1 and Java 2 (Ver1.2)

Having observed the advantages and limitations of Cluster Computing and those of the Java Programming Language, here we propose JESSICA. JESSICA combines the merits of both technologies. It offers a parallel execution environment over a cluster of computers and supports the shared-memory programming model using threads.

1.3 What is JESSICA

This thesis discusses our work on JESSICA, our solution that addresses the issues mentioned previously using the Single-System-Image (SSI) approach. JESSICA stands for 'Java-Enabled Single-System-Image Computing Architecture', a platform for executing multi-threaded Java applications [5] over a cluster of computers running the standard UNIX operating system. It is an integration-software that allows multi-threaded applications to exploit the maximal parallelism obtainable from the cluster. JESSICA is implemented on top of the UNIX operating system and is therefore portable. It acts as a middle-ware that encapsulates the distributed nature of the cluster hardware. It provides the applications with an illusion that they are running on a single multiprocessor-computer, that is, a Single-System-Image.

The Single-System-Image 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 over the cluster, for the containment and execution of threads to be created by the application. 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 any location dependencies are taken care of by the JESSICA system. 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 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.

In order to optimize resource utilization in the cluster, JESSICA supports the Serial-Program-Parallel-Subsystem (SPPS) computing paradigm [14] where parallel execution can be achieved by simply creating as many Java threads as needed, just as the case when it is running in a single execution environment. The system can migrate threads around the cluster automatically to maximize parallelism. It handles thread-to-processor mapping transparently and maintains the integrity and consistency of shared resources, which may be distributed all over the cluster (Fig 1.2).

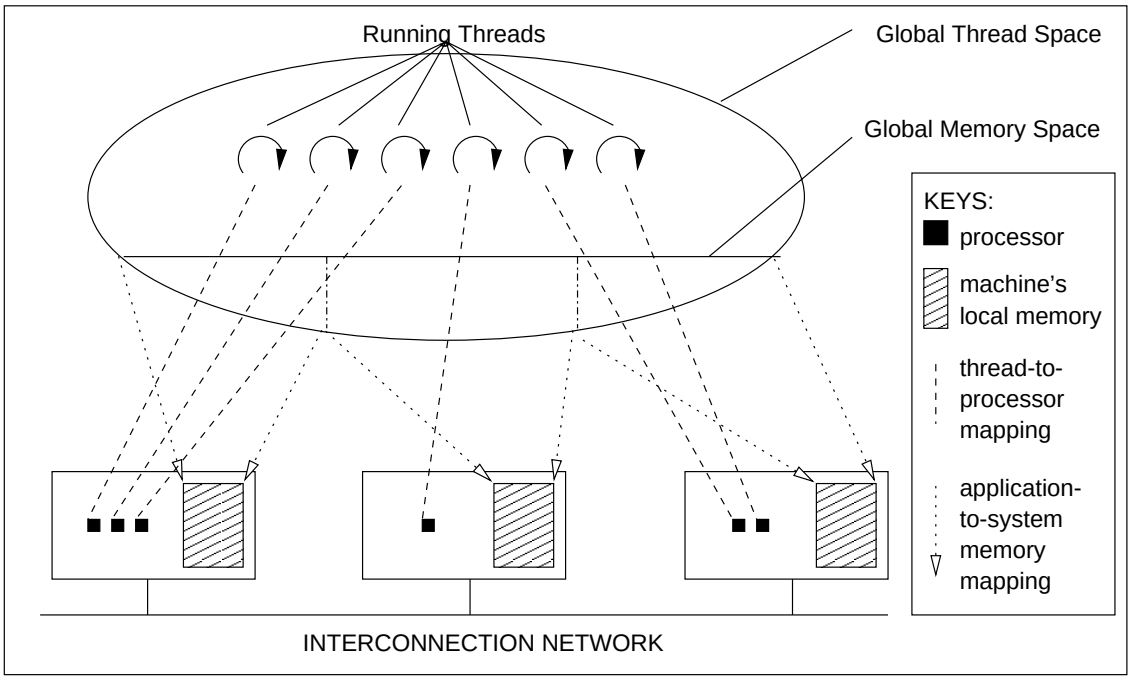

Fig 1.2: JESSICA's SSI approach: A Global Thread Space

The main characteristics of JESSICA include:

1.3.1 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 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, since the thread migration mechanism helps to exploit the maximal parallelism and to optimize utilization of shared resources.

1.3.2 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 stopped at any time, be migrated to another processor and be resumed 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, which

ensures the greatest portability and immediately migratability of any existing Java applications. A detail discussion of Delta Execution is provided in Chapter 5.

1.3.3 Support Migration Transparency

Migration transparency is upheld in JESSICA by leaving helper threads (also known as master) at the source 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. Detail discussion on how to support migration transparency and the implementation of Cooperative Semaphore is presented in Chapter 6.

1.3.4 Support the Shared-Memory Programming Model

JESSICA is able to hide the distributed nature of the cluster and to unify the local memory region attached to each processor to produce a uniform Global Object Space shared by all the processors. Consequently, programmers are able to follow the multi-threaded programming model that is a direct extension of the shared-memory model for developing parallel applications. The multi-threaded model is considered simpler and less error-prone. Furthermore, since JESSICA does not introduce any new computational model, parallel applications developed on JESSICA are, in fact, standard multi-threaded applications that are also runnable on standard uni-processor virtual machines, and vice versa.

1.3.5 Existing Applications Ready to Run in Parallel

Since the implementation of the JESSICA system conforms to the Java Virtual Machine Specification [36], the vast number of existing multi-threaded Java applications require no modification to run on JESSICA. They can benefit from the parallel execution environment JESSICA offers and gain speedup immediately.

1.3.6 Extendable to a Heterogeneous Cluster

The Java features of machine-independent bytecode and virtual machine execution mean that it is possible to extend JESSICA to run on a cluster of heterogeneous computers. It can include hardware of various platforms such as PC compatibles, workstations and symmetric multi-processing machines (SMP). In fact, the design and implementation of the Delta Execution have already taken care of heterogeneity so that only the machine independent execution context of a Java thread is migrated. The limitation in the current implementation is that the DSM subsystem cannot work over heterogeneous hardware. To support a heterogeneous cluster, the current implementation can either replace the DSM subsystem with

one that supports heterogeneous hardware, or to implement an address translation service such as the one described in Java/DSM [39].

1.3.7 Designed for Java

JESSICA supports one of the most popular programming languages of the 21st Century, Java provides a parallel execution environment for a wide range of Java applications that are developed or currently under development. At the same time, the huge developer base of Java can also help to promote the acceptance of JESSICA and to enlarge its installation base.

1.3.8 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 Sprite [12], which can be much more tedious and time-consuming.

1.4 Contributions of the Thesis

JESSICA is the first system that supports transparent Java thread migration and provides a Single-System-Image over a cluster of computers. It offers a viable, high-performance and yet low-cost platform for parallel execution of Java applications. The contributions of our work are listed as follows:

1. JESSICA provides a parallel execution environment that does not introduce any new parallel computational model, programming API, or language construct. Because of the Single-System-Image encapsulation, existing multi-threaded Java applications are immediately runnable and ready to reap the parallel-processing capability available from the cluster.
2. Delta Execution provides a transparent and preemptive Java thread migration mechanism, which can help to exploit the maximal parallelism in the cluster and optimize the use of computing resources.
3. The concepts of Delta Execution and the Master/Slave design for supporting migration transparency is language-neutral and platform-independent. They can be applied to other programming languages that execute by means of virtual machines for implementing transparent thread migration.

4. The JESSICA prototype and its thread migration capability are readily extendable to establish a Single-System-Image over a heterogeneous cluster. This is a significant step forward to achieving the goal of Global Computing, as visualized in the Millennium [40] and the Legion Projects [17].

1.5 Thesis Organization

This thesis is made up of 8 chapters. Chapter 1 is the introduction. Chapter 2 presents the background and issues on providing Single-System-Image and thread migration in a cluster. Besides, the JESSICA approach is also discussed in the chapter. Chapter 3 describes approaches taken by related research that tackle thread migration and Single-System-Image. Chapter 4 gives the system architecture of JESSICA in details. Chapter 5 introduces Delta Execution, our solution to preemptive Java thread migration. Chapter 6 discusses how Single-System-Image transparency is supported in JESSICA. Chapter 7 evaluates the performance of JESSICA by studying a number of experiments performed. Finally, Chapter 8 concludes our implementation experience with JESSICA and gives directions for future work.

Chapter 2

The JESSICA Approach

2.1 Overview

JESSICA is designed around the concept of Single-System-Image (SSI), to provide application with an illusion that they are running on a single computer system. This chapter presents some background of the SSI concept and the issues of concern when implementing it on a cluster. Furthermore, the approach that JESSICA has taken to achieve SSI will also be discussed. Apart from providing an SSI illusion over a cluster, another JESSICA's goal is to improve resource utilization in the cluster. JESSICA achieves this goal by implementing a transparent thread migration mechanism where threads can move to underload nodes and continue execution there. The issues for implementing thread migration and the JESSICA approach to thread migration will be dealt with in the later part of the chapter.

2.2 Single-System-Image and its Levels of Abstraction

In [14], Pfister believes that the definition of Single-System-Image (SSI) should be part of the definition of a cluster: *A collection of machines is not a cluster until it is used as a single computing resource, and SSI is what makes a cluster a cluster*. Single-System-Image is an important concept in Cluster Computing for it can simplify the implementation of the operating system, the applications and even management of running jobs in the cluster, depending on which level the SSI illusion is provided. The idea behind SSI is to encapsulate the system resources below it and to create a layer of abstraction, such that the components sitting on top of the layer can view the encapsulated resources as a single and unified entity. By establishing a logical boundary that encloses the components lying on top of the abstraction, the distributed nature of the resources located below the layer of abstraction is hidden by the SSI from the components above. As a result, the unified view, that is, the logical boundary provided by an SSI, is dependent on which level in the system the abstraction is made. Pfister [14] identifies that in general, four different levels of abstraction can be applied to a cluster. They are the hardware-level, the kernel-level, the middle-ware level and the application level, each of them provides a different SSI boundary that are perceived only by components that it encloses.

2.2.1 Hardware Level

At the hardware level, the lowest level resources such as memory and I/O devices can be linked together by special circuits to provide a unified memory and I/O device space. SSI at this level is usually found on symmetric multi-processor machines (SMP). The operating system layer immediately above the hardware would perceive the hardware to have a continuous memory space and a set of globally accessible I/O devices. In this case all the software layers, including the kernel, operating subsystems and the applications are enclosed by the SSI boundary. Note that the SSI abstraction provided is at the lowest level.

2.2.2 Kernel Level

SSI at the kernel level means the kernel implementation is responsible for hiding the distributed nature of the hardware and providing the unified illusion to the software subsystems and applications above. Any system calls executed by the subsystems and the application code have to provide the same consistent and coherent SSI illusion. It is believed that providing SSI at this level is the most desirable. Now that the kernel has the best knowledge of all the activities and resources that are distributed around the cluster, it can perform resource management in the most effective manner and the system resembles an SMP kernel. On the other hand, SSI at this level is also the most difficult to achieve. It is because the system has to maintain the consistency of huge amount of state information that is distributed around the cluster and to provide transparency to access at the same time. Traditional operating systems, such as UNIX, are not designed with the SSI concept in mind, extending them requires substantial skill and expensive software effort. Contemporary message-passing kernels such as Mach [7] and Amoeba [34] are proved to be a more viable choice because the most critical task one has to implement is to extend the message-passing mechanism to function across machine boundaries transparently, and the result will imply a cross-machine SSI.

2.2.3 Middle-ware Level

Middle-ware includes software components that are not integral parts of the operating system and provide desirable or necessary services to application programs. SSI at this level is available to applications when they make use of the services offered by the middle-ware. The SSI boundary in this case is defined by the services offered. As long as applications only use the services offered by the SSI middle-ware, they will see a single-system. An example of SSI implementation at this layer is the SUN's Network File System (NFS). The SSI implementation at this layer may not be as comprehensive as that at the kernel layer because some of the necessary state information may have hidden under the kernel and is not accessible. On the other hand, since the middle-ware approach can focus and work on one

subsystem at a time, with visible gain from each increment rather than working on all aspects once and for all, it is considered more manageable than the kernel approach. Besides, this approach is more promising because it does not require any modification to the operating system below or to the applications above which make use of its services.

2.2.4 Application Level

This is the highest level where the SSI abstraction can be installed. An application encapsulates the distributed nature of the underlying system and exports an SSI illusion to the end-users who operate on the application. The SSI boundary in this case is therefore the application itself. An example of this approach is the batch job submission system. Nevertheless, there are some drawbacks in this approach. The applications developed from it are less portable and existing applications would require modifications, which may not be feasible, to incorporate the SSI abstraction.

2.3 The JESSICA Approach

An important characteristic about the Java Programming Language is that applications are executed by means of a virtual machine called the Java Virtual Machine (JVM). A JVM runs as a user process on a standard operating system and interprets the virtual machine instructions (called bytecode) that constitute the application. This virtual machine approach enables cross-platform compatibility for Java applications because there can be a specific implementation of JVM for each platform.

This virtual machine approach is considered as an advantage to JESSICA for achieving SSI over a cluster. This is because the virtual machine provides a means for JESSICA to implement SSI at the middle-ware level, to avoid any modification to the underlying operating system in order to improve portability. Furthermore, the virtual machine approach makes the construction of a heterogeneous cluster possible, different versions of virtual machines running on different hardware platforms can collaborate and create a single, logical execution environment for running applications.

Hence, the JESSICA approach is to implement an SSI at the middle-ware level in the form of a distributed virtual machine. There is a distributed virtual machine daemon running on each node of the cluster, they work together to create a single computer illusion for Java applications. The SSI boundary created by the distributed virtual machine is a Global Thread Space that spans across the cluster and hides the physical boundaries between nodes in the cluster. Threads created by an application live inside this Global Thread Space, the system

can freely move them around and bind them to the processors of any nodes for execution. A Global Object Space is also installed within the Global Thread Space that allows threads to access any memory objects independent of their current location. As a result, even if threads are running on separate hardware, they will be able to share data just as they were running on the same machine. In addition, threads living in the Global Thread Space can signal and synchronize with each other no matter where they are; their physical locations are transparent to themselves.

2.4 The Challenge

To establish an SSI view at the middle-ware layer, the challenge is to maintain the same virtual machine semantics within the Global Thread Space when spanning across the machine boundaries in the cluster. The distributed virtual machine has to deal with the concurrency and system resources that are offered by each operating system running on the cluster of computers. These resources, such as CPU cycles and system memory, are to be collectively managed by JESSICA. JESSICA is responsible for repackaging the distributed resources and offering them as system services to the threads executing within the Global Thread Space, in a manner that resembles the system services offered by a standard JVM running on a single computer. At the same time, in order to maximize resource utilization, JESSICA is also required to exploit the parallelism and the multiple system resources that are offered by the underlying cluster (Fig 2.1).

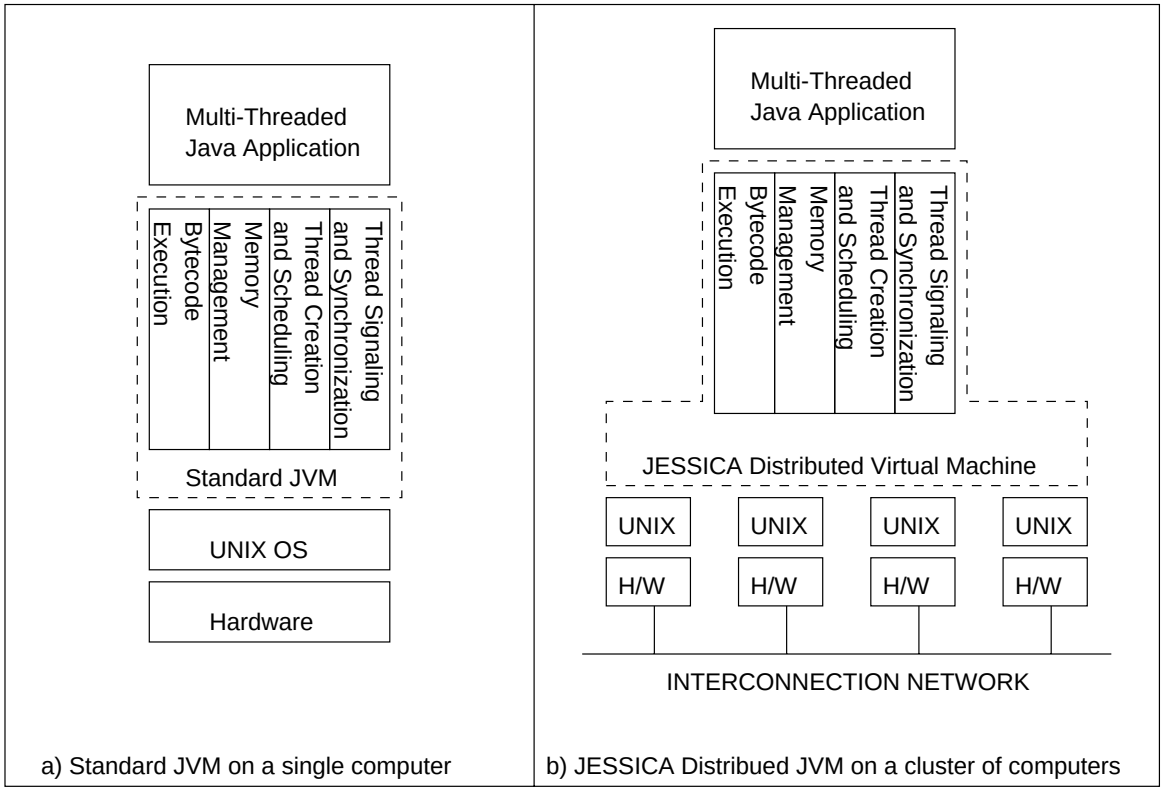

Fig 2.1: JESSICA provides the same system services as a standard JVM and at the same time exploits the parallel execution capability of the cluster

2.5 The JESSICA Solution

There are four basic system services that are provided by the standard JVM; namely, bytecode execution, memory management, thread creation and scheduling, and thread synchronization and signaling. The JESSICA approach in dealing with each of the services are presented as follows:

2.5.1 Bytecode Execution

This is to provide an execution engine for the execution of virtual machine instructions. The execution engine is a stack-based machine where each thread has its own associated execution stack for the engine to operate upon. JESSICA provides an independent bytecode execution engine on each node of the cluster. Multiple execution engines can proceed simultaneously by having one execution engine operating on the execution stack of one of the threads. This scheme forms the basis for parallel execution in the cluster.

2.5.2 Memory Management

This service is responsible for allocating memory for object creation requests, to keep track of all objects in the system and to reclaim memory space of unused objects by garbage collection. JESSICA provides a Global Object Space by employing a distributed-shared-memory (DSM) system that combines the system memory available on all the nodes. The Global Object Space allows the same operations to be used for accessing both local and remote memory objects transparently.

To maximize parallelism, JESSICA follows a decentralized approach whereby each node is responsible for managing its own share in the global memory. An object allocation request at a given node will be satisfied locally with a memory chunk obtained from the same node, while at the same time this object is accessible by threads running on other nodes. The system now has to implement a distributed garbage collection algorithm because objects may be referenced locally as well as remotely.

2.5.3 Thread Creation and Scheduling

A standard JVM is responsible for creating new thread instances and map them to the processor for execution, in addition to thread scheduling. JESSICA has to provide similar services. The Global Thread Space created by JESSICA offers a global name space for threads. In other words, threads created on any nodes in the cluster are labeled with unique identifiers. A decentralized scheme is also adopted for thread scheduling, threads residing on each node are scheduled locally by the local thread manager.

Load distribution in the cluster is achieved by a preemptive thread migration mechanism called Delta Execution. The execution states of a migrating thread is broken down into multiple units of states called *delta sets*. These delta sets are sent to the destination node in an incremental manner where the execution states of the migrated thread proceeds by *delta* every time a set is sent. Hence the name Delta Execution. With the help of such migration mechanism, thread managers running on each node can work together to perform thread-to-processor mapping. Threads running on an overload node can be migrated (re-mapped) to other underload nodes to improve the overall resource utilization. More discussion on load distribution in JESSICA will be presented later in the chapter.

2.5.4 Thread Synchronization and Signaling

In a multi-threaded execution environment, the standard JVM as well as JESSICA have to provide a mechanism for threads to synchronize with each other and to communicate. It is particularly important to maintain consistencies of shared objects because multiple threads

may try to update a shared object at the same time, producing a race condition. Thread synchronization in the standard JVM is supported by semaphore. Moreover, a thread can deliberately wait for another thread and block itself, the other thread can notify and resume the waiting thread by sending it a signal.

The Global Thread Space provided by JESSICA extends the semantics for thread synchronization and signaling to work even when threads are physically residing on different nodes. When a thread running on a node is trying to acquire a semaphore of a shared object, this semaphore may have been acquired already by another thread that is running on another node in the cluster, the first thread is therefore blocked. However, when the second thread has finished with the shared object, it will release the semaphore which then will cause the thread manager in the first node to wake up the first thread. Similarly, when a thread tries to notify another thread, the system will automatically check if the other thread is residing on the same node, if not, the signal will be directed to the right node and be delivered to the other thread there.

2.6 Augmenting System Classes

Apart from the standard system services described above, the standard JVM allows the provision of extended system services by implementing additional packages of system classes. For example, in the core Java class library [3] the graphical window subsystem is implemented in the `java.awt` package, the BSD socket communication subsystem is implemented in the `java.net` package. For better performance and portability, these system packages are usually native implementations that directly access system services provided by the underlying operating system, such as the `java.awt` package for UNIX is implemented on top of the X-Windows system. This modular approach also benefits our implementation of JESSICA. When we are trying to extend the system classes to work on the cluster, we can focus on one package at a time, to incorporate the SSI feature into the package and to make it workable on the cluster. In addition, this approach also permits us to rework the internals of the package implementation without changing the interface of the package, thereby providing transparency and portability to applications.

If a set of the system classes provides location-dependent services to applications, augmentations to their implementations are necessary to uphold migration transparency. Examples of such classes are the `java.io` package and the `java.net` package which make use of the I/O services offered by the operating system and in turn provide file and

network I/O to applications. JESSICA adopts the technique of message redirection to maintain transparency for location-dependent services. To support message redirection, a helper thread called the master is left behind at the source node where a thread has migrated. The master will then perform any location-dependent operations on behalf of the migrated thread which is identified as its slave. The system classes are augmented such that before a thread tries to perform a location-dependent operation, it will first check if the current thread is a migrated thread. If the current thread is a slave, then instead of performing the operation directly, it will contact its master at the source node by sending it a message and the master will perform the operation accordingly at its location. In addition, the master at the source node is also responsible for forwarding messages to the slave, such as signals, being sent to the latter by other threads in the Global Thread Space. A detail discussion on the implementation of the master/slave design for the helper/migrated thread and that of the message redirection are to be found in chapter 6.

An example of location-dependent operation is the `<currentTimeMillis()>` method provided by the `java.lang.System` class. The method returns the current system time in milliseconds by invoking the `<gettimeofday()>` system call provided by the underlying operating system. This operation is location-dependent because the system time of nodes in the cluster may not be synchronized. If a thread of an application running on node A queries the system time and after a while, this thread is migrated to node B to perform the same query there, then it is possible that the thread may observe the system clock to be 'going backwards'. This happens when the system clock in node B is substantially slower than that in node A. To avoid such a scenario from happening, all the system time queries are therefore redirected back to the node where the thread comes from. Bear in mind that all the modifications necessary are made to the implementation of `java.lang.System` class alone, without changing its interface or any other classes.

2.7 Other Design Issues

In [43], Xu, Wang and Hwang identify the following additional issues when designing SSI:

2.7.1 Symmetry

Applications can be executed from any node of the cluster and obtain the same result. The level of illusion, services and functionality are all symmetric when viewed from any node in the cluster; except those which are protected by the usual access rights, if any.

In JESSICA, symmetry is realized within the Global Thread Space as the same system services are offered to applications no matter on which node they are started. By deploying a DSM system, the same Global Object Space is observed by threads independent of their locations.

2.7.2 Transparency

Applications can access system resources from any location. The issue can be further sub-divided into location transparency and network transparency. Location transparency means applications can access the system resources without knowing where the physical locations of the resources are.

In JESSICA, the Global Object Space provides a global name space for objects and threads where applications cannot derive their physical locations from the reference handles. Network transparency means communication channels are adaptive and self-adjustable such that the channels still sustain themselves even when one of the communication end-points of the channels that are bound to the system has changed. JESSICA adopts a master/slave design for the helper/migrated thread pair. It can redirect any location-dependent operation automatically, including network communications, to the correct place for execution irrespective of where a thread has migrated.

2.7.3 Single Entry Point

The system provides the same environment on all the nodes for end-users to start their applications. No matter where the application is invoked, the system will select automatically the best node to execute the application so as to balance load.

JESSICA supports the concept of single entry point. The transparent thread migration mechanism of JESSICA together with a simple thread migration policy can automatically perform thread-to-processor re-mapping if load imbalance occurs after the application is started.

2.7.4 Single File Hierarchy

The cluster provides the illusion of a single file system by integrating all the local and global disks and other file devices into one file hierarchy. This issue is partially solved by the available distributed file systems such as the SUN's Network File System (NFS).

Since JESSICA runs on top of the standard UNIX operating system, it can make use of the distributed file system provided by the operating system directly. The JESSICA prototype is currently running on an environment where NFS is available.

2.7.5 Single I/O System Image

The cluster should have a single I/O space and a single networking. In other words, all the I/O devices and network interfaces attached to the nodes of the cluster are uniformly accessible as if they were attached to one single computer. To achieve this, there should be some kind of scheme by which all the devices and interfaces can be addressed independent of their locations, or a uniform device naming as proposed in Solaris MC [42]. In addition, a data forwarding mechanism should also be available for redirecting data between two nodes when a device in node A cannot be accessed by node B directly.

Although JESSICA does not support single I/O space and single networking at the moment, implementing such a mechanism is feasible. For example, we can install a lookup table within the distributed virtual machine that maps uniform device names to the corresponding location-dependent device names. Moreover, the message redirection mechanism is already in place in JESSICA.

2.7.6 Support for Fast Message Passing

There should be some kind of fast message passing mechanism between nodes in the cluster that can act as a reasonably fast internal data bus in the virtual single computer. Since JESSICA runs on top of standard UNIX, the implementation can make immediate use of the existing fast message passing package if installed, such as the Directed Point [24].

2.7.7 Support for Job Management

An SSI cluster should provide a global job management system for load sharing and improve resource utilization in the cluster. JESSICA supports transparent thread migration and when combined with its simple thread migration policy, the system can support dynamic load balancing across the cluster to optimize resource utilization. Another possible extension to JESSICA is to implement a Java shell for job management to support multiple Java applications to run simultaneously on JESSICA just like a multi-computer. In order to achieve this some kind of domain protection mechanism is required for individual application processes co-existing in the same execution environment.

2.8 Load Distribution in JESSICA

As discussed in section 2.7.3, one of the SSI implementation issues is how to support a single entry point for applications, where the system can select automatically the best node to execute the application so as to avoid any load imbalance, no matter on which node the application is invoked. JESSICA supports a single entry point by implementing a preemptive thread migration mechanism called Delta Execution within the Global Thread Space. The system can migrate threads running on an overload node to other underload nodes transparently without the application's knowledge.

The goal of JESSICA is to optimize resource utilization by supporting the Serial-Program-Parallel-Subsystem (SPPS) computing paradigm. This SPPS feature of JESSICA frees application programmers from worrying about the actual number of processors available. They can simply create as many threads as necessary in their applications, the JESSICA system can migrate threads around the cluster automatically to maximize parallelism. To support the SPPS paradigm, apart from a preemptive and transparent thread migration mechanism, a thread migration policy is also enforced so that the system knows when, where and which threads to migrate across the cluster.

2.9 Issues for Implementing Thread Migration

In [6], Milojicic, Douglis, Paindaveine, Wheeler and Zhou identify the following 6 issues for consideration when implementing thread or process migration in a cluster of computers.

2.9.1 Complexity

This issue gauges how complex or how much software effort is required to implement a thread migration mechanism. Similar to SSI, implementation of thread migration can also be carried out on different levels; namely, the micro-kernel level, the kernel level, the middle-ware level and the application level. It is agreed that complexity increases when moving from the application level to the kernel level, except on the micro-kernel level where simple abstractions and minimal functionality reduce the complexity.

As JESSCIA executes applications by means of a virtual machine, Delta Execution is implemented at the middle-ware level. The complexity is manageable because the execution states of a running application are readily available from the virtual machine. In addition, the UNIX operating system can encapsulate any platform specific issues and provides a standard POSIX execution environment. Consequently, the implementation can be carried out entirely in the high-level C/C++ language, making it very portable.

2.9.2 Performance

This issue deals with the overhead on carrying out thread migration. Performance is measured by the initial and run time costs introduced by the act of migration. A major part of the overhead is the cost to extract and transfer the execution states of the migrating thread [12]. To reduce this initial overhead, strategies such as pre-copy [26] and on-demand [12] are proposed, whereby instead of moving all the states at once at the migration point, the copying of states will be scattered throughout the execution. However, this kind of strategies will increase the runtime overhead.

In JESSICA, the initial overhead for copying states is reduced to a minimal with the help of the DSM system. Objects that are referenced by the migrating thread are located in the Global Object Space and need not be copied, and they are forwarded to the destination node transparently by the DSM system on demand. The largest data structure to be transferred when a thread migrates is the execution stack; and that the size of a typical execution stack is in the order of 100 bytes. Moreover, the execution states of a migrating thread are divided into a number of separate Delta sets that are only sent to the destination node when needed. In other words, JESSICA takes an on-demand approach to reduce migration latency and improves performance.

2.9.3 Transparency

Transparency requires both the migrated thread and the remaining threads in the system not to be aware of the occurrence of migration, all the threads have to interact the same way as if the migration has never taken place, except that some changes in performance may be perceived by them. Since the transparency requirement coincides with that of SSI, migration transparency follows automatically when SSI is supported. This is the case for JESSICA. The Global Thread Space hides the physical boundaries between nodes in the cluster, together with the message redirection mechanism and the Global Object Space, they allow threads to compute and interact with each other using the same set of operations independent of their physical locations. Consequently, migration transparency can be achieved.

2.9.4 Residue Dependency

This issue covers the amount of execution state information of a migrated thread that remains at the source after the thread has resumed its execution at the destination. Residue dependency is undesirable because it has a negative impact on the reliability and fault resilience of the system. On the other hand, residue dependency is sometimes unavoidable.

For instance, in order to support communication transparency, some state information of the migrated thread may have to remain at the source node for message redirection.

JESSICA takes the message redirection approach to support migration and SSI transparency. A helper thread (master) is left behind at the source node to perform any location-dependent operations on behalf of the migrated thread (slave). This master/slave design can simplify the implementation of redirection but at the same time some residue dependency is retained.

The residue dependency introduced by such migration mechanism has to be carefully dealt with, especially when a thread is migrated multiple times. This happens when a migrated thread needs to move again due to circumstances such as a surge in loading in the current node. In order to avoid introducing multiple forwarding points after successive migrations, in which a redirection has to go through multiple helper threads before reaching a thread's real destination, a migrated thread has to first retreated back to the source node if it is required to migrate for a second time.

2.9.5 Scalability

This issue examines if the thread migration mechanism and its performance are related to the scale of the underlying execution environment, such as the number of nodes in the cluster. The size of the cluster affects the organization and management of structures that maintain the residual execution states of migrated threads. If the size of migration related state information grows exponentially with the number of migrations performed, the system cannot be scalable.

JESSICA avoids the migration related states from growing too fast with the number of migrations performed by first moving the thread back to the source node before migrating to another destination. Apart from making the system more scalable, this design also helps to prevent the migrating thread from leaving its residue dependency all around the cluster as it moves from one node to another.

2.9.6 Heterogeneity

This issue considers the capability of migrating threads across machines with different hardware architectures. With the help of special compiler support, it is possible to insert checkpoints into the binaries for two different platforms of a given program. The positions of checkpoints are chosen so that there is a one-to-one mapping of execution states between the two platforms. At any one of the checkpoints, the program running on the source platform can be stopped and to have its states extracted, translated and transferred to the destination

platform where the program resumes its execution at a later time. Examples of systems following this technique are Emerald [10] and Tui [28].

As JESSICA takes on the virtual machine approach, the set of execution states of a thread is in fact part of that of the virtual machine. The set of states can therefore, in general, be expressed in a machine independent manner. It can be migrated to another computer of a different hardware architecture and resumes execution there. However, heterogeneous migration is not currently support because the DSM system that the JESSICA prototype is using does not support heterogeneity. As a result, a migrated thread cannot reference any object if the computer on which the migrated thread is executing and that the memory objects are residing are of different hardware architectures. This can be solved either by replacing the DSM system with a heterogeneous one or implementing an address translation service such as the one described in Java/DSM [39], hence heterogeneous migration in JESSICA is achievable.

2.9.7 Conclusions

Since some of the issues discussed above may be conflicting with one another, there are tradeoffs when deciding which alternatives should be taken. For example, to support communication transparency, it is necessary to retain some residue dependency at the source node for message redirection; to provide an implementation that can migrate quickly, the implementation may leave too much residue dependency at the source node and degrades the reliability and performance of the migrated thread [12]. The JESSICA implementation focuses on transparency and simplicity. It is willing to tradeoff and to tolerate a certain amount of residue dependency and decrease in performance.

2.10 Thread Migration Policy

Once the thread migration mechanism is implemented, the next thing is to decide the policy for thread migration. In other words, it is to determine the following:

- activation policy – *when* to trigger a migration
- location policy –*where* the thread should migrate to
- selection policy – *which* thread should migrate

These decisions should be made based on the load information of nodes running across the cluster. The load information should be able to represent the current degree of resource utilization at the nodes, such as system load index, memory utilization and I/O device

throughputs. Depending on the approach of how this information is managed, the information is disseminated to the nodes around the cluster in a certain manner. For example, if the system takes on a centralized approach, the load information of each node will be sent to a load information manager node for analysis. The load information manager collects all the information across the cluster and then performs the necessary analysis in order to make a load distribution decision. When the analysis is completed and the three aspects of migration policy mentioned above are decided, the system is ready to trigger migration operations once a load imbalance is detected.

As JESSICA focuses primarily on how to implement a transparent and effective thread migration mechanism, at the moment we only include a simple migration policy in the prototype for the sake of completeness. JESSICA adopts a centralized approach by which the load information of each node in the cluster is sent to a manager. The load information considered is the one-minute system load average returned by the `uptime` UNIX command. Its activation policy is simple: the system load average of computer nodes across the cluster are collected every minute and the average of these numbers evaluated thus giving the one-minute average of the overall system load in the cluster. For its location policy, if the manager finds a pair of nodes with system load 20% or more above the global average in one of them and 20% or more below the global average in the other, threads running on the higher-loaded node will be selected to migrate to the lower one. JESSICA's selection policy is again simple: the current thread that is executing will be selected and migrated to the lower-loaded node. Note that the policies implemented here and the numbers chosen are ad-hoc, further investigation is necessary to achieve reasonable result.

Apart from automatic load balancing by thread migration, JESSICA also supports remote execution. Once it is created, a thread can be migrated to an unoccupied node immediately before it starts execution. This facility is useful before a better migration policy can be devised.

2.11 Migration Granularity

The migration granularity in JESSICA is per bytecode instruction. A running thread in JESSICA can be stopped and migrated between the execution of successive bytecode. It is at these points where the execution states of the thread can be extracted from the virtual machine and represented in a machine independent manner. Otherwise suspending the virtual machine execution at any other points would mean a bytecode instruction being half-way executed, this would cause some internal states of the JVM implementation, which can be platform

dependent, to be introduced into the execution states of the running thread. This situation is analogous to stopping the execution of a CISC CPU before it can finish the current machine instruction, at this point the CPU should be in the midst of executing a set of micro-code instructions that implement the current machine instruction. To represent the state of the CPU at this point would introduce micro-code level states.

2.12 Comparison with Traditional Process Migration

The implementation of the thread migration mechanism in JESSICA is considered less complicated and demanding than that of the traditional process migration in UNIX-like system. First of all, Java thread is a first class citizen in the Java Programming Language, state information of a thread is well encapsulated in its instance object, and the unit of migration can be well represented. Secondly, as an application is executed by means of a virtual machine, the execution states of a thread instance can be readily obtainable from the virtual machine. Thirdly, the set of system services provided by the standard JVM is much reduced and simpler than that provided by UNIX. For example, inter-thread communication is supported by only the object semaphore and the wait-notify mechanism, while in UNIX there can be pipe, signals, semaphore, … et cetera. So the implementation of communication between migrated threads is much simplified. Fourthly, despite of the fact that the major software effort is now shifted to the implementation of SSI, transparent thread migration follows with little effort due to the high degree of SSI transparency supported by JESSICA.

Chapter 3

Related Works

3.1 Overview

This chapter provides a brief survey of works that are related to the JESSICA research. The majority of them focus on migration of computation load in a network of computers, either in the form of mobile object, thread, or process migrations. In general, their goal is to improve resource sharing in the network by migrating computations to idle machines. Some of them aim for improving reliability by migrating processes away from failed machines, others try to obtain speedup by migrating threads to multiple machines and execute them in parallel. One of the works is a distributed version of a commercial operating system that focuses on establishing a Single-System-Image (SSI) over a cluster.

The approaches that these research works have taken can be categorized into micro-kernel augmentation, monolithic kernel augmentation, encapsulation at the middle-ware level, programming language and software library support, and user level checkpoint-and-restart. The works selected here are the landmark cases in each of the categories.

3.2 Micro-kernel Augmentation

In this approach, the micro-kernel is extended to support load distribution by migrating tasks across the network. Task is the unit of program execution in contemporary micro-kernels like the Mach [7]. Task migration on the Mach micro-kernel is the leading example in this approach. The micro-kernel approach is believed to be the most convenient to support transparent migration because in a micro-kernel, only the basic abstractions are supported within, and other functionality is provided in user space.

3.2.1 Task Migration on Mach

The work of task migration on the Mach micro-kernel [7] demonstrates the flexibility and simplicity for implementing load migration at the micro-kernel level. Mach is a message passing based micro-kernel that succeeds the work of Accent at the Carnegie-Mellon University. The micro-kernel approach means that the Mach kernel only provides the smallest set of critical services such that other services can be built on top of them. Critical

services include task creation and scheduling, virtual memory management, and inter-process communication. Other services such as file storage and network communication are implemented at the user level in the form of user level servers. Task is a unit of active execution entity in Mach that corresponds to process in UNIX. It consists of an address space, some execution threads and a set of channel endpoints called ports for inter-process communication. A UNIX process is equivalent to a task in Mach with a single thread of execution.

Task migration in [7] takes advantage of the Mach NORMA version which supports in-kernel distributed inter-process communications (DIPC) and distributed shared memory (DSM), and the implementation can be much simplified. To migrate a task, it involves the extraction and transmission of the task's states across the network. These states include the task address space, the communication port capabilities, thread states and others. The DSM handles all the memory transfer and the DIPC performs the necessary message forwarding, which are done transparently and automatically. As a result, the modification to the kernel only adds up to about 300 lines of C code, the rest are implemented as a user level migration server which accounts for another 600 lines of code. The migration server is responsible for migrating the task states such as thread states and capabilities. It is also used for studying different address space migration strategies such as pre-copying and copy-on-reference, using the NORMA DSM mechanism.

The work of task migration on Mach has verified that relying on DIPC and DSM support can be a great help to achieving migration transparency. In light of this, JESSICA also incorporates a DSM system for object sharing between migrated threads. This simplifies the implementation substantially because the memory migration is handled transparently by the DSM system. In addition, JESSICA also implements its migration facilities in the user level above the UNIX kernel. However, the states of a migrating thread in JESSICA are comprehensively encapsulated within the virtual machine. They are readily extractable from the virtual machine in the source and are migrated to the destination. There is no need to modify the underlying kernel as in the case for Mach. On the other hand, JESSICA relies on the communication libraries such as the BSD Socket provided by the underlying UNIX operating system to support network communications. It implements its own message redirection functionality for supporting location transparent communication for migrated threads, which is more flexible than in the case for Mach.

3.3 Monolithic Kernel Augmentation

In this approach, the monolithic kernel of the traditional UNIX operating system is extended to support SSI over a cluster and/or to support process migration. Since the kernel possesses the best knowledge of the system resources available in each node of the cluster, this approach should be able to achieve the optimum SSI transparency as well as process migration transparency. On the other hand, working at this level is proven to be the most difficult and costly because modifying a monolithic UNIX kernel to support transparency would require substantial software effort as UNIX was not designed with the concept of SSI or process migration in mind. The leading examples in this approach are the NOW MOSIX [2] of the Hebrew University and the Solaris MC [42] of the SUN Microsystems.

3.3.1 NOW MOSIX

NOW MOSIX is one of the landmark systems that implements transparent process migration in a monolithic kernel and achieve good load balancing performance. MOSIX is a UNIX compatible operating system developed at the Hebrew University of Israel. It was originally developed as an SSI operating system for clusters of computers where the whole cluster appeared to user processes as one single system. However, the implementation became too complicated and too difficult to manage as the system evolved. It is redesigned later for a Network of Workstations (NOW) configuration known as NOW MOSIX using the "home" model whereby user processes are created to run seemingly at the user's "home" workstation. After switching to the "home" model, the implementation becomes relatively simple because it is confined to the development of the process migration and load balancing mechanisms. For the rest of the system services NOW MOSIX relies on existing mechanisms provided by UNIX, such as NFS, without modifying the internal structure of the UNIX kernel.

Migration granularity in NOW MOSIX is in the unit of a UNIX process. Migration transparency is achieved by separating a migrating process into two parts – the *body* and the *deputy*. The *body* contains all the system independent states and may be some system dependent states of the migrating process whereas the *deputy* contains only the system dependent states. The states contained in the *body* are location independent and hence the *body* can migrate to any machine on the cluster, while the *deputy* contains location dependent states and is not migratable, hence it has to stay at the "home" node of the process and to acts as a proxy for the *body*. When the *body* tries to execute a system call that is location independent, it will do so directly on the local kernel. Otherwise the system call is redirected back to the "home" node and it is the *deputy* that executes the system call on the *body*'s behalf, the result is eventually forwarded back to the *body*. At the same time, the *deputy* is also responsible for looking out for any asynchronous events such as UNIX signals sent from

the home kernel and forward them to the *body*. It is the cooperation between the *deputy* and the *body* of a migrating process that provides the necessary location transparency in MOSIX.

Thread migration in JESSICA also follows a similar approach as that in NOW MOSIX. In JESSICA, the system will extract all its state information from the virtual machine when a thread migrates. The thread states may contain machine dependent states as well as machine independent states. The dependent states can be a result of calling a Java method that is implemented in the hardware native code. A *slave* of this migrating thread is then created at the destination node. Only the machine independent states that correspond to bytecode methods are transmitted to this *slave* thread. The original thread at the source node now becomes the *master* and is responsible for completing the execution of any native method calls whose states are not migratable. In addition, the *master-slave* pair is also responsible for forwarding any location dependent system calls and asynchronous events between them similar to NOW MOSIX for maintaining network and location transparencies. Hence, the cooperation between the master-slave pair of a migrating thread in JESSICA supports heterogeneous migration of threads in addition to providing migration transparency. We call this technique "Delta Execution" and a more detailed discussion is provided in chapter 5.

3.3.2 Solaris MC

Solaris MC is a more recent work that implements SSI at the kernel layer. Solaris MC is a prototype distributed operating system for running over a cluster of computers and is developed by the Sun Microsystems Laboratories [42]. It extends the existing Solaris operating system to provide a single system image (SSI), a cluster that appears to users and applications as a single computer running the Solaris operating system. Modifications to the Solaris kernel are kept to a minimum, most of the extended components appear as loadable kernel modules to the Solaris kernel. The implementation and extension to the Solaris kernel are done using the high-level C++ Programming Language and the CORBA object model. Since object code compatibility and the kernel API are maintained, existing applications and device drivers are runnable and require no modifications. There are four major components in Solaris MC that make up the SSI illusion. The first is a global file system called the Proxy File System (PXFS). It is built on top of the existing Solaris vnode interface. The second component is the global process layer that provides global process management to the operating system and the physical locations of processes are transparent to users. The global process layer supports remote execution of process and redirects operating system signals to the node where the process physically resides. The third component is a global I/O device space where it is possible to transparently access any devices from any nodes regardless of where the devices are physically attached. The last component is a global network subsystem

which provides an illusion such that the set of real network interfaces available in the cluster appears to be local to each other. Transparency is achieved by implementing a packet filter on each node to forward data packets to and from the network interfaces.

In the global process layer, there are the virtual process (vproc) objects which represent processes running on the nodes of the cluster. On each node, there is a node manager that handles the mapping between the vproc objects and the local process running on the node. The vproc objects maintain information about the parent-child relationships between processes, they communicate with vproc objects running on other nodes through the CORBA IDL interfaces. Process-related system calls are also redirected in this layer. The vproc objects are useful for implementing remote fork and process migration. The proposed scheme to maintain migration transparency is to leave behind a shadow vproc at the source node when a process migrates; this shadow vproc can help to forward any operations received to the vproc on the node where the migrated process resides.

The SSI boundary created by Solaris MC can be considered as a global process space that spans across the cluster. Processes living within this global process space can be uniquely identified and have their physical locations hidden. The global process space supports remote creation of processes and operating-system-related messages are transparently redirected to the node where the processes reside. This global process space is analogous to the Global Thread Space in JESSICA that hides the physical boundaries between machines. The Global Thread Space also supports remote creation of threads and system services are also transparently redirected to the right location. On the other hand, threads living in the Global Thread Space can freely migrate around the cluster while those in the global process space in Solaris MC cannot as yet.

The Solaris MC idea of leaving a shadow virtual process behind when a process migrates is similar to JESSICA which leaves a helper thread behind in the source node, they both help to provide migration transparency by redirecting location dependent operations and messages. Other Solaris MC ideas can also be enlightening to the work on JESSICA; for example, the ideas of a global network subsystem and using a packet filter for network data redirection can be incorporated into java.net class library even though they will be done at the user level. After all, Solaris MC stands out to be a good reference for the implementation of SSI system such as JESSICA.

3.4 Encapsulation at the Middle-ware Level

In this approach, SSI can be achieved by having a middle-ware running between the cluster of computers and the applications. The middle-ware encapsulates the distributed nature of the cluster hardware and provides a single virtual parallel machine for the applications to execute upon. A leading example that is very similar to JESSICA is the work of Java/DSM [39] at the Rice University, another example that also makes use of an DSM system is the Millipede system developed at the Technion Israel Institute of Technology [18].

3.4.1 Java/DSM

Java/DSM is a distributed Java virtual machine (DJVM) that runs on a cluster of heterogeneous computers. It provides an illusion to the Java applications that they are running on a single JVM with multiple processors attached. Compatibility is accomplished by maintaining the same system interface as the standard JVM, therefore standard Java applications are also runnable on Java/DSM without modification. Parallel execution of Java applications is possible by having multiple Java threads running on multiple nodes in the cluster.

Java/DSM provides a DSM for all the nodes running across the heterogeneous cluster. A major issue in providing a heterogeneous DSM is how to handle the hardware differences when moving data across machine boundaries. For instance, an integer on hardware platform A can be represented by a 32-bit word in Big-Endian byte-ordering, while on another hardware platform B the same integer can be represented by a 64-bit double-word in Small-Endian byte-ordering. Java/DSM proposes to handle the issue by performing data conversion. Since the Java Programming Language is strongly typed, type information of any Java objects can be embedded into the runtime of the DJVM. When accessing a Java object that is stored in a remote page, this type-information can be retrieved to check what kind of data conversion is necessary for the object. With this data conversion mechanism installed, the hardware differences of the underlying heterogeneous cluster can be hidden from the Java applications.

Both Java/DSM and JESSICA follow the same approach by implementing a distributed virtual machine at the middle-ware level. They utilize DSM systems to simplify the implementation. However, in Java/DSM load distribution is achieved by remote invocation of Java thread alone, while JESSICA supports also transparent thread migration. Furthermore, the current Java/DSM prototype mainly focuses on supporting DSM in a heterogeneous environment; other issues such as location transparency and signal forwarding are not addressed, therefore it is not an SSI system.

3.4.2 Millipede

The Millipede system is developed at the Technion Israel Institute of Technology. It 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 and employs the DSM technique to simplify the migration implementation. 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 (MGS) who is responsible for collecting load information and making migration decisions.

A major issue in supporting thread migration is how to ensure the memory references that are found inside the stack of a migrating thread to remain valid after a migration. In Millipede, the DSM system automatically guarantees that any references to global data that are pointing to the shared memory must be valid no matter how many times a thread has migrated. However, for local memory references that are pointing to the locations within the thread stack memory itself, they need to be dealt with carefully. Consider if the stack memory of a migrated thread starts at a location that is different from that before the migration, then all local memory references within the stack memory will be invalid. Millipede solved this problem by statically creating a fix number of worker threads in all application instances when they are instantiated. Worker threads are threads that are migratable because they do not possess any location dependent resource. During migration, instead of creating a new thread at the destination, the execution states of a migrating thread are copied to the corresponding worker thread at the destination that is residing at the same memory location. As a result, all memory references are now valid and no address translation is required.

In Millipede, for a worker thread that is running in an application instance, all its thread counterparts that are occupying the same memory slots 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, in order to facilitate the thread migration mechanism explained above. 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 requirement on its new location, and there is no address translation required.

3.5 Programming Language and Software Library Support

This approach requires application programs to be written using some special languages with additional features for mobility or calling a set of software application programming interfaces (API) to perform thread or process migration. We are going to discuss here two examples using this approach, the Telescript [19] created by the General Magic Inc and the Arachne user-level thread package [9] developed at the Purdue University.

3.5.1 Telescript

Telescript is an object-based environment developed at General Magic Inc. It is designed for intelligent agent computing where a high degree of mobility is supported. There are four main components in the Telescript technology. The first is the Telescript programming language which is an object-oriented language similar to Postscript. This Telescript language is designed to support dynamic and autonomous agents whose nature is to deliberately move around a network and get a task done, such as migrating to a database server and make an onsite query. The second component is a Telescript engine called Magic Cap that interprets and executes programs written in the Telescript language. This Magic Cap virtual machine can run on small consumer electronic devices such as personal digital assistance (PDA) as well as workstations and large application servers. They create a heterogeneous environment for distributed computing. These messages are Telescripts programs themselves that can contain data as well as active agents for execution. The third component is the Telescript communication protocol which governs the encoding and decoding of Telescript agents when sending them between Magic Cap engines. The forth component is the set of software tools that supports the development of Telescript applications.

Two key elements in Telescript are agents and places. Agents are autonomous entities that can move around and perform communication and computation tasks. Places are virtual locations for holding agents; they are bound to Magic Cap engines and do not move. Agents can come together within a place to meet and interact with one another or with the place itself. Agents living in different places can also communicate by sending messages between them. A Magic Cap engine can support concurrent execution of multiple agents and places.

Both Telescript and JESSICA support moving objects under active execution from one place to another. However, the application programmer is responsible for dealing with the movement of agents in Telescript – to decide when and where to navigate; while thread migration in JESSICA is deliberately hidden from the knowledge of programmer. On the other hand, since the nature of the two systems are very different, one being designed for intelligent agent programming while the other is for supporting a more general purpose

programming paradigm, direct comparison between them may not be appropriate. General Magic has recently demonstrated this intelligent agent programming paradigm can be implemented using the Java Programming Language. Odyssey is a set of Java class libraries that allows the creation of agents and places in Java programs; it supports the same level of agent mobility as in Telescript [15].

3.5.2 Arachne

Arachne is a portable user-level programming library that supports thread migration over a heterogeneous network of computers. 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. Moreover, Arachne does not support asynchronous interrupt and provides only a cooperative thread scheduling mechanism, a thread will only be scheduled out if it relinquish itself or if it has reached some suspension point by calling some Arachne primitives, such as `<a_create()>` to create a thread. This is a problem because when a migrating thread reaches its destination node, the scheduler will not be aware of its arrival if there is another active thread running. The migrated thread will not be scheduled to execute until the current active thread has terminated or reached some suspension point, even if this migrated thread has a higher priority than the active thread.

In order to support heterogeneous migration, Arachne avoids using the native runtime stack provided by the underlying operating system as the execution stack for performing function calls in an application. Instead, for each function defined, it implements an artificial stack that is allocated from the heap to store all the local variables of this function. A preprocessor is used before compiling the application to modify the application code so that accesses to any local variables are now changed to access the variables defined in this artificial stack. As a result, the content in the execution stack of a function at the point of migration is readily obtainable from the artificial stack. Furthermore, since the data-types of variables defined in the artificial stack are known at the compile time, their values can be translated to formats that conform to different hardware architectures at migration. Moreover, in order to allow a thread to resume correctly from the point where it was suspended, Arachne inserts an artificial program counter and address label at every prospective suspend point in the application code. When a thread is resumed, it will try to check the value stored in the artificial program

counter and jump to the address location where it was last suspended. A shortcoming of this design of artificial stack and artificial program counter is the extra overhead it introduces during the execution of a function, even if migration has never taken place. Experiment shows the extra overhead due to this design can account for 61% of the execution time.

Another problem with the Arachne thread package is that it does not allow sharing of any data between threads if they are running on different nodes. At the point of migration, any functions called by the migrating thread that have not returned yet cannot contain any memory reference to the local heap as the system does not support remote memory access or data forwarding. All the local variables of these functions can only be of simple data types. Besides, object migration is also not supported; that is, there cannot be a local variable, whose type is non-simple, such as a C data structure or a C++ class. It is because the content in such a non-simple variable cannot be properly marshaled and migrated, unless a data marshaling and demarshaling routine is defined by the programmer for each non-simple variable concerned.

When comparing Arachne with JESSICA, JESSICA provides a more flexible, portable, efficient, and useable thread package for application programmers. First of all, migration is entirely transparent to programmers in JESSICA. There is no migration primitive that programmers have to insert into their code; therefore existing applications can run on JESSICA without modification and benefit from thread migration. 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, may it be simple or complex. Thirdly, JESSICA supports asynchronous interrupt and a migrated thread can be scheduled to run immediately if it has the highest priority among other threads in the destination node, there will be no starving for a migrated thread. Fourthly, JESSICA is more portable as it does not introduce any new keyword to the Java Programming Language.

3.6 User-Level Checkpoint-and-Restart

This approach relies on a stable storage where the states of a running process are extracted and saved in. The states are restored on another machine or on the same machine at a later time. The approach safeguard any failure of the original machine and allows for load balancing when the process is restored on a machine with less loading. Here we are going to discuss the Condor distributed processing system [25] developed by the University of Wisconsin-Madison, which is a classic example using this approach; and a more recent Tui

System created by the University of British Columbia whose aim is to support heterogeneous process migration [28].

3.6.1 Condor

Condor is a user-level batch processing system running on top of UNIX and supports load sharing over a cluster of computer. It allows the CPU cycles of otherwise idle computers to be used in a pool, by having a central load manager to monitor the utilization of computers across the cluster. A user can submit a job through the load manager that will locate an idle computer and start the job. When Condor detects there is an active user reclaiming the computer, it will first create a checkpoint file of the running job before killing it. The checkpointed job is placed in a queue to be restarted later when Condor spots another idle machine in the cluster. The checkpoint file contains all the necessary states of the running process so that it can be resumed correctly at the point when it was checked. The process is effectively migrated from one machine to another and the migration provides a kind of load balancing service over the cluster.

The Condor checkpoint-and-restart mechanism is entirely implemented at the user level which requires no modifications to the UNIX kernel or the application code, though an application has to link itself to the Condor checkpointing library in order to utilize the Condor facility. Correct implementation of migration depends on whether the states of a checkpointing process can be accurately captured. The states include the address space, open files, pending signals and CPU states. One of the less trivial tasks is to capture the states of any open files, without modifying the application code. This is accomplished by providing Condor's own version of `open()` and `close()` system calls which are then linked to the application code. The Condor implementation will in turn invoke the actual system calls by using the `syscall()` interface. After linking the application with Condor's version of augmented system calls, any files that are opened or closed by the application can now be recorded in the Condor runtime, so that they can be reopened at the restoring node afterwards, if necessary. In addition, migration transparency is maintained by redirecting location dependent system calls, such as file I/O, back to the home machine. A home machine is one where the application is originally started, and Condor will maintain a shadow process there to handle any system calls request that may be redirected from the migrated process sometime in the future. The system calls are transparently redirected without the application's knowledge by using the method of system call augmentation described above.

Library linking in Java follows some kind of delay-binding. The binary code of a class is not loaded and linked into the system until the time when the class is first referenced. In other

words, in JESSICA there is no need to re-link the augmented system classes to the applications because they will be so linked automatically by the system when the applications invoke them. On the other hand, although Condor and JESSICA follow two entirely different approaches to handle load migration, their migration transparency mechanisms do share some similarities. For instance, both systems will leave a shadow process/master thread at the home node to handle any location dependent operations.

3.6.2 Tui

Tui is a system that supports heterogeneous process migration by employing the checkpointing technique, which is developed at the University of British Columbia. It supports process migration over four different types of hardware architecture; namely, m68000, SPARC, i486 and PowerPC. A major issue in providing heterogeneous migration is the requirement of translating the entire state of the migrating process so that it can be understood by different machine architectures. This requires knowledge on the type of all data values that are used by the process as they are being stored in the global variables, stack frames, and the dynamic heap. Tui solved the problem by using special compiler support.

To migrate a process, a modified compiler is first used to compile the source program and to generate binaries for the various hardware architectures. The compiler will insert preemption points into the binaries at various locations where the execution states can be expressed equivalently in all the four architectures. In this way when migration is triggered, checkpointing is carried out at the next preemption point reached. Furthermore, the compiler will also insert stubs at function entry points that describe the data type information of the input parameters and the return value in the function calling stacks. At migration time, an external program called “migrout” will analyze the executable binary of the migrating process and extract the type information of all the data used in the process. With the help of these type information, “migrout” checkpoints the migrating process at the next preemption point and creates an intermediate checkpoint file that encodes the data values and other states of the process in a machine independent format. This machine independent checkpoint file is then shipped to the target machine where a new process is created and its execution state is then initialized using the information from the checkpoint file.

Providing heterogeneous migration in JESSICA is simpler than in Tui because Java programs are interpreted. The execution states are readily extractable and represented in a hardware independent format that follows the standard JVM specification [36]. In addition, the strong-typed nature of the Java Programming Language can also ease the task of data conversion when migrating a Java thread: type information of data storing in the stack is known because

it is operated upon using strong-typed instructions such as `pushi`, `fadd`, and `aload`. As in JESSICA, entries stored in the execution stack of a thread are either simple data type such as integer or object references, they are tagged with type information as they are pushed, popped or modified. Finally, both JESSICA and Tui require migration that only takes place at certain points where the execution states can be represented unambiguously across machines. These points have to be close enough so that migration can take place as soon as possible. In Tui they are the preemption points whose locations are determined by the special compiler, they are usually inserted at the beginning of a loop and at the end of each compound statement. In JESSICA, the preemption point is chosen naturally at the end of each bytecode instruction where the execution states can be represented in a machine-independent way.

3.7 Summary

Finally, we summarize the chapter by providing below a table of comparison between JESSICA and the characteristics of the related works discussed.

	JESSICA	Mach	NOW MOSIX	Solaris MC	Java/DSM	Millipede	Telescript	Arachne	Condor	Tui
Level of Approach	Middle-ware	Micro-kernel	Monolithic kernel	Monolithic kernel	Middle-ware	Middle-ware	Programming language support	Application programming interface (API)	User-level checkpoint-ing	User-level checkpoint-ing
Method of Load Distribution	Thread migration	Task migration	Process migration	Remote execution of process	Remote execution of thread	Thread migration	Migration of mobile object	Thread migration	Process migration	Process migration
Implementation Techniques	DSM + Message redirection by helper threads	DSM + DIPC	Message redirection by shadow process	Message redirection by vproc + CORBA RPC + packet filtering	DSM + data translation	DSM + Migration Server	Execution engine (Static Place) + Program scripts & Data (Mobile Agent)	Provide a set of migration related routines to be included into applications	Provide customized system call library to be relinked with applications	Checkpoints inserted by special compiler support
Characteristics	Execution by means of a virtual machine	Demonstrated that micro-kernel approach is suitable for implementing transparent task migration	Process migration following the home model	Commercial OS with contemporary software techniques	Execution by means of a virtual machine, heterogen-eous	A pool of worker threads is reserved on each node	Execution by means of a virtual machine, supports heterogeneous migration	Heterogeneous migration, no data sharing between migrated threads	Use shadow process to provide message redirection	Heterogen-eous migration
Support Migration or SSI Transparency	Full SSI & migration transpar-ency	Full migration transparency	Full migration transpar-ency	Full SSI transparency	Not supported	Limited migration transpar-ency	Not supported	Not supported	Fair migration transpar-ency	Not supported
Support SSI	Yes	No	No	Yes	No	No	No	No	No	No

Table 3.1: Comparison of characteristics between JESSICA and the related works discussed

Chapter 4

System Architecture and Implementation

4.1 Overview

JESSICA is designed as a group of daemons running on all the nodes in a cluster of computers. They execute as user level processes on top of the UNIX operating system and it is the collaboration and coordination between these JESSICA daemons that offer an SSI illusion to Java applications. Together they act as a distributed system that integrate the resources such as system memory and CPU cycles which are distributed around the cluster and offer them as system services from a single virtual computer. A Java application can be started on any node and it will appear to be running on a JVM that has multiple processors. In other words, the SSI illusion offered to Java applications is a JVM that can execute multi-thread applications in parallel.

4.2 The Console Node

Java applications can be started on any node in the JESSICA cluster. They are running on the distributed virtual machine that follow a *home model*, which is similar to that in Sprite [12] and NOW MOSIX [2]. 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 that are located at other nodes are all become transparently accessible by the application, as if they were located at the console node. The access of remote processors is supported by the transparent thread migration mechanism where an active thread of the application can be moved and bind to a remote processor and execute there. The access of remote memory pages is supported by the DSM system where remote pages can be paged in and cached locally when necessary, the cache coherent protocols implemented in the DSM system maintains the consistency of shared memory. The access of remote files is supported by NFS installed in the underlying operating system (Fig 4.1).

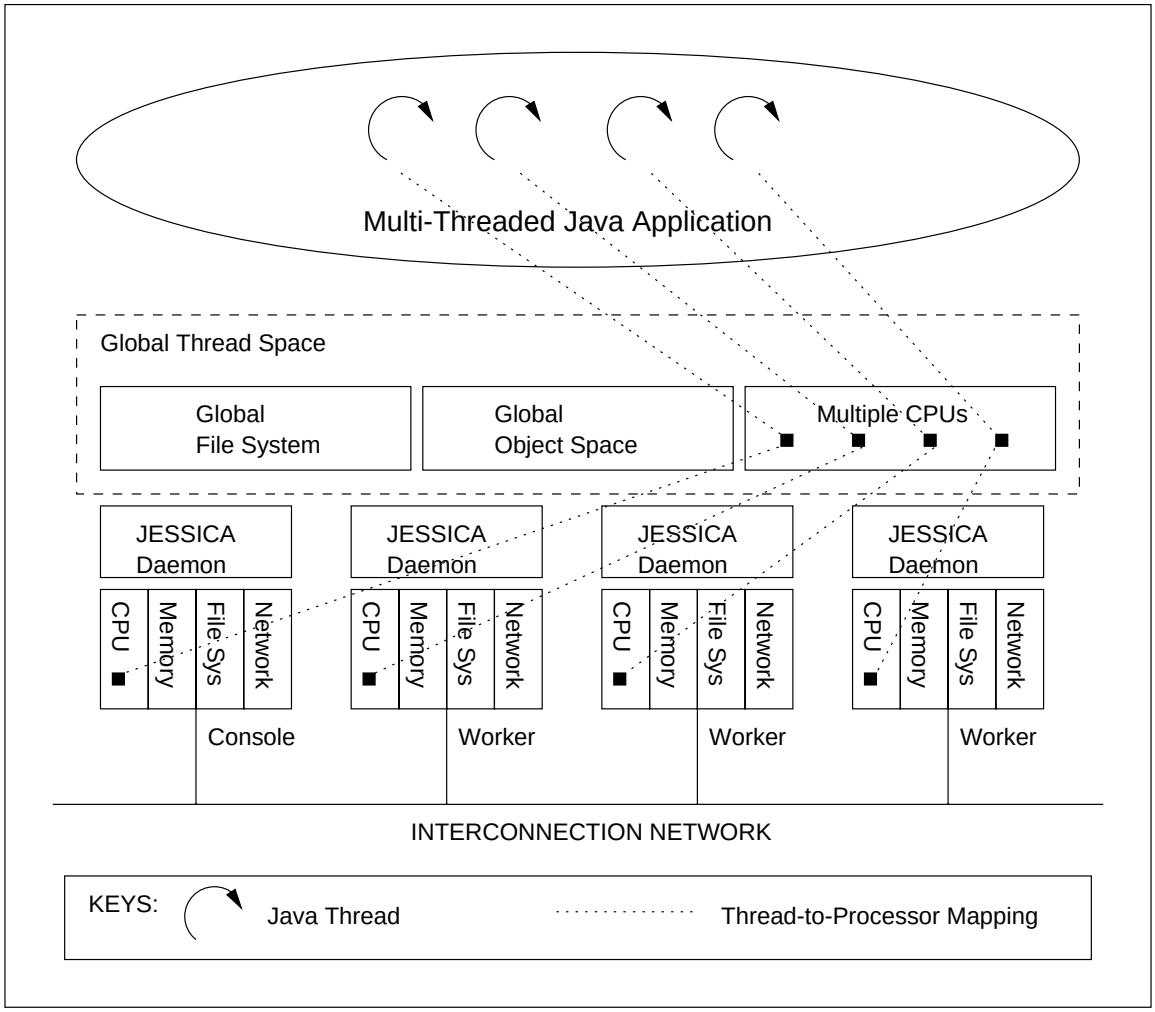

Fig 4.1: Entities in a JESSICA system

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 resources request made by the migrated thread will now be served by the local daemon running on the destination node. However, when the daemon discovers the requested service call is location-dependent, the call will be redirected back to the console node for it should be handled there. For instance, if a migrated thread wants to write a message to the application's standard output, the request will be forwarded back to the console node and the message will be displayed on the console's screen. In general, a thread can access devices and network interfaces that are attached to the console node only because they are location-dependent resources and service requests made by a migrated thread will be automatically directed back to the console. In other words, JESSICA currently does not support the concept of global device space and global networking as that in Solaris MC [42], where devices and network interfaces in the cluster can be accessed from any node as if they were attached locally. It is possible to implement similar feature in

JESSICA by providing a universal name space for devices and network interfaces that are attached to the cluster. JESSICA can maintain an internal table that maps the name of a device to its physical location, and a thread can then transparently access the device by redirecting the access request to the node where the device is attached.

Note that this console node is also responsible to enforce the migration policy by managing the load information of itself as well as that sent from other nodes in the cluster. Although this centralized approach may not be scalable with the number of nodes in the cluster, the primary goal of JESSICA is to provide a transparent migration mechanism and therefore the current scheme of load distribution is rather primitive, and the migration policy devised is simple. On the other hand, JESSICA also supports remote execution of Java threads, where a user can specify the initial degree of parallelism when a multi-threaded application is invoked. The specified number of threads will be migrated to other nodes in the cluster when they are first created. As a result, they will resume and execute from their first instruction on the remote processors, and the user can observe the effect of parallelism immediately.

4.3 The 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 node to execute the application and they now act as an extension to the console. They are the *worker nodes* with respect to the console, who 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. For example, when a worker node receive a migration request from the console, it will create a new thread instance to represent the migrating thread. The execution states of the migrating thread, which are marshaled into a machine independent format by the console, will then be shipped to the worker. The worker node will have to demarshal the state data, allocate a thread execution stack and other necessary system resources locally, and to use the data to initiate the newly created thread to a state which is identical to the original at the console just before the migration is triggered. The thread then resumes execution at the worker node and it has now become the migrated thread.

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 without the knowledge of the thread. In addition, the worker node will also prepares itself to receive any asynchronous messages that may be sent from the console. These messages can be the result of threads that are running elsewhere who want to signal this migrated thread. The worker node has to forward these messages to the right migrated thread.

The worker node takes its part in the DSM system by forwarding its memory pages to the console or other worker nodes, if the other nodes try to access objects that are stored in its local memory pages. Furthermore, the worker node is also responsible to periodically send the system load information back to the console to facilitate the enforcement of the migration policy.

4.4 System Components in the JESSICA Daemon

Each JESSICA daemon is composed of four major components (Fig 4.2); namely the Bytecode Execution Engine (BEE), the Distributed Object Manager (DOM), the Thread Manager (TM) and the Migration Manager (MM). They provide the following system services the same way as a standard JVM to Java applications:

- Bytecode execution
- Memory management
- Thread creation and scheduling
- Thread synchronization and signaling

Furthermore, they are also responsible to coordinate with other daemons to create an SSI in the form of a Global Thread Space. This SSI illusion is supported by:

- a Global Object Space for distributed data sharing between threads
- a transparent thread migration mechanism for moving threads around to execute on different processors
- a message redirection mechanism for migrated threads to maintain the SSI transparency

This section discusses each of the four components in more details.

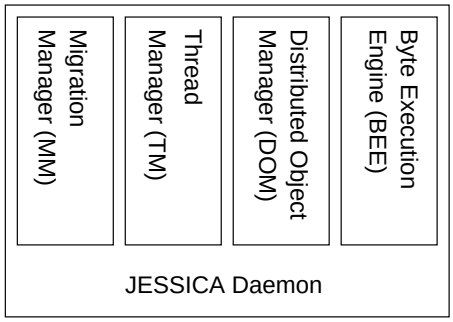

Fig 4.2: System components in a JESSICA Daemon

4.4.1 Bytecode Execution Engine (BEE)

The BEE is the heart for the execution of Java applications. It is designed to bind to an active thread and to execute its `start()/main()` method until the method's point of return is reached. The BEE is a stack-based engine that follows the JVM Specification [36] where BEE operates on the execution stack of the thread that is bound to. A *method call stack* is implemented in the BEE that allows the thread to perform repeated method invocations. Notice that the execution states of the BEE for an active thread is equivalent to the execution state of the thread itself. To facilitate thread migration the states of the BEE are designed to be readily extractable and be shipped to other worker node. Detailed discussion of how the states are extracted and migrated can be found in the Chapter 5, which is dedicated to the discussion of Delta Execution.

Since the states of the BEE need to be marshaled into a hardware independent format when they are shipped to another node during migration, the type information of values stored in the thread execution stack has to be known beforehand. This is achieved by implementing an auxiliary stack in conjunction with the thread execution stack. The auxiliary stack is operated on simultaneously with the execution stack, however only type information will be pushed and popped from the auxiliary stack. In this way the data type of a value stored in the thread execution stack can be found by looking up the corresponding slot in the auxiliary stack.

The task of the BEE is not solely to operate on the execution stack of a thread alone, it is also responsible to maintain the global class repository. When BEE encounters a bytecode that creates an object instance of a given class, it is responsible to check if the corresponding class is already loaded into the global class repository, and to load the class if it is not so. The class is loaded by executing a class loader method of the system. Once the class is completely loaded and verified, the BEE will then request the DOM to allocate space from the Global

Object Space for the new object. Finally BEE executes the class constructor to instantiate the new object instance.

4.4.2 Distributed Object Manager (DOM)

The DOM in a JESSICA daemon is responsible to manage the local memory resource and to cooperate with DOMs running on other nodes to create a Global Object Space, where object in the global space can be accessed independent of their locations. The DOM is implemented on top of a DSM system. The Global Object Space is collectively managed by all the participating DOMs where each DOM is responsible to manage its own share of local memory resource. A DOM will forward any of its memory pages to other DOMs for caching to support remote memory accesses made by threads running on other nodes. It employs cache coherent protocol to maintain the consistency of the shared memory. The global class repository is also part of the Global Object Space. Since it is possible for multiple BEEs on different nodes to upload classes into the repository simultaneously, semaphore is used to avoid any race condition and to maintain consistency. The design of the global class repository is to allow a single instance of each class to be shared by all the nodes in the cluster.

Object allocation requests made by threads are always satisfied locally. When a migrated thread on a worker node issues an object allocation request, the DOM will allocate the space from its local memory instead of forwarding the request back to the console. This design is justified by the principle of locality where the migrated thread on the worker node is likely to access the object again and again in the near future. It should be more effective to allocate from the local memory of the same node where the thread resides.

Each DOM manages the local memory resource by dividing the memory space into a number of big memory trunks of various sizes; their sizes are multiples of the size of a memory page. A fix amount of space is reserved at the beginning of each memory trunk to facilitate accounting and garbage collection, the remaining space is divided into many equally sized free blocks. These free blocks from all the trunks are then chained together to form a number of free memory lists for new object allocation. The block sizes in all the trunks range from a few bytes to sizes that are of multiple page large, so that the free lists formed contains blocks of many different sizes which are comprehensive enough to satisfy different object allocation requests. When a new object is allocated, the DOM will locate the memory trunk where the object is belonged to, and to record the type information of the object in a corresponding slot in the header of the trunk. The type information is recorded by saving the memory reference that points to the class of the object in the global class repository. In this way the class type

can always be determined given any objects. The header of a trunk is also useful for garbage collection, where it can provide a flag to indicate whether an object has already been visited or not during the mark phase.

When a DOM tries to perform garbage collection, it will obtain the execution stacks of all the active threads from the TM, so that it can start tracing recursively there and look for any objects that are reachable from those threads. In the simplest case when no migration has ever taken place, the remaining objects that are not reachable after this mark phase are inactive and so can be safely freed. On the other hand, if there is migration, the unreachable objects may still be referenced directly or indirectly by threads running on other nodes. Therefore a distributed garbage collection algorithm need to be deployed in order to prevent freeing any active objects inadvertently. Further discussion on our proposed scheme of distributed garbage collection is given in chapter 6.

4.4.3 Thread Manager (TM)

The TM is responsible to create, destroy and schedule active Java threads running on the local node. In the Global Thread Space the physical boundaries between all the nodes in the cluster is gone, threads running on different nodes can see each other as if they were running on the console node. The mapping between threads to node-processors is dynamic and transparent to the threads themselves. Consequently, thread scheduling in the cluster is performed in a distributed fashion. Each TM is responsible to schedule threads that are running on its local node, and at the same time, threads running on different nodes can be synchronized and signal each other by sending messages between the TMs. The implementation of distributed synchronization and remote signaling will be discussed in chapter 6.

Thread scheduling in the TM is implemented using a set of ready queues with multiple priorities. Note that as the case in the standard JVM, the scheduling mechanism implemented is not on a time-slicing basis, where threads of the same priority are given equal time slice to execute in a round-robin fashion. Instead, when there are more than one ready thread having the same highest priority, only some of them, a number that is equal to the number of the available processors will be scheduled. The rest of the ready thread will be kept waiting in the ready queue until one of the running thread deliberately relinquish itself, or is blocked by a semaphore, or is trying to read from/write to a file descriptor that is not ready. To implement this blocked on semaphore and blocked on I/O mechanism, there is one block queue for each semaphore and opened file descriptor. When, for example, the BEE that is executing the current thread tries to acquire a semaphore, it will check if the current thread is already the owner of the semaphore. If it is not the BEE will notify the TM to de-schedule the

current thread and put it onto the semaphore's block queue. Later when the semaphore is released by some other thread and it comes to the thread's turn, the TM will place the thread back to the ready queue.

In the process of migration, it is the TM who is responsible to extract, marshal and transport the execution states of a migrating thread to the destination worker node. And the TM at the destination is also responsible to receive, demarshals and instantiates the migrated thread so that it can resume from the same point when it was frozen by the TM at the console. TM supports remote execution of threads by allowing a user to explicitly specify the degree of parallelism at the time when the application is invoked. As a result, whenever a new thread is created by the TM, it will migrate the thread to other worker nodes until the specified degree of parallelism is reached. On the other hand, migration can also be triggered internally by the MM. When MM discovers there is any load imbalance, it will notify TM which thread should be migrated and to where.

A thread can migrate multiple times: when the MM at the worker node of a migrated thread discovers there is load imbalance again, the migrated thread is possible to be selected again as the candidate for another migration. However, instead of directly migrating to another worker node, the TM will first migrate the thread back to console node, where the TM at the console will look for another worker node to migrate the thread to. This approach is taken because if the migrated thread is directly migrated to another worker node without moving back to the console, it will again leave behind residue dependency at the first worker node so as to maintain transparency. From then on messages forwarded from the console will have to first go through this worker node before they can reach the new home of the migrated thread, which result in one more level of redirection. If further migrations are made, there will be more redirection of messages and the migrating thread will leave behind too many residue dependencies on multiple nodes. The multiple forwarding points so introduced are not desirable, as the redirection overhead in this case will be proportional to the number of migrations. However, if the thread is first migrated back to the console before taking another migration, the residue dependency on the first worker node will be removed with the leaving thread. On the other hand, it is also possible to support direct migration to a second worker node, by notifying the console TM that the thread has moved to a new location. However, this would involve readjustment of all the communication links of the migrated threads between the console TM and the worker TM. The current implementation opts for a simpler approach by first migrating the thread back to the console. Another point to note is that the cost for moving the migrated thread back to the console node will be much smaller than that for the initial migration, since in the former case only the content in the execution stack will

be copied back to the console. Besides, in the latter case extra time is required to re-instantiate the migrated thread and to setup communication channels between the console TM and the worker TM.

4.4.4 Migration Manger (MM)

The MM is responsible to collect load information in the local node and to exchange the information with MMs running on other nodes so as to enforce the 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, and the TM will perform the migration operation on behalf of MM. A sophisticated scheme can take into account of various system parameters such as CPU load, memory access and I/O throughput. For example, in the case of Millipede [18], threads that are accessing a significant amount of shared data or making frequent communications are moved together to the same node to optimize the locality for memory access or to minimize communication overhead. This is achieved by having the system to monitor the amount of remote memory accesses made to the underlying distributed-shared memory system. On the other hand, because the primary goal of JESSICA is to implement an effective migration mechanism, we have only devised a simple migration policy. In JESSICA, the one-minute system load average obtained from the `uptime` UNIX command is used as the load information. In addition, JESSICA takes a centralized approach in managing the load information; the MM of the console node is responsible to manage all the load information across the cluster and make migration decision.

To perform load management in JESSICA, each MM obtains the one-minute system load average from its operating system every minute. The MMs on all the worker nodes will forward this average value back to the console node. The MM at the console node computes the average of all the system load values received and the result is an estimation of the overall one-minute system load average across the cluster. The console MM will then try to compare the system load average of each node with this global average. If it can find a pair of nodes whose system load average is 20% or more higher than the global average for one of them, and 20% or more lower for the other, it will trigger a migration. The console MM will notify its counterpart at the higher-loaded node to migrate a thread to the lower-loaded node. At the higher-loaded node, the MM will simply select the thread that is currently executing as the candidate for migration. Note that the current migration policy is an ad-hoc design. It is implemented just for the sake of completeness of the whole system, and further investigation is necessary to obtain reasonable result.

The following figure illustrates the interactions between the four system components in a JESSIA daemon.

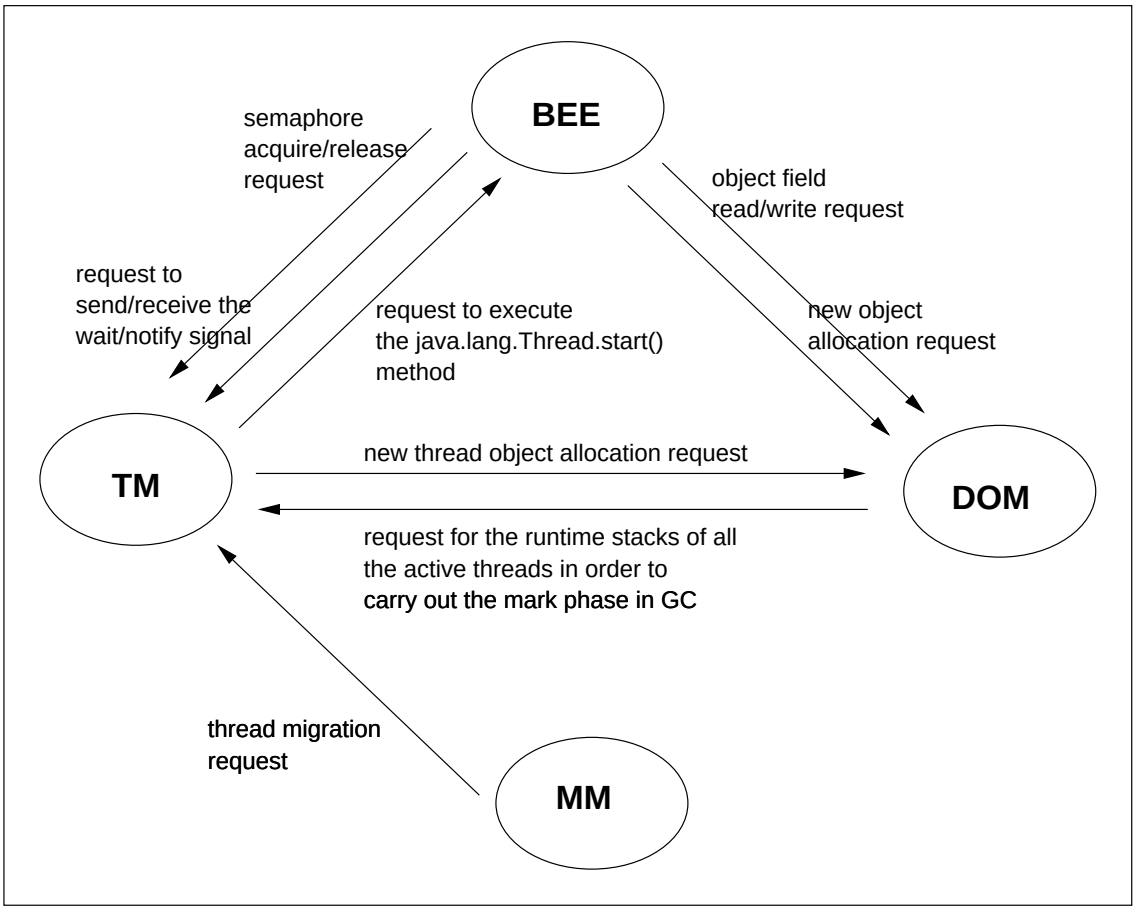

Fig 4.3: Interactions between system components in JESSICA

4.5 Implementation of JESSICA

The implementation of JESSICA is based on Tim Wilkinson's work of the KAFFE virtual machine [37]. KAFFE is an open source implementation of JVM that supports execution of Java applications in both the interpreter mode and the just-in-time compiler mode. We began our work with KAFFE implementation version 0.9.1 that is compliant to the SUN JDK 1.1.2. The KAFFE code provides the basis for implementing the bytecode execution, the thread system and the memory system in JESSICA.

We have made considerable modifications and extension to the KAFFE code in order to support the Global Thread Space in JESSICA. For example, we have replaced the original method calling mechanism with our own method calling stack and stack frames, so that the method calling sequence of a Java thread can be expressed in a machine independent manner. In addition, message-forwarding mechanism is added into the network and system classes in order to support the required SSI transparency in communication.

We employ the Treadmarks package [1] for the distributed-shared memory support in JESSICA. Treadmarks is a DSM system that runs on clusters of computers of various architectures, such as SUN SPARC and Intel i386. JESSICA has incorporated Treadmarks version 1.0.1 as its DSM layer, which does not yet support heterogeneity. We have augmented the signal handlers of the Treadmarks system to include our own processing for segmentation fault and I/O events. The current version of JEESICA runs on the SUN Solaris 2.6 operating system.

The implementation techniques presented in the following sections cover various areas of the JESSICA implementation. These areas are careful selected such that their implementation contain certain subtleties which are worth to mention. The purpose of including them here is to help people who would like to study the source code of JESSICA to understand the implementation better. While the discussions on Delta Execution and the implementation of SSI transparency shall be provided in detail in the two separate chapters that follow.

4.5.1 A Portable User-Level Thread System

The thread system in the original KAFFE source is implemented using assembly code. We have reworked the system to make it more portable by using the UNIX `setjmp()/longjmp()` system calls and implemented a thread system at the user level. Notice that when a user process first executes the `setjmp(jmpbuf)` call, the values of the CPU registers such as program counter (PC) and stack pointer (SP) are all saved in an array

pointed to by the `jmpbuf` variable. So that when the process later executes the `longjmp(jmpbuf, val)` call with the `jmpbuf` as its argument, the CPU registers and the state of the process will be restored back to the point when `setjmp()` is first executed, only that the value return by `setjmp()` this time is non-zero. Here, the idea is to overwrite the SP value in the `jmpbuf` array such that when the process executes the `longjmp()` call the second time, the process is redirected to execute within the execution stack of the user thread, and from this point onwards, the process can be set to execute the corresponding function that is bound to the thread. So the next step is to identify the position where the SP is stored in the `jmpbuf` array. This position is a machine dependent value where different architectures can save the SP at different positions. For the case of our implementation, the SP is located at the 2nd element within the `jmpbuf` array, i.e. `<((void **) jmpbuf)[1]>`. Therefore, to instantiate a user thread, we do the following:

```
00  void threadInit(thread t) {
01    If (setjmp(t->jmpbuf) == 0) {
02       memcpy(t->stackBase,((void **)t->jmpbuf)[1], t->stackCopy);
03       ((void **) t->jmpbuf)[1] = t->stackBase + t->stackCopy;
04    } else {
05      t->function();
06    }
07  }
```

At line 01, the thread's jmpbuf is initialized to point to the array that stores the CPU registers, and when `setjmp()` is invoked the first time, control flow will goes to line 02, where the stack content of the process is copied to the thread's stack space. At line 03, the SP value in the `jmpbuf` is overwritten with the top of the execution stack of the thread. As a result, when the process later execute `longjmp(t->jmpbuf, val)` again with `jmpbuf` as its argument, the control flow will return to line 04 as if `setjmp()` has returned with a non-zero value. Now if the thread proceed to call its own `function()` at line 06, the thread will execute within its thread stack.

To complete the discussion, here we introduce how to schedule out the current thread and let another thread to gain the CPU control.

```
01  void threadSwitch(thread to, thread from) {
02    if (setjmp(from->jmpbuf) == 0) {
03          longjmp(to->jmpbuf, 1);
04    }
05  }
```

Now when the current thread (`from`) wants to let another thread (`to`) to resume its execution, at line 02 the current thread will first save the current CPU registers in its own `from->jmpbuf` with the `setjmp()` call, and then jump to the execution context of the `to` thread by the `longjmp(to->jmpbuf, 1)` call at line 03, using the `to->jmpbuf` that was previously initialized by a `setjmp()` call made by the `to` thread itself.

4.5.2 Installing Custom Signal Handlers

In JESSICA, the BEE is required to catch any SIGSEGV signal that may be generated by faulty Java applications and to throw the null pointer exception as a result. In addition, it is required to catch the SIGIO signal to handle asynchronous I/O. However, the Treadmarks DSM package that we are using has already installed its own handlers for the SIGSEGV and the SIGIO signals: to detect any page faults due to remote memory access and to update its remote memory cache when the remote pages arrives. If we simply installed our own handlers with the `signal()` system call, then the original handlers installed by the DSM package will be overwritten and the DSM system cannot function correctly. On the other hand, we cannot modify the DSM signal handlers and include our processing code directly because source code for the DSM system implementation is not available. We have solved the problem by first extracting the DSM signal handlers immediately after the system has started up, and then register our own custom signal handlers that will invoke the DSM signal handlers eventually. The following code illustrates how it is done for the SIGSEGV signal.

```
01  void(*)(int) original_sigsegv_handler = NULL;
02  void my_sigsegv_handler(int signal) {
03    // my own processing here
04    // ...
05    (*original_sigsegv_handler)(signal);
06  }
07
08  void main() {
09    //...
10    struct sigaction act;
11    // replace SIGSEGV handler
12    sigaction(SIGSEGV, NULL, &act);
13    old_sigsegv_handler = act.sa_sigaction;
14    act.sa_handler = (void(*)(int)) new_sigsegv_handler;
15    sigaction(SIGSEGV, &act, NULL);
16    // ...
17    // ...
```

When the application has been started, inside the `main()` block, the original signal handler for SIGSEGV is first extracted using the `sigaction()` system call at line 12. The original handler is then saved in the `orginal_sigsegv_handler` variable so that it can be later invoked by the new signal handler at line 05. Afterwards, the custom signal handler is

registered with the system at line 15. From this point onwards, if the process generates any SIGSEGV signal, it is the `my_sigsegv_handler` that will be invoked. And before this handler return, the original handler will also be invoked at line 05.

4.5.3 Detecting Migration Event

The migration granularity in JESSICA is per bytecode instruction. That is, when the MM issues a migration request to the TM, the migration process can be triggered once the current bytecode instruction has completed its execution. The bytecode execution is implemented in the BEE as a `switch` statement within a `for` loop to handle different bytecode instructions as follows:

```
01  for (;;) {
02    pc = npc;
03    npc = pc + instr_length[code[pc]];
04    switch (code[pc]) {
05    case NOP:
06    break;
07    case ILOAD_0:
08    // load integer zero onto the stack ...
09    // ...
10    // ...
11    } // end switch
12  } // end for
```

Our first approach is to insert the necessary checking as an `if` statement at the top of the `for` loop at line 04 to see if migration has been triggered after finishing the last instruction. However, we discovered this would mean the checking for migration will be executed together with *every* bytecode instruction. This scheme will incur too much overhead and slow down the execution tremendously. Eventually, we solved the problem by adding a pseudo-instruction MIGRATE and deliberately change value in the next-program-counter (`npc`) variable and the `code` variable so that the next instruction is to execute MIGRATE. The following code illustrates how.

```
00    int pc, npc;
01    ByteCode[] code;
02    ByteCode migration_code[1] = { MIGRATE };
03
04    // to be called by migration manager MM
05    void triggerMigration() {
06          npc = 0;
07          code = migration_code;
08          // ...
09          // ...
10    }
11    // ...
12    // ...
13    // inside the BEE implementation
14    for (;;) {
15          pc = npc;
16          npc = pc + instr_length[code[pc]];
17          switch (code[pc]) {
18          case NOP:
19          break;
20          // ...
21          // ...
22          // ...
23          case MIGRATE:
24          TM->performMigration(currentThread);
25          break;
26          } // end switch
27    } // end for
```

When the MM would like to trigger a migration, it calls the `triggerMigration()` function which will change the next-program-counter (`npc`) and the `code` variable to points to the instruction MIGRATE (line 06, 07). Now within the BEE when the execution of the current bytecode has finished and the control goes back to line 15 to load the next instruction. The change made by the `triggerMigration()` function will caused the MIGRATE instruction to be loaded (line 17). And as a result, the TM will be invoked to perform migration at line 24. Notice that the `case` statement for the MIGRATE instruction is inserted at the end of the `switch` statement so that it will not introduce any extra comparison operation under normal situation.

4.5.4 Migrated Thread to Call Native Method

JESSICA allows a Java thread to call native methods that are implemented using another programming language such as C/C++. These native methods have to be first compiled and be prepared in the form of a dynamic linked library so that their binary image can be linked to the JESSICA daemon runtime address space when the methods are being invoked later. Within the implementation of BEE, when it discovered the method that the current thread is invoking is a native method (line 01 of the code example in the following page), it will check if the method has already been initialized (line 02). If not, the corresponding dynamic linked

library is loaded into the daemon process space (line 03) and the beginning address of the method is located its value is assigned (line 04). Finally the native method is invoked (line 06). The situation is illustrated using the following code.

```
00  // within the BEE, where it is about to invoke a method <meth>
01  if (method->isNative) { // the current method is a native method
02    if (method->nativeCode == 0) {
03      lib = LoadNativeLibrary(method->signature);
04      method->nativeCode = lookupMethod(lib, method->signature);
05    }
06    (*(method->nativeCode))(argc, argv);
07  } else { // it is a bytecode method ...
08    // ...
```

This implementation is fine if there is no migration. On the other hand, if migration has taken place, there will be problems when a migrated thread tries to execute a native method again which was previously executed at the console node before the migration. This is because in this case the `method->nativeCode` has already been initialized and its value is pointing to the function whose location is only meaningful within the JESSICA daemon at the console node. Notice that for a dynamic linked library, the location where it is loaded to a process space can be different every time it is loaded into the space, which means the same native function can be located at a different address inside the JESSICA daemon space at the worker node. In other words, the value pointed to by the `method->nativeCode` variable is the most likely meaningless at the worker node, and executing `method->nativeCode()` as a function call (line 06) can be catastrophic. The correct solution is to lookup the address of the native function in the worker daemon again before executing.

```
00  // within the BEE, where it is about to invoke a method <meth>
01  if (method->isNative) { // the current method is a native method
02    if (method->nativeCode == 0|| currentThread->hasMigrated) {
03      lib = LoadNativeLibrary(method->signature);
04      method->nativeCode = lookupMethod(lib, method->signature);
05    }
06    (*(method->nativeCode))(argc, argv);
07  } else { // it is a bytecode method ...
08    // ...
```

The above code shows that the `method->nativeCode` should be re-instantiated again if it has not been initialized or if the current thread has migrated to a worker node (line 02), before invoking the native function.

Chapter 5

Delta Execution

5.1 Overview

Preemptive thread migration can be an effective means to support load distribution in a cluster of computers by moving threads from highly loaded nodes to execute on other underload nodes. This chapter introduces Delta Execution, a preemptive thread migration mechanism that we have devised for JESSICA by using an analytical approach. The availability of a migration mechanism such as Delta Execution allows a cluster to enforce its load distribution policy so that the overall resource utilization in the cluster can be improved.

In the Delta Execution approach, we have formulated a structural expression for representing the current execution states of a running thread. The expression enables us to identify and separate the machine dependent execution states from the machine independent states of a running thread, where the machine independent states are well defined in the expression. Consequently, we are able to develop a high level thread migration mechanism so that only the machine independent states of a migrating thread are extracted and are moved to a destination node for execution. We have successfully avoid the manipulation of any machine dependent states by leaving them behind at the source node so that any execution that involves these machine dependent states will still be performed at the source.

The machine independent states of a migrating thread at the source node are encoded into multiple units of execution called *delta sets*, between the *delta sets* are the non-migratable machine dependent execution states. Active execution of the migrated thread will be observed as shuttling back and forth between the source and the destination node. A *delta set* is executed on the destination and then followed by the execution of non-migratable machine dependent instructions back at the source. This process is repeated until all *delta sets* are exhausted (Fig 5.1).

Because Delta Execution avoids the manipulation of any machine dependent state information, the implementation of the migration mechanism can be done without touching the low-level details of the operating system or the hardware. The result is a very portable implementation. In addition, since only the machine independent states are migrated, it is

possible to extend the mechanism for supporting heterogeneous migration, where a thread is migrated to a computer that is of different hardware architecture.

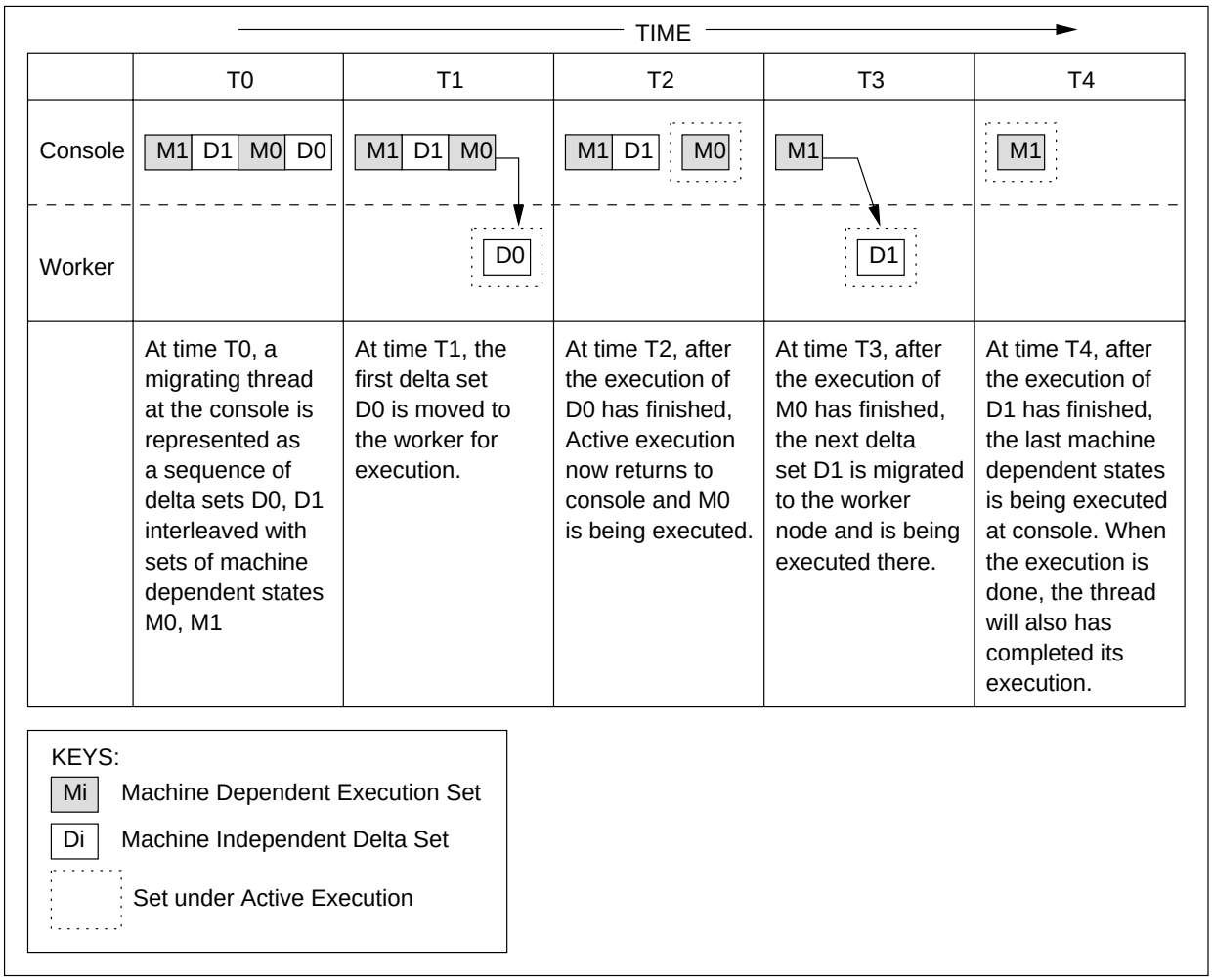

Fig 5.1: Delta Execution in action

In general, the Delta Execution approach can be applied to implement thread migration on any language systems that execute by means of a virtual machine. Because all the machine dependent operations are performed back at the console node, correctness of execution can be guaranteed. Furthermore, thread migration is transparent to application programmers and all existing programs will remain functional and be ready to execute.

5.2 Capturing the Execution Context of a Running Thread

The primary task for performing thread migration is to capture the execution context of a migrating thread completely so that the context can be correctly reproduced at the destination worker node. The set of execution states so reconstructed will be used to instantiate a newly

created thread instance, which represents the migrated thread and resumes execution at exactly the same place as it was frozen at the console node.

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 has sub-classed the `java.lang.Thread` class. The bytecode instructions of this method will be executed by the BEE, which may further call other methods and so on. The thread completes its task when this `<java.lang.Thread.start()>` 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.

To implement the method-invocation mechanism in JESSICA, we have defined a structure called *Java Method Frame* (JMF) to construct the method-invocation stack of a thread. The purpose of JMF is to store the context of the current executing method. A JMF contains:

- a program counter PC which points to location of the bytecode currently being executed
- the next instruction NPC, it points to the next instruction to execute after finishing the current PC
- a stack pointer SP that points to the top of the current method stack
- an array for storing the local variables of this method
- the method stack
- other miscellaneous information such as Java Exception information

Note that information contained in a JMF is architectural neutral.

When a running thread that is executing a method (M-0) tries to invoke another method (M-1) from within (M-0), the thread will instantiate a new JMF (JMF-1) for storing the execution context of the newly called method (M-1). This JMF-1 is then pushed onto the runtime stack of the running thread immediately before method (M-1) starts to execute. Note that the JMF sitting on the top of the runtime stack represents the execution context of the current method. When the method (M-1) returns at a later time, its method execution context will also be discarded, which is done by popping the JMF-1 from the runtime stack. As a result, the execution context of the previous method (M-0), that is, JMF-0 is also restored automatically. This is because JMF-0 was pushed onto the runtime stack before JMF-1 when method (M-0) was invoked. Therefore when JMF-1 is popped, JMF-0 will again be sitting on the top of the runtime stack. Now since the NPC of JMF-0 at this point is pointing to the next instruction immediately following the one that invoked M-1, the thread can continue to execute correctly.

Consequently, with this JMF structure defined the method-invocation stack of a running Java thread can be represented as a sequence of JMFs stored in the runtime stack of the thread.

5.3 Java Thread Migration – First Attempt

To illustrate how thread migration is carried out and how execution states are captured and transferred based on the JMF structure, let us follow the Factorial.java example below, which recursively calculate the factorial of *10* (Fig 5.2, line 06). The compiled bytecode instructions are also shown in Fig 5.3. Note that the numbers following the line numbers in Fig 5.3 represent the corresponding location of the bytecode instructions as stored in the program counter (PC).

```
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 }
```

Fig 5.2: Factorial.java

```
00 Method void main(java.lang.String[])
01    0 bipush 10
02    2 invokestatic #4 <Method int f(int)>
03    5 pop
04    6 return
05
06 Method Factorial()
07    0 aload_0
08    1 invokespecial #3 <Method Object()>
09    4 return
10
11 Method int f(int)
12    0 iload_0
13    1 iconst_1
14    2 if_icmpne 7
15    5 iconst_1
16    6 ireturn
17    7 iload_0
18    8 iload_0
19    9 iconst_1
20   10 isub
21   11 invokestatic #4 <Method int f(int)>
22   14 imul
23   15 ireturn
```

Fig 5.3: Disassembled bytecode of the class `Factorial`

When the Factorial program is invoked, the *main* thread of the virtual machine will enter the `<main()>` method of the `Factorial` class (Fig 5.2, line 06), pushing the first JMF onto the runtime thread stack. And when it executes the 2nd bytecode instruction (Fig 5.3, line 02), it will call the factorial method `<f()>`, which will cause another JMF to be pushed. After that, the method `<f()>` will recursively call itself at PC = 11 (Fig 5.3, line 21), and push more JMFs onto the runtime stack.

After a number of iterations, suppose `<f()>` has recursed 3 times and the running thread is frozen at PC = 8 of the method `<f()>` (Fig 5.3, line 18), when the local Migration Manager (MM) triggers a migration. After the current instruction at PC = 8, i.e. `<iload_0>` has been completed, the thread execution context can be represented as pictorially follows:

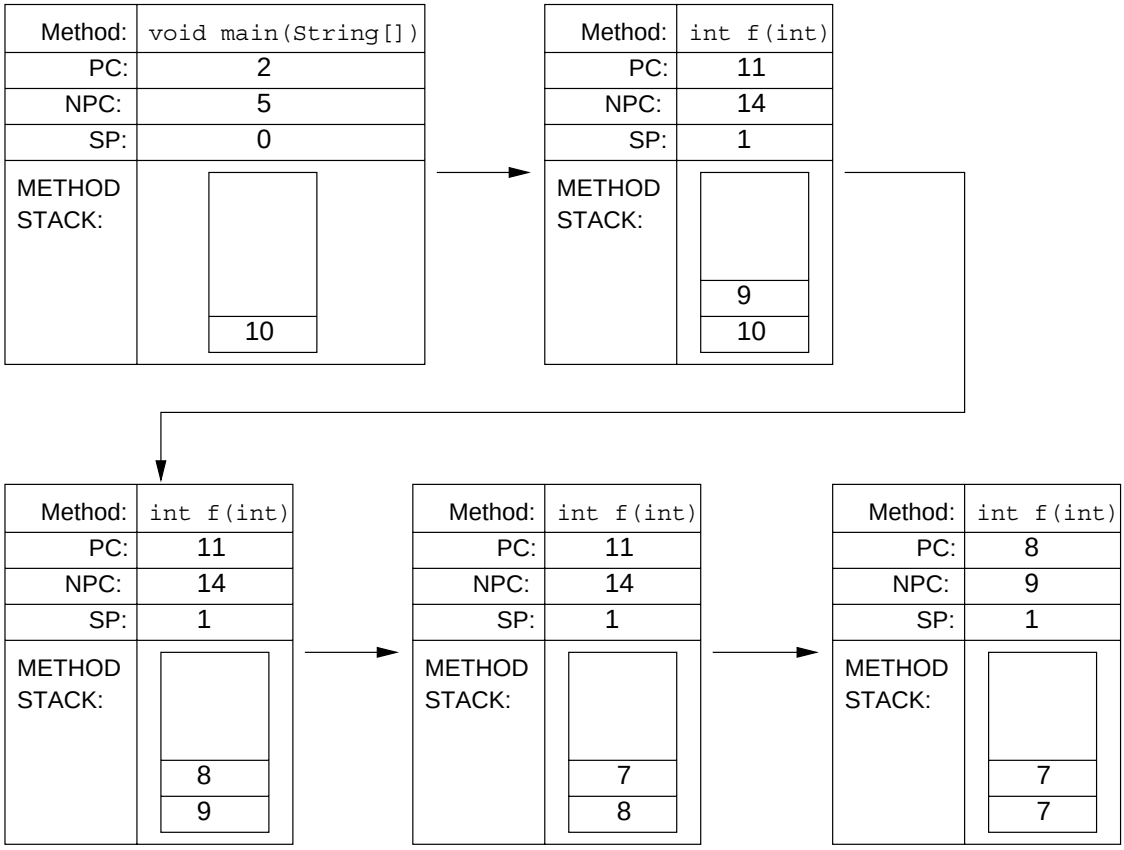

Fig 5.4: Execution context of thread *main* after a number of iterations

Note that for the tail-JMF there are two elements on the method stack, both of them are values of the method calling parameter integer `n`, after the execution of the load instruction `<iload_0>` twice at PC locations 7 and 8 (Fig 5.3, line 17, 18).

Suppose at this moment this *main* thread migrates, the above 5 JMFs which constitute the entire execution context of *main*, are packed and shipped to a worker node by the local Thread Manager (TM). After the execution context arrives at the destination worker node, they are unpacked and reconstructed at the worker node, and eventually bind to the execution context of a newly created *slave* thread. When the *slave* thread resumes execution, it continues from the tail-JMF, and updates the value of PC to PC = 9 `<iconst_1>` (Fig 5.3, 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 execution correctly from the point where the *master* was previously stopped, just as if the thread migration had not taken place. 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 integral elements: *8*, and *5040*, i.e. the value of `n` and the evaluated value of `<f(7)>` respectively. BEE would then updates the value of PC to

14 <imul> (Fig 5.3, line 22), which would multiplies the 2 elements on the method stack together, the result <f(8)> is finally return to the upper level when BEE executes the next instruction, PC = 15 <ireturn> (Fig 5.3, line 23). The *slave* executes this way until all the JMF are exhausted, by then the *slave* will have completed its task. The console TM will eventually receive a "done" message from the worker node.

5.4 Machine Dependent States

If a thread completes its execution without calling other methods except the <java.lang.Thread.start()>; or the calling of other methods is performed by directly executing one of the `invokeMethod`[1] bytecode instruction, then the scheme of using a sequence of JMFs described above can sufficiently capture and represent the states of this running thread at any point of its execution. On the other hand, if this thread calls a native method or the BEE tries to execute a bytecode instruction that within its implementation would call another method, this will generate machine dependent states that a sequence of JMFs alone cannot sufficiently capture the entire execution context. The following example illustrates this.

```
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 }
```

Fig 5.5: Bar.java and Foo.java

[1] A Java method can be invoked by executing one of the `invokevirtual, invokespecial, invokeinterface and invokestatic` bytecode instructions.

```
00 Method Bar()
01    0 aload_0
02   1 invokespecial #4 <Method java.lang.Object()>
03   4 aload_0
04   5 getstatic #5 <Field int count>
05   8 dup
06   9 iconst_1
07  10 isub
08  11 putstatic #5 <Field int count>
09  14 putfield #6 <Field int id>
10  17 return
11
12 Method <clinit>
13   0 bipush 100
14   2 putstatic #5 <Field int count>
15   5 return
```

Fig 5.6: Disassembled bytecode of the class `Bar`

```
00 Method Foo()
01   0 aload_0
02   1 invokespecial #5 <Method java.lang.Object()>
03   4 return
04
05 Method void main(java.lang.String[])
06   0 new #1 <Class Bar>
07   3 invokespecial #4 <Method Bar()>
08   6 return
```

Fig 5.7: Disassembled bytecode of the class `Foo`

In this example, the static block in the class `Bar` defines the class initialization routine (Fig 5.5, line 03), internally denoted as method "`<clinit>`" (Fig 5.6, line 12), is to be executed once when the class is first loaded by the JVM. According to this example, the static block will initialize the class variable `count` to *100* when `Bar` is loaded (Fig 5.5, line 03).

When the Foo program is executed, supposed the *main* thread of the virtual machine, enters the `<main()>` method of the class `Foo` (Fig 5.5, line 08) and executes the first instruction at PC = 0 `<new>` (Fig 5.7, line 06), to create a new object instance of class `Bar`, a JMF for the `<Foo.main()>` method will be pushed onto the runtime thread stack. At the point, the BEE which tries to execute the instruction `<new>`, will lookup and load the class `Bar`, as given in the argument to the instruction (Fig 5.7, line 06). After the class is loaded, BEE will then check if there is any class initializer to execute. Finally BEE will request the Distributed Object Manager (DOM) to allocate a memory block to hold the `Bar` object. The following

code segment shows a simplified internal implementation of the <new> bytecode instruction in C/C++ for the BEE, which summarizes the above actions:

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

Fig 5.8: Simplified implementation for instruction new in BEE

When the *main* thread invokes the "<clinit>" method at line 05 of the <instruction_new> function (Fig 5.8), it will push another JMF onto the runtime thread stack to record the entering of a new method context. If at the moment the local MM initiates a migration when the thread is trying to execute the first instruction at PC = 0, <bipush> (Fig 5.6, line 13), the *main* thread will be frozen after the <bipush> instruction has been completed successfully. And the execution context of the thread can then be represented by the following sequence of JMFs:

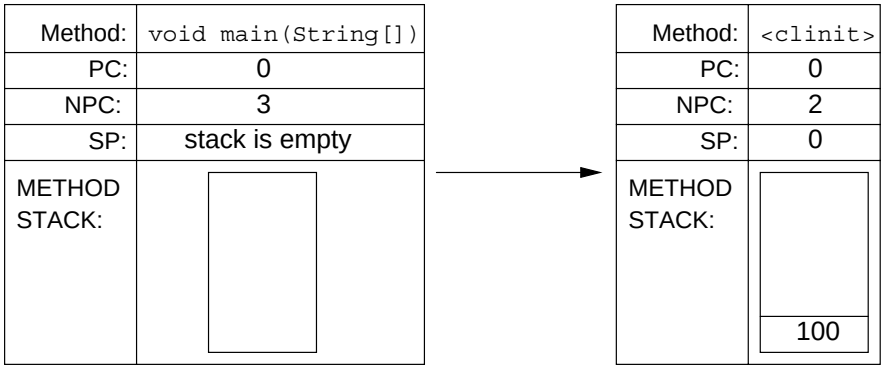

Fig 5.9: Execution context of thread *main* after entering the class initializer "<clinit>"

When the *slave* thread resumes 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 return (Fig 5.6, 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 5.3, we will continue the execution within the method context of <Foo.main()> and execute the next instruction at PC = 3 <inovkestatic> (Fig 5.7, 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 continue at line 06 (Fig 5.8) of the <instruction_new()> function code after the "<clinit>" method return. That is to set the needInitialize flag to false, obtain a memory block from DOM, and finally push the memory handle onto the method stack (Fig 5.8, line 06-13); before returning to the method context of <Foo.main()> at PC = 3 <invokestatic> (Fig 5.7, line 07). In other words, there are certain state information of the execution context failed to be represented between the two JMFs (Fig 5.10).

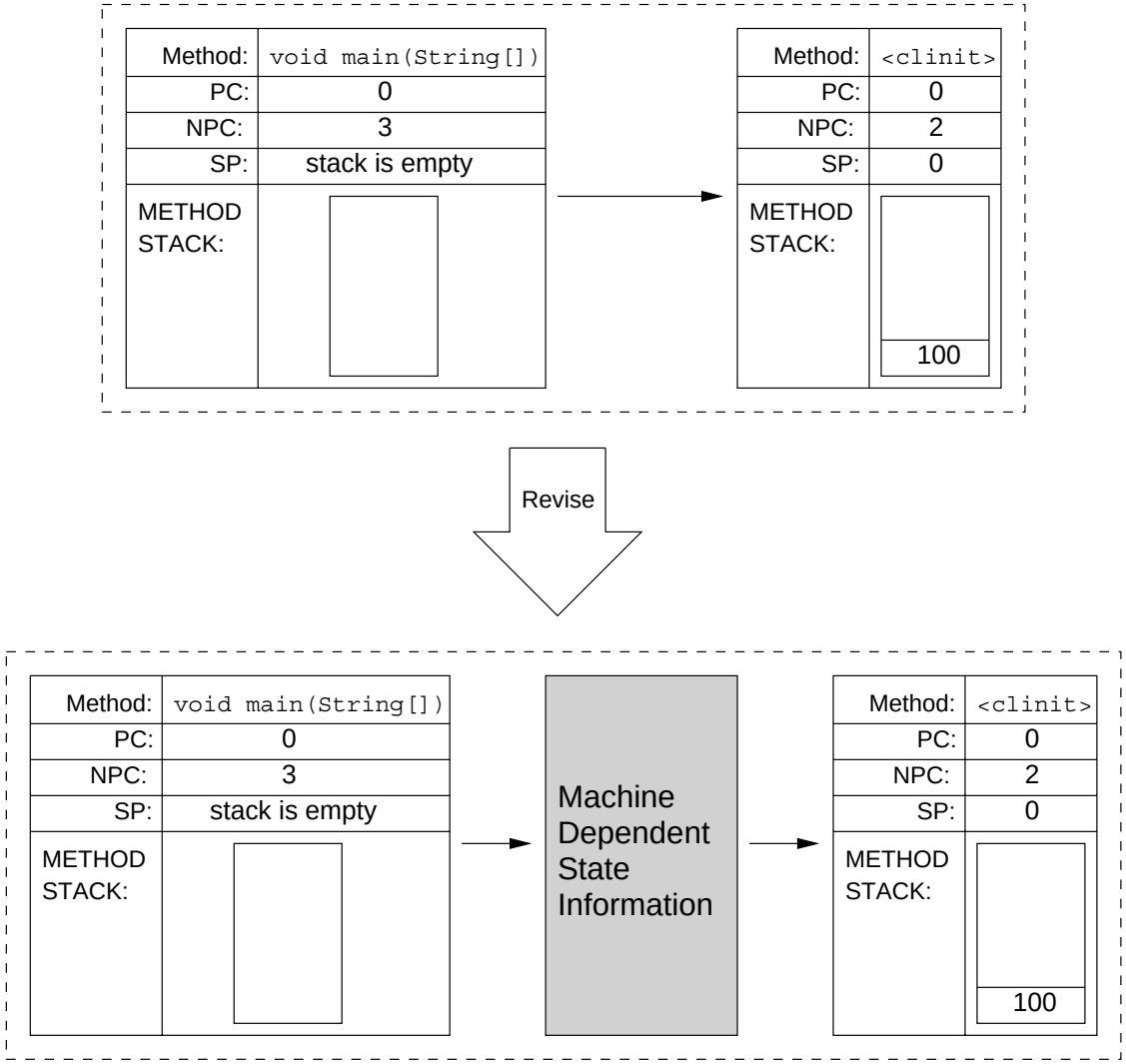

Fig 5.10: Revised execution context of thread *main* after entering the class initializer "<clinit>"

The missing information, including the return address at line 06 of the <instruction_new()> implementation code (Fig 5.8), are in fact stored in the runtime C-stack[2] of the JESSICA Daemon at the console node. They are by all means implementation and hardware dependent, as illustrated in Fig 5.10 above.

[2]The C-stack we described here is the runtime stack of a UNIX process. We called it a C-stack to differentiate from the runtime thread stack of the BEE.

5.5 Incremental Thread Migration by means of Delta Execution

As the shaded block in Fig 5.10 above is machine dependent and so cannot be shipped across, here we propose an alternative approach to thread migration that is independent to the lower level implementation of the operating system.

5.5.1 Two Types of Java Method Frames

Concluding from the two examples discussed above, we can categorize JMFs into two types:

"B" Frame –

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 (`invoke[virtual| special| interface| static]`) instructions. For example, in Factorial.java, the calling of `<Factorial.f()>` at PC = 2 of `<Factorial.main()>` (Fig 5.3, line 02) as well as the recursive invocation in `<Factorial.f()>` itself at PC = 11 (Fig 5.3, line 21) will both push a "B" frame to the stack.

"C" Frame **–**

The JMF that is pushed to the runtime thread stack when the implementation code (assuming in the language C/C++) of BEE or a native method explicitly invokes a method through the `<vmExecuteJavaMethod()>` function, such as at line 05 of the `<instruction_new()>` implementation shown in Fig 5.8. It is the machine dependent states introduced by the implementation code before pushing this "C" frame onto the thread stack that cannot be captured directly.

Consequently, for any thread execution context, the correct representation should be a sequence of B frames and C frames interleaving between each other, with a set of machine dependent states inserted before each C frame. Note that every queue must begin with a "C" frame, as any thread execution must be initiated by the VM implementation itself (Fig 5.11).

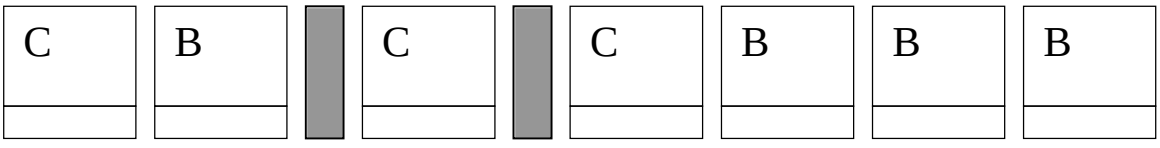

Fig 5.11: Revised thread execution context representation, shaded blocks are the sets of machine dependent states not captured by JMFs

5.5.2 Delta Execution

Our approach is to partition a thread execution context into chunks of JMFs, known as *delta sets*, ΔE_i. (Fig 5.12) 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 local TM of the console node performs a migration operation, it packs and ships the tail *delta set* in the sequence of thread execution context. The *slave* thread at the destination worker node will bind to this *delta set* and resume execution until the JMFs in this set is exhausted. After which the execution control will return back to the *master* thread, to let it finish whatever machine dependent context whose states cannot be shipped to the destination, such as from line 06 to line 13 in the `<instruction_new()>` example (Fig 5.8). Immediately after the machine dependent execution is completed, control flow will migrate back to the *slave* with the next *delta set* so as to continue execution at the worker node.

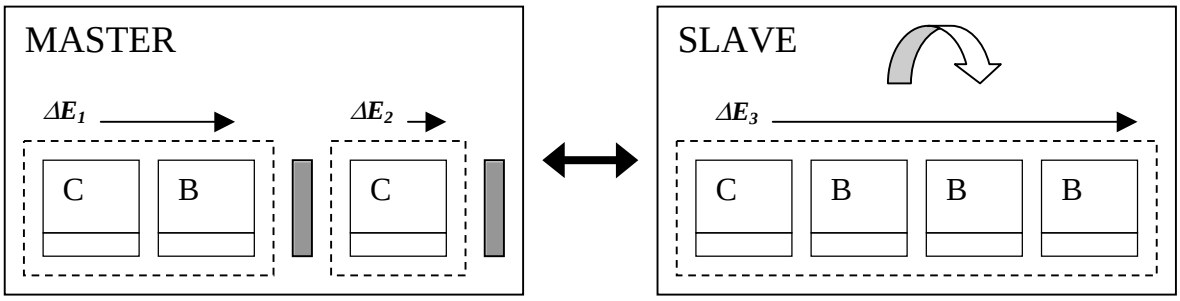

Fig 5.12: *Delta sets* are sent to the *slave* thread one by one for execution

With this arrangement of *Delta Execution*, every time the destination only incrementally advances the execution states of the migrating thread by *delta*. We are able to isolate those non-migratable machine dependent context and to have them executed locally at the console; while at the same time machine independent context are able to migrate across machines of any hardware architecture and execute remotely.

Chapter 6

SSI Transparency Support

6.1 Overview

This chapter discusses the implementation of the support for SSI transparency in JESSICA. SSI transparency is an indispensable feature in JEECIA, with this the support of migration transparency for threads can follow immediately. It is this SSI transparency that provides a Global Thread Space for threads executing on different nodes in the cluster to see each other as if they were running on the same virtual machine, where threads can freely move around the cluster without breaking this view. Following the *home model* as described in chapter 4, it extends the parallelism of a JVM that spans over a cluster without changing the semantics of runtime interactions between objects, therefore making JESSICA compatible with the standard JVM, in addition to support the Serial-Program-Parallel-Subsystem (SPPS) programming model. The core implementation of SSI transparency 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 transparent network communication and file operations, distributed thread synchronization and remote exception. Their implementations are discussed in the following sections.

6.2 The Master-Slave Design for Thread Migration

In JESSICA, when a thread running on the console node migrates, it does not actually pack up itself and *move* to the destination worker node. Instead, it is split into two cooperating entities, with one running at the original console node, called the *master;* and the other running at the destination node, called the *slave*. The slave thread is in fact created at the destination node anew and acts as the migrated image to continue the execution of the original thread. The master thread remaining at the console node is actually the original migrating thread, which is now reduced and be responsible to perform any location-dependent operations like I/O on behalf of the slave thread, plus other message forwarding between the slave and the rest of the system. The master and slave pair is responsible to carry out the interactions between the console and the worker nodes as described in chapter 4 in order to maintain migration transparency.

The following steps are taken when the current thread running on the console tries to migrate to another worker node:

1. The Thread Manager (TM) at the console node freezes the migrating thread and extracts the execution states in the form of a series of Java Method Frames (JMF) from the local Bytecode Execution Engine (BEE). The JMFs are then grouped into a sequence of *delta sets* in order to isolate the machine dependent execution states, as explained in chapter 5.
2. The Migration Manager (MM) then identifies a destination worker node for the thread to migrate to, and to notify the corresponding MM at the destination node to prepare for the migration.
3. After receiving the migration notification, the destination MM requests the local TM to create a new thread instance to represent the migrated thread. This newly created thread instance is known as the *slave* thread.
4. The original thread at the console detaches itself from the local BEE and obtains its execution states from the TM as a sequence of *delta sets*. It is now known as the *master* thread which is responsible to control the execution of the slave thread that is created at the destination node.
5. After the instantiation, the slave thread at the destination creates a dedicated communication channel with the master at the console. This dedicated channel is used for sending control information and message redirection between the master and the slave.
6. Finally the slave thread at the destination node sends a ready message to the master to signify that it is ready to resume execution. The master thread at the console then sends the first *delta set* to the slave for execution.
7. The slave thread resumes its execution by using the JMFs from the delta set that it received. After finishing execution of the given delta set, it sends a 'more' signal to the master and ask for the next delta set to execute. The master thread at the console after receiving this 'more' signal, will complete any machine dependent execution that are not migratable, and then send the slave the next delta set to execute.
8. The last step is repeated until the whole sequence of delta sets is exhausted, which implies the original thread has completed the execution of its primary function `<java.lang.Thread.start()>`, or its equivalent. At this point, the master will notify the slave with an 'end' signal signifying that the execution has been completed, so that both threads will eventually terminate themselves.

Note that the 7th step of the above scheme is the critical step for the system to maintain migration transparency, it is in this step where redirections take place. While the slave thread is executing the delta set at the destination node, the master thread is also responsible to monitor the communication channel and see if there are any messages of redirection requests

sent from the slave thread. These messages can be network channel read/write operations or semaphore acquire/release operations where the master has to perform on behalf of the slave back at the console node, so as to maintain the network and location transparency. In addition, the master is also responsible to redirect any asynchronous signals sending from other running threads on the console node, so that the original thread appears to other threads as if it was still running on the console node and the migration has never took place.

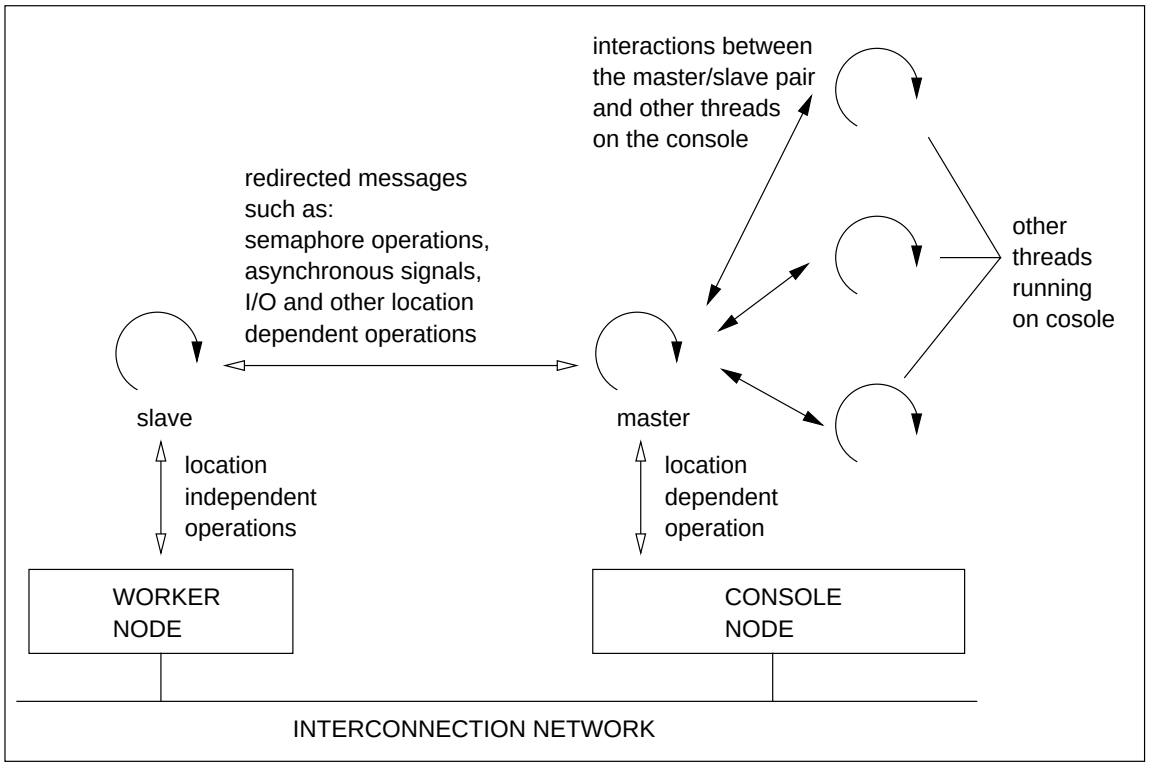

Fig 6.1: Interactions between the master and the slave thread that transparently hide migration from the rest of the system

With this design we are able to create a Global Thread Space and maintain the same semantics and relationships between all the objects in the execution environment as if there was no migration (Fig 6.1). This is because for the rest of the threads that are running on the console node, only the master thread is visible to them, they are not aware of the existence of the slave thread. All the interactions between the slave thread and the rest of the threads have to go through the master, and the fact of migration is transparently hidden by the master, who redirects the all interactions to the slave. 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. In addition, for the slave thread which is now running at a worker node, it will also appears that it is running on the console node, since all the location-dependent operations are transparently redirected back to the console and be performed by the master. As a result, the

execution environment observed by a running thread in JESSICA is the same as that of a standard JVM, no matter whether the thread has been migrated or not. Hence, multi-threaded applications runnable on a standard JVM can also be executed correctly on JESSICA.

In addition, the master-slave design also simplifies the implementation of network transparency and other location-dependent operations, by having the master to act as a proxy to forward any messages between the slave thread and the rest of the system. The redirection code can be inserted within the implementation of the operations themselves, without changing the interface definitions for invoking the operations; hence no modification is necessary for existing applications that perform the operations. For example, the change of communication end-points when a thread has been migrated can be transparently hidden by the master. The master takes care the redirection of network messages, so that the other parties trying to communicate with the migrated thread cannot aware of the change of location. The redirection code can be inserted within the `<read()>`, `<write()>` and other socket operations in the set of `java.net` classes without changing their method signatures. Moreover, because applications will be automatically linked to the modified network classes when they are first invoked, they do not need to be re-compiled before starting on the JESSICA system and still be able to correctly handle the changes in socket end-points as a consequence to thread migrations.

Finally, the design of having a master thread running at the source node and a slave thread running at the destination node is also necessary for implementing *Delta Execution* for heterogeneous thread migration. This is because by leaving the master thread behind at the source node, we are able to complete all the machine dependent executions that are not migratable with the help of this master thread, while all the machine independent executions can be performed by the slave thread running elsewhere.

6.3 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.

A possible solution is to implement these mechanisms in a centralized approach. TMs running on all the nodes are integrated together so as to perform cooperative scheduling in a global manner. When a thread *T* running on node *A* tries to release a semaphore, which would in turn causes another thread *T'* running on node *B* to be re-scheduled, the TM of node *A* will send a reschedule message to node *B* to place thread *T'* back to the ready queue. In addition, a thread cannot directly operate on a semaphore, since this will create another race condition when more than one thread tries to operate on the same semaphore at the same time. Instead, operations of a semaphore have to be performed by a manager of the semaphore, which can be the thread who created the semaphore. However the global scheduling scheme in the centralized approach can make the solution less scalable and less reliable. This is because the scheme requires each TM to be aware of the existence of all the other TMs running in the system and their locations, so that scheduling messages can be directed to the right TM. Moreover, all the TMs have to be notified whenever a new node is added to or removed from the system. In addition, the implementation of forwarding semaphore operations back to their managers is also non-trivial.

Instead, 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 following code segments illustrate how the above mechanism is implemented at the master and the slave thread:

```
// Master waiting for
// redirection request
// at the console ...


00  while (true) {
01    req = waitForRequest();
02    switch (req.cmd) {
03    case REMOTE_SEM_ACQUIRE:
04    semAcquire(req.sem);
05    replyResult(ACQUIRE_OK);
06    break;
07    case REMOTE_SEM_RELEASE:
08    semRelease(req.sem);
09    replyResult(RELEASE_OK);
10    break;
11    // ...
12    // ...
13    // ...
14    } // end switch
15  } // end while
```

```
// Slave performs the following
// remote semaphore operations
// to redirect the operations back
// to the master

00  void remoteSemAcquire(Sem s) {
01    Request req;
02    req.cmd = REMOTE_SEM_ACQUIRE;
03    req.sem = s;
04    sendRequest(req);
05    Reply rep = waitForReply();
06    assert(rep == ACQUIRE_OK);
07  }
08
09  void remoteSemRelease(Sem s) {
10    Request req;
11    req.cmd = REMOTE_SEM_RELEASE;
12    req.sem = s;
13    sendRequest(req);
14    Reply rep = waitForReply();
15    assert(rep == RELEASE_OK);
16  }
```

Fig 6.2: Pseudo-code for implementing Cooperative Semaphore in JESSICA

6.3.1 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.

6.4 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 6.3). 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.

```
00  class Foo {
01    public float divide(int a, int b) {
02          float c;
03          try {
04                c = a / b;
05          } catch (ArithmeticException e) {
06                System.out.println("divide-by-zero!");
07                c = float.NaN;
08          } // end try-catch
09          return c;
10    } // end divide
```

Fig 6.3: A simple try-and-catch example of exception

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 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.

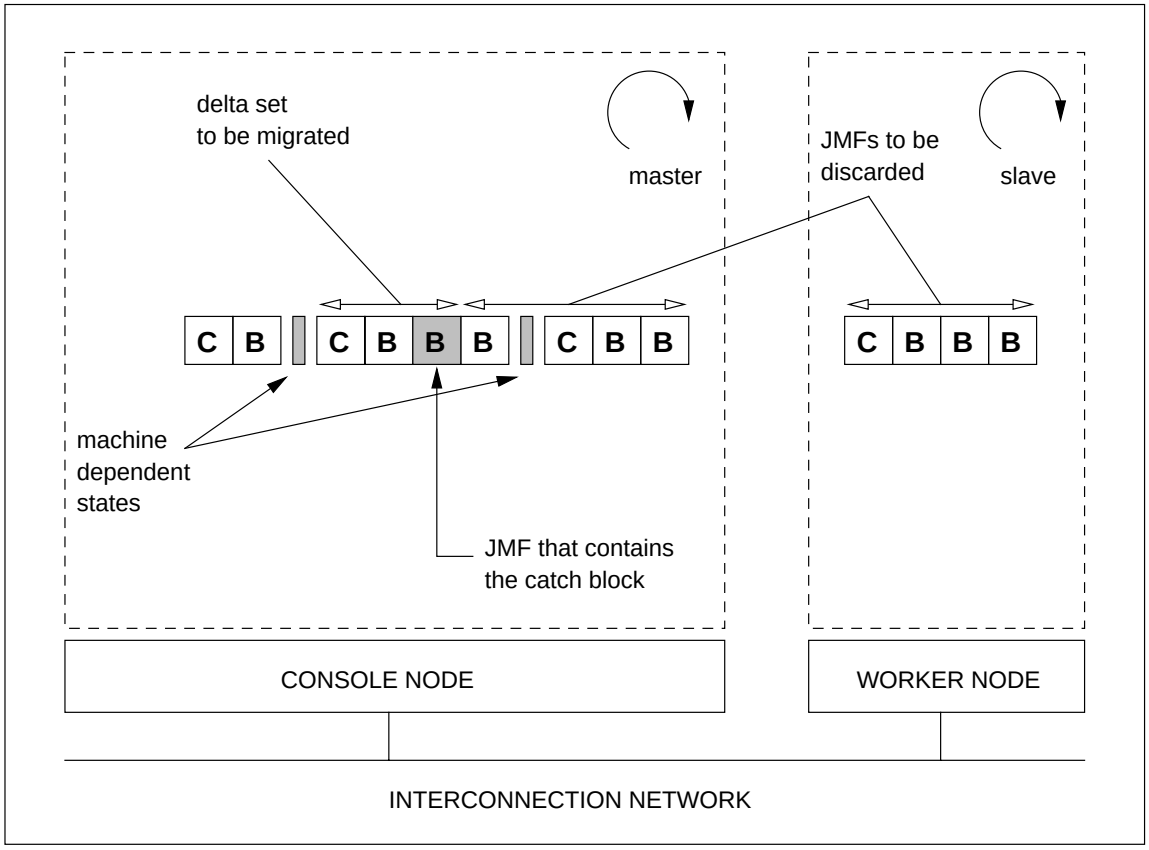

Fig 6.4: The mechanism for handling Remote Exception

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 6.4). 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.

6.5 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-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 following code segment demonstrates how the `java.io.FileInputStream.read()` (Fig 6.5) is augmented. Note that because the method needs direct access to the underlying operating system, it is a native implementation.

```
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 }
```

Fig 6.5:Code segment for the implementation of `java.io.FileInputStream.read()` with redirection

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:

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

The following code segment demonstrate how the to modify the method for creating a datagram socket:

```
00 // implementation of java.net.DatagramSocket-
01 // .datagramSocketCreate() method with redirection
02
03 void java_net_DatagramSocket_dataSocketCreate (
04    struct Hjava_net_PlainDatagramSocketImpl *this) {
05    if (currentThread->isSlave) {
06          Request req;
07          req.cmd = JAVA_NET_DATAGRAMSOCKET_DATAGRAMSOCKETCREATE;
08          req.obj = this;
09          sendRequest(req);
10          Reply rep = waitForReply();
11          assert(rep.result == SUCCESS);
12    } else {
13          int fd = socket(AF_INET, SOCK_DGRAM, 0);
14          unhand(unhand(this)->fd)->fd = fd;
15          return;
16    }
17 }
```

Fig 6.6: Code segment for the implementation of `java.net.DatagramSocket.datagramSocketCreate()` with redirection

Now for the master thread to be able to handle the redirection request correctly, its service loop has to be augmented as follows:

```
// Master waiting for redirection request at the console ...
00  while (true) {
01    req = waitForRequest();
02    switch (req.cmd) {
03    case REMOTE_SEM_ACQUIRE:
04    semAcquire(req.sem);
05    replyResult(ACQUIRE_OK);
06    break;
07    case REMOTE_SEM_RELEASE:
08    semRelease(req.sem);
09    replyResult(RELEASE_OK);
10    break;
11    // ...
12    // ...
13    case JAVA_IO_FILEINPUTSTREAM_READ:
14    Reply rep;
15    rep.result = java_io_FileInputStream_read(req.obj);
16    replyResult(rep);
17    break;
18    case JAVA_NET_DATAGRAMSOCKET_DATAGRAMSOCKETCREATE:
19    Reply rep;
20    java_net_DatagramSocket_dataSocketCreate(req.obj);
21    rep.result = SUCCESS;
22    replyResult(rep);
23    break;
24    } // end switch
25  } // end while
```

Fig 6.7: Service loop of the master thread that handles redirection request of location-dependent operations from Slave

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).

6.6 Distributed Garbage Collection

JESSICA employs a decentralized approach for memory management, where the Distributed Object Manager (DOM) running on each node of the cluster is responsible to manage its own share of distributed shared memory. This scheme allows multiple DOMs to perform memory management operations concurrently, and the parallelism of the system can be improved. On the other hand, care has to be taken by these DOMs when they are performing garbage collection because objects that are not referenced by local objects may still be referenced by

other objects that are located on other nodes of the system. JESSICA proposes a simple mark-and-sweep algorithm for garbage collection.

In the simplest case when there is no migration, to performing garbage collection the DOM will start tracing from a list of local root objects which include all the running threads in the local node, and to mark all the objects that can be reached from them. Any remaining local objects that are not marked will imply they are not reachable by any means and therefore can be safely reclaimed. However, when there is migration, those unmarked objects may still be reachable either directly or indirectly by migrated threads running on other nodes in the system. A necessary requirement for performing garbage collection correctly is therefore to identify all the local objects that are not reachable from the list of local root objects but are still being referenced remotely.

JESSICA proposes a simple solution to address the issue by partitioning the global shared memory into a number of shares so that each DOM is responsible to manage its own share, as a range of contiguous memory. Each DOM satisfies local object allocation requests with the memory space from its own share. As a result, by looking at the address of where an object is located, it is simple to deduce which DOM is the owner of this object. Hence, during the marking phase of garbage collection, each DOM will maintain a list of remote objects for each of the other DOMs where all the objects in a given list belong to the same DOM. At the end of the marking phase these lists of objects are then forwarded to their respective owners, which as a result would enable a DOM to identify all the objects that are being referenced remotely by objects residing on other node of the cluster. Eventually, the DOMs are able to carry out the sweeping phase and reclaim objects that are not referenced by anyone, and the operation of distributed garbage collection can then be completed successfully.

Notice that distributed garbage collection is a classical research problem by itself and here we have only opted for a trivial solution for the sake of implementation. Interested reader can refer to [8] for more information.

Chapter 7

Performance Evaluation

7.1 Overview

In this chapter we present the results of our experiments with the JESSICA system. The experiments have verified that considerable speedup is achievable by migrating Java threads transparently over a cluster and to execute them in parallel. Section 7.2 studies the operating cost of primitive operations introduced as a result of allowing threads to distribute around a cluster and to execute on remote nodes. Section 7.3 examines the migration latency for resuming a migrated thread. Section 7.4 demonstrates the performance improvement with various types of multi-threaded applications. The applications are standard applications that can be found in the literature of Parallel Computing, they include an approximation of the value π by evaluating an integral, a recursive ray-tracing program on a simple 3D scene and a Red-Black Successive Over-Relaxation program where elements of a huge matrix (1024x1024) are interpolated. Finally, we draw our conclusions on the performance of the JESSICA system from the experiments conducted in section 7.4. In addition, techniques of how to improve performance in JESSICA are also presented in the last section.

7.1.1 The Experiment Environment

The experiments presented in this chapter are conducted on the HKU Pearl Cluster, using 12 SUN SPARC Ultra-1 machines that are connected together by a Fore ASX-1000 ATM switch. The ATM switch supports a point-to-point bandwidth of 155Mbps. Of the 12 SUN SPARC Ultra-1 machines, 8 of them are Model 140 with 64MB RAM and the remaining 4 are Model 180 with 128MB of internal memory, they are all running the Solaris 2.6 operating system. Experiments in this chapter that are running on eight or less processors are all conducted on the Model 140 machines.

7.2 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.

7.2.1 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 7.1), 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.

```
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 }
```

Fig 7.1: Class `Foo` measures the time it takes to update an integer object variable 1,000,000 times

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 7.1, 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 7.2. From the result shown, it is found that the access penalty for distributed object accesses is about 9 times more than the case when objects are allocated locally.

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

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:

Distributed Object Variable Access Penalty *(simple data type)* **:** *Local Object Variable Access Penalty (simple data type)* **:** *Local Stack Variable* **Access Penalty** *(simple data type)* = ***19.54 : 2.17 : 1***

7.2.2 Overhead from Cooperative Semaphore Operations

As explained in section 6.3, 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.

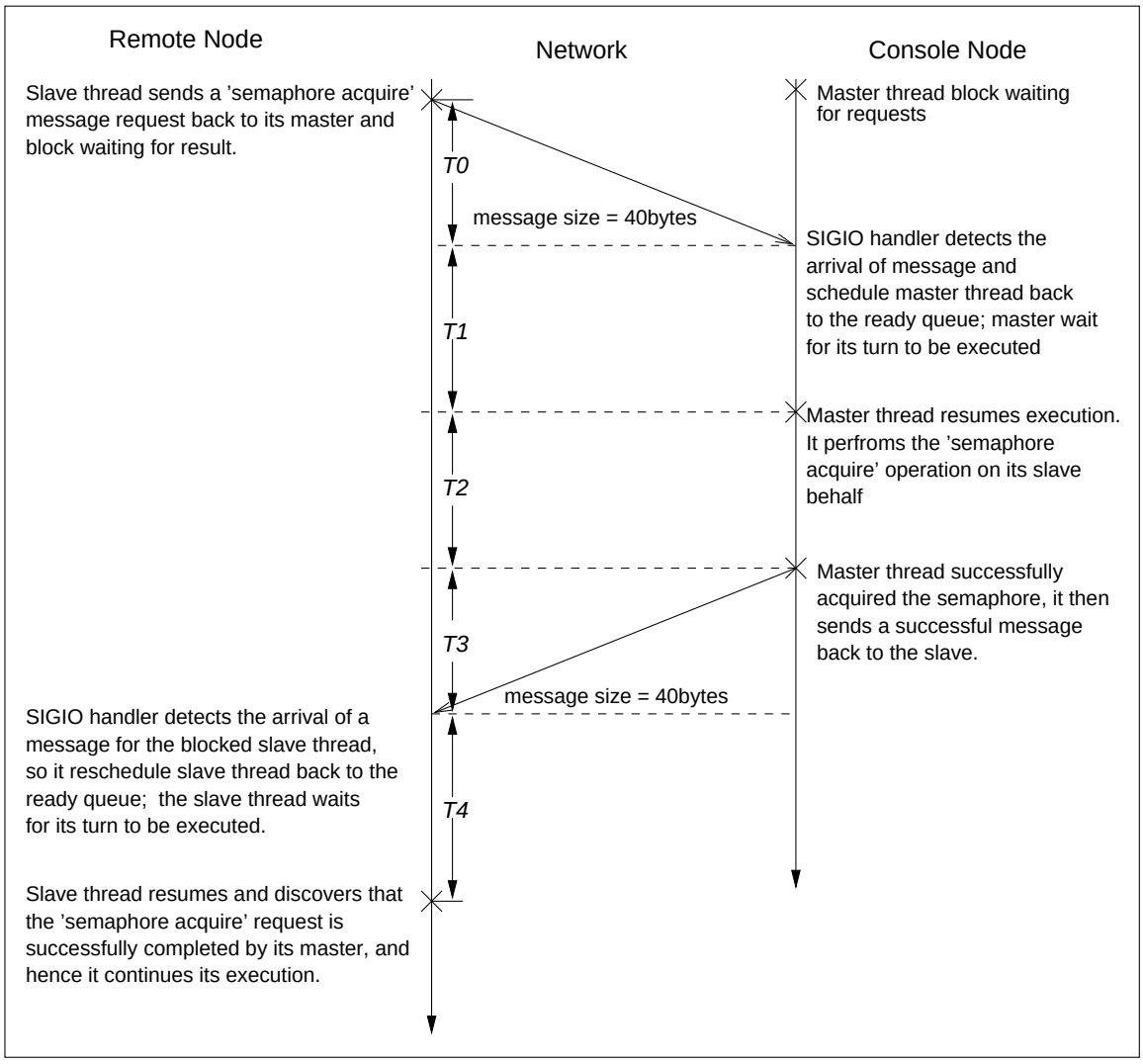

Fig 7.3: Cooperative Semaphore in action: a migrated thread performs an acquire operation

Consider when a migrated thread tries to acquire a Cooperative Semaphore (Fig 7.3), 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 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 7.3, 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.

7.3 Migration Latency

This section discusses the overhead for triggering a thread migration at the console and the time taken before the migrated thread can resume its execution at a remote node.

When the Migration Manager (MM) at the console triggers a migration, it notifies the Thread Manager (TM) with which thread and where to migrate. The TM will freeze the execution of the migrating thread and to contact the destination node asking it to prepare for the thread migration process. When the TM at the destination node receives the migration request, it will create a new thread object by cloning the migrating thread, using the `java.lang.Object.clone()` method, in order to inherit the internal states of the migrating thread. This newly created thread object becomes the slave and it represents the migrating thread at the destination. Afterwards, a new communication end point is created at the destination node for the establishment of a dedicated communication channel between the slave and the master for future message redirection. The slave thread then sends back the connection information, which include the destination's IP address and the port number, of this communication end point back to the console and waits for the migrating thread to connect to it. At the console, when the migrating thread receives the connection information, it will then establish the dedicated channel with the slave thread using the information received. After establishing the dedicated channel, the slave thread will send a ready signal back to the migrating thread signifying that it is ready to accept delta sets for execution.

When the migrating thread receives the ready message, the migrating thread will be turned into a master, to marshal and to ship the first delta set to the slave for execution. After sending a delta set, the master will be blocked waiting for the slave until it receive a notification from the slave saying the delta set has been completed. By then the master will continue its execution until the next set of machine dependent states are completely consumed, after which the next machine independent delta set will be sent to the slave. This process is repeated until all the machine independent delta sets and the machine dependent states are consumed, at this point the execution of the migrated thread will also be completed.

All the communications conducted between the entities residing on the console and that on the destination are done in a blocking fashion. When a thread tries to receive a message from its counterpart but the message has not yet arrived, the thread will be blocked and scheduled out by the local TM. When the message later arrives, the system will generate a SIGIO signal and causes the thread to be rescheduled back to the ready queue. The scenario described here is illustrated in Fig 7.4.

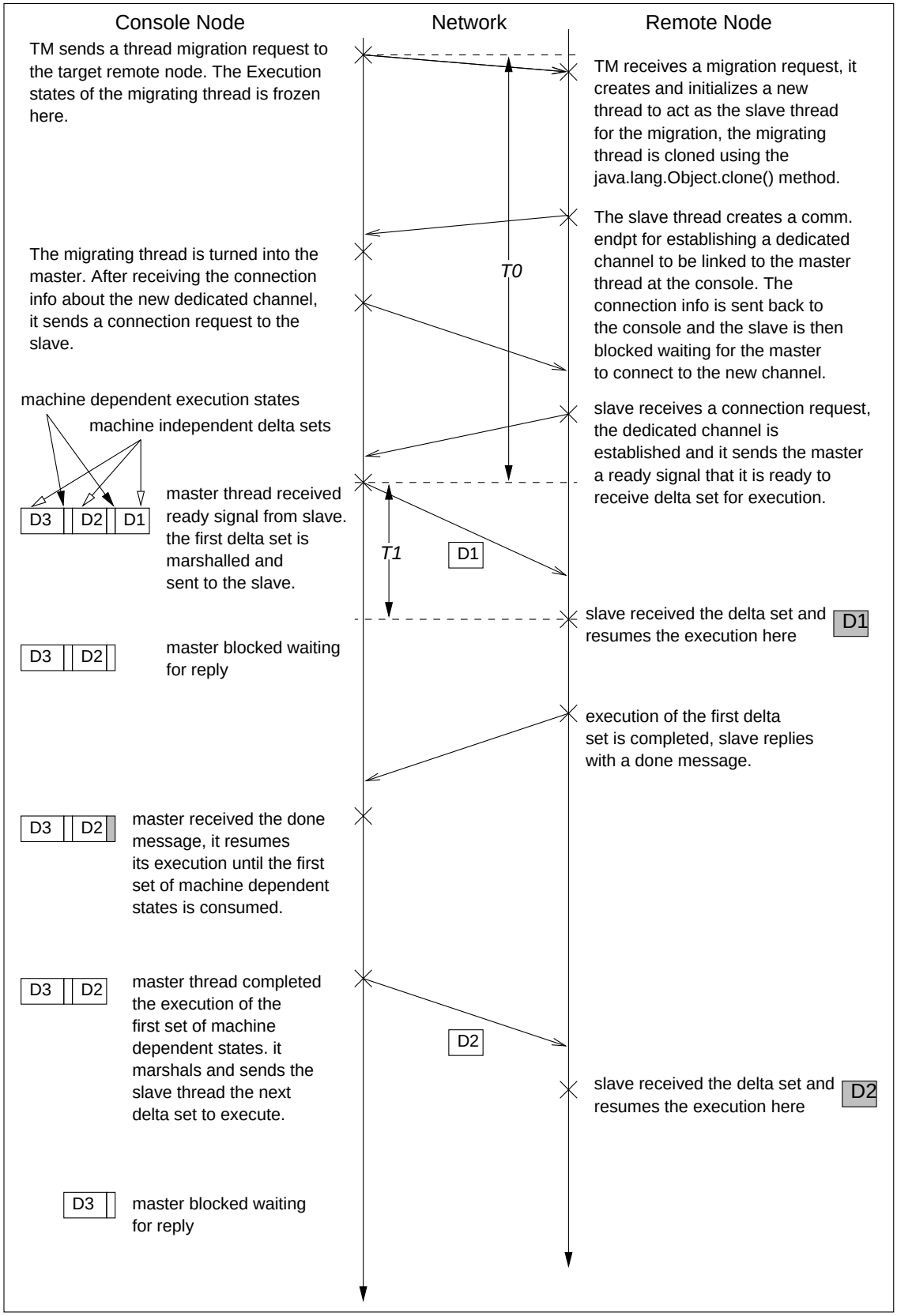

Fig 7.4: Interaction between entities in the home and the destination node when a thread is migrated

From the diagram, it can be seen that the migration latency is the time between the migrating thread is first frozen by the local TM and that 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 have the destination node to 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, sending the marshaled data across the network, and eventually be de-marshaled at the destination node. The value of T1 is therefore proportional to the size of the transferring delta set. As discussed in chapter 5, a delta set is a sequence of Java Method Frames (JMFs) that represents the method calling sequence of the current thread. Hence, the number of JMFs in a delta set depends on how many levels of method calls the current thread has made. Furthermore, each JMF stores the execution context of the current thread at a given method calling level that includes the method calling stack. The size of this method calling stack varies from cases to cases. Due to the varying factors, the migration latencies for different sizes of the delta sets are measured and the result is presented as follows:

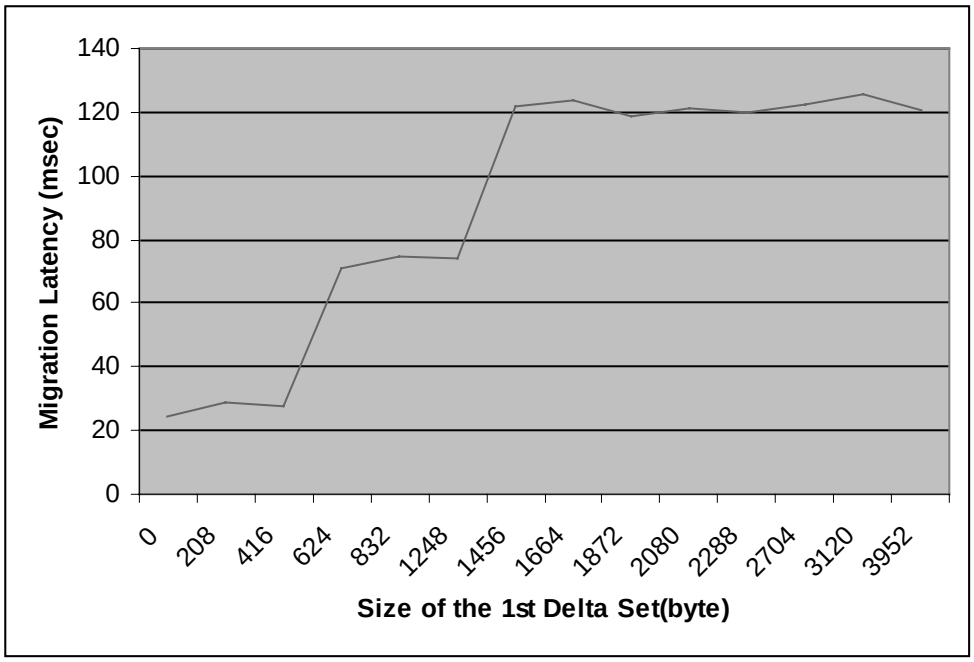

Fig 7.5: Graph of migration latency against size of the 1st transmitting delta set

According to the data collected, when the size of delta set is set to 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 4 handshake messages between the master and the slave (Fig 7.4). A further break-down of this T0 value reveals that the time required for executing the `java.lang.Object.clone()` method takes about 17 milliseconds, while on average it takes about 2 milliseconds for a

handshake message to be sent between the master and the slave. Now consider the case when a thread is migrated just before it starts executing the first instruction, the delta set will only contain one JMF and the method calling stack will be empty. The JMF will only contain the local variables of the current method and state variables such as program counter and stack pointer. If there is no local variable defined in the method, the size of this minimal JMF will be 208 bytes. In this case the minimum migration latency is therefore about 28.12 milliseconds.

In MOSIX, the minimum migration latency (projected) is about 3 milliseconds for a zero-size process, while in Arachne, the minimum time to migrate a thread with a 12-byte active record takes about 4.3 milliseconds. It can be seen that the time of minimum migration latency in JESSICA is about an order of magnitude higher than the other two systems. This is because the minimum time in JESSICA is dominated by the overhead for executing the `java.lang.Object.clone()` method, which accounts for over 60% of the migration time. What the clone method does is to create a new object and to copy data stored in the source object to the newly created object. Since both the source object and the newly created object are located in the DSM, the access cost is much higher than that for the local memory. In addition, the overhead of invoking the Java method also contributes to the relatively longer migration time.

7.4 Application Performance

This section presents the performance results of a number of multi-threaded Java applications that were tested on the JESSICA system. JESSICA was set to perform load distribution through remote execution – a worker thread will be automatically migrated to a remote node immediately after it is initiated. Newly created worker threads are migrated to remote nodes until there is a migrated worker thread running on each of the remote node. Since the applications are set to execute using the same number of threads as the number of processors available, we are able to study the effect on the execution performance when all the threads are running in parallel.

The execution times presented here do not include the sequential initialization time, such as the time taken for initializing elements of a matrix or that for loading data from a file. The timer is started immediately before the parallel computation begins so as to arrive at a more accurate estimation on the performance improvement.

7.4.1 Approximation of the value π (PI) by Integration

This is a simple parallel application that is adapted from the popular MPI book [38]. The application is ported to Java and the message-passing code of MPI is replaced by using Java Thread. The application is based on the formula:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \qquad \text{(Eq. 1)}$$

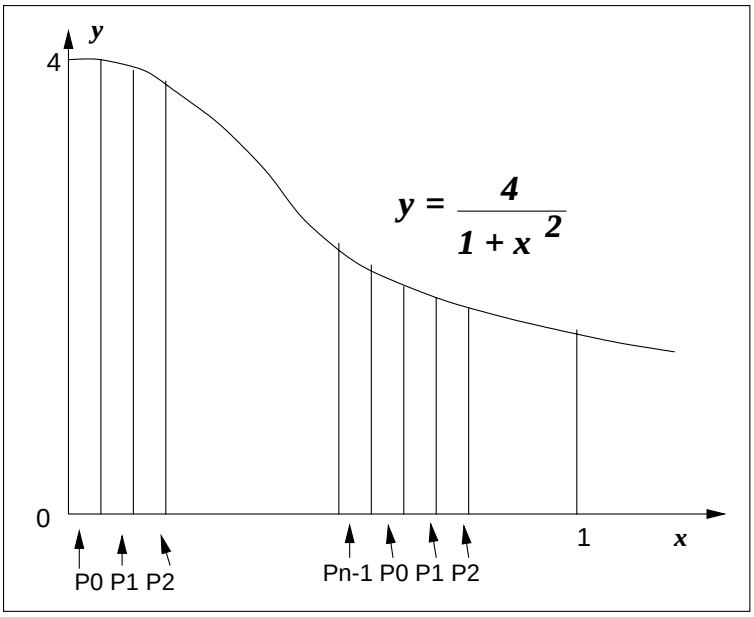

Fig 7.6: The graph of $y = \frac{4}{1+x^2}$

From the diagram above, the value of π is given by the area under the curve from 0 to 1. The computation is therefore straightforward: the [0, 1] interval in the integral is divided into 100,000,000 sub-intervals and each thread is responsible to compute a sub-area under the graph and to accumulate the partial sum. The value of π can then be obtained by adding together the partial results of all the threads after they have finished. The following code segment shows the main loop of a worker thread. Note that the sub-intervals for the worker threads to compute are interleaved between each other (Fig 7.7, line 08).

```
00 public void run() {
01    double partial_sum = 0;
02    int N = 100000000; // no of sub-interval
03    double h = 1.0/((double) N);
04    // my_thread_id ranges from 0 to no_of_worker_thread - 1
05    int begin = my_thread_id+1;
06    int end = N;
07    int width = no_of_worker_thread;
08    for (int i = begin; i <= end; i += width) { // the mainLoop
09          double x = h*((double i) - 0.5);
10          partial_sum += 1/(1+x*x);
11    }
12    result[my_thread_id] = 4*partial_sum;
13 }
```

Fig 7.7: Main-loop for each worker thread to compute a partial sum for the value π (PI)

The application was executed on 1, 2, 4, 8 and 12 processors, with 1, 2, 4, 8 and 12 worker threads running respectively. The result is presented as follows:

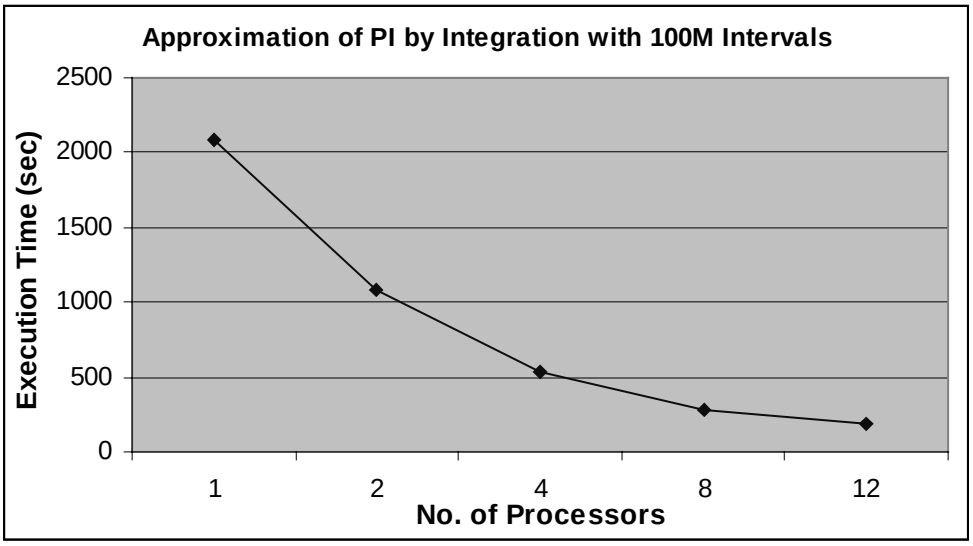

Fig 7.8: Total execution time against no. of processors

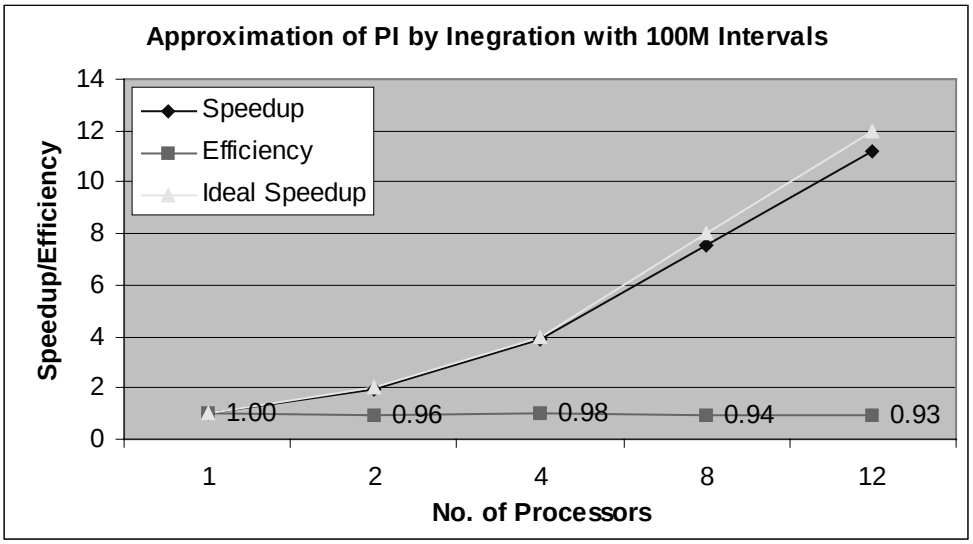

Fig 7.9: Speedup/efficiency against no. of processors

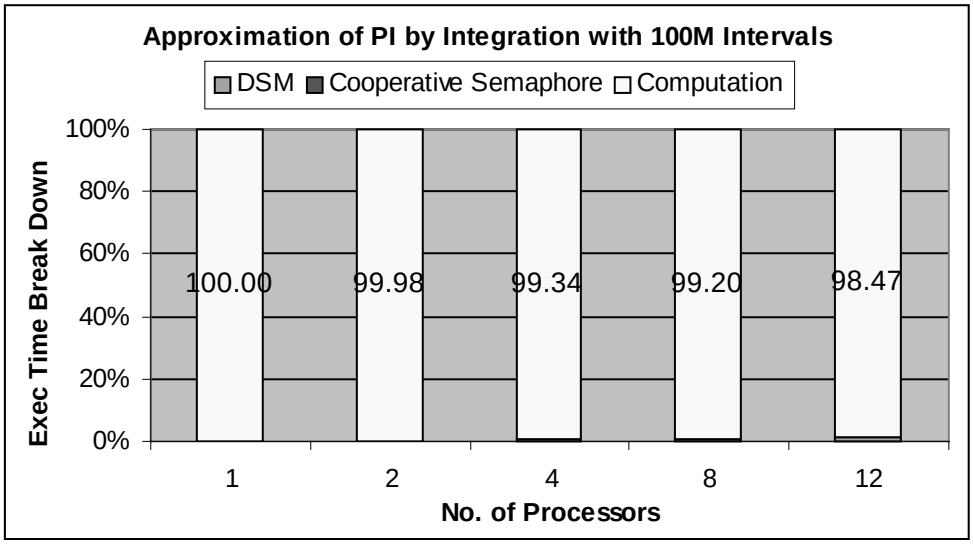

Fig 7.10: Percentage of execution time break down against no. of processors

From Fig 7.9 it can be found that we are able to obtain almost ideal speedup and efficiency. This is because once the computation is started, there is no communication nor coordination between the worker threads until the computation is over, hence there is no remote synchronization overhead. In addition, because all the variables that the worker threads accessed during the main computation loop are of simple data type, which are allocated from the local method stack, there is no additional DSM access penalty. As a result, we are able to achieve close to optimal speedup.

7.4.2 Recursive Ray-Tracing

Recursive ray-tracing is a technique in Computer Graphics to produce photo-realistic image for a 3D scene. It computes the pixel values of points lying on a 2D window when the 3D scene is projected onto it with respect to a given viewpoint (Fig 7.11). Each pixel value is the amount of light that passes through the window at the corresponding location and enters the viewer's eye at the viewpoint. To perform recursive ray-tracing, for each point lying on the surface of the window, a ray is traced backward from the viewpoint into the 3D scene through the point, until it hits the first object or the ray goes off to infinity. If the ray does not hit anything, then the amount of light that passes through the point will be simply the amount of the ambient light in the 3D scene. Otherwise, the amount of light emitted from the light sources in the environment that is reflected at the point of intersection back to the viewpoint is evaluated. In addition, a reflected ray at the point of intersection is spawned and trace into the scene recursively to determine the amount of light entering the viewpoint that is due to indirect reflections from other objects in the scene. When calculating the pixel value, the optical properties of objects' surfaces, such as diffuse and specular reflectivity are considered. In addition, the position and brightness of the light sources and the attenuation are also taken into account. On the other hand, for simplicity we do not consider refraction.

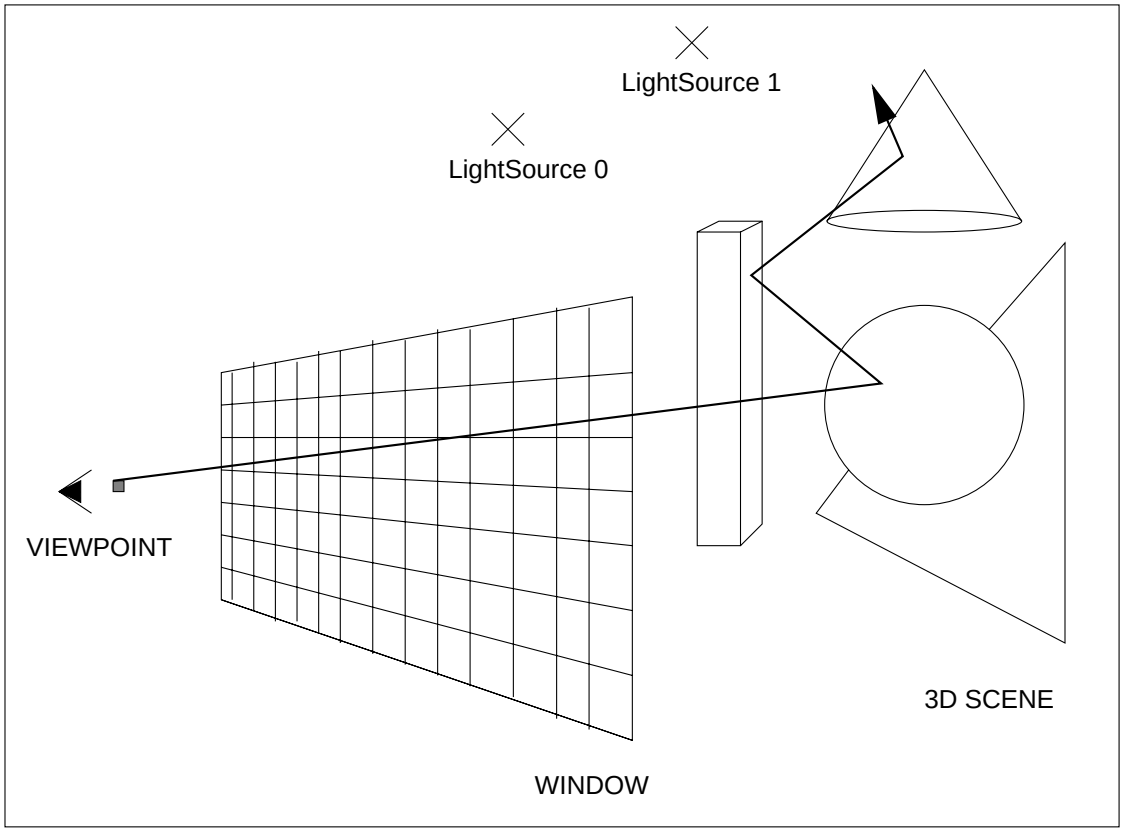

Fig 7.11: Recursive ray-tracing

Recursive ray-tracing is an ideal candidate for parallel computing because there are plenty of pixel values to compute. The computations of the pixels are independent and can be conducted in parallel.

We have implemented a multi-threaded recursive ray-tracer in Java for testing on the JESSICA system. The program is adapted from a single-threaded Java ray-tracer written by Sahin [11]. The ray-tracer supports simple object shapes such as sphere, triangle, and 2D plane. When the program is started, the 3D scene is loaded from the file system and stored into the DSM so that any migrated threads can have immediate access to the objects in the scene. After loading the 3D objects, a worker thread will be created and migrated to each of the remote node and start computing the pixels. We have also implemented an application-level load balancing scheme in the ray-tracer where each thread has to access a globally shared job queue to obtain the next line of pixels to trace. With such a scheme installed, all the participating processors will complete the computation at about the same time. The following code segment shows how the global job queue is implemented.

```
00 class RayTracer {
01 int job; // the global job queue
02
03 public synchronized int getJob() {
04 return --job;
05 }
06
07 RayTracer(int w, int h, ...  ) {
08 // job is initialized to the height of the image
09 job = h;
10 //
11 ...
12 }
```

.

Fig 7.12: Implementation of `getJob()` in class `RayTracer`, notice that the value of `job` is initialized to the height of the image

```
00 class WorkerThread extends Thread {
01 public void run() {
02    int line; // line is the next scanline to trace
03    while((line = rayTracer.getJob()) >= 0) {
04    for (int i = 0; i < w; i++)
05          // to calculate the pixel value at coordinates (i, line)
06          rayTracer.rayTrace(i, line);
07    }
08 }
```

Fig 7.13: Main-loop of `WorkerThread` to compute pixels in the image

The `getJob()` method (Fig 7.12, line 03) returns the next line that the worker thread needs to work on. The value of job is initialized to be the height of the image (Fig 7.12, line 09), so that when the `getJob()` method is called by a worker thread the first time, the bottom line of pixels will be traced. After that, the lines are traced from the bottom towards the top, with the value of the `job` variable decremented by one every time the `getJob()` method is invoked. The `getJob()` method has to be defined as `synchronized` (Fig 7.12, line 03) because multiple worker threads may execute the `getJob()` method at the same time (Fig 7.13, line 03) to obtain the line number to work on. The `synchronized` keyword guarantees that at any time only one thread can enter the body of the method, any other threads trying to execute the method will have to wait until the current thread have left the method body. This arrangement can therefore prevent any race condition to occur during the update of the `job` variable and help to ensure the value to be consistent.

The synchronization mechanism of the `getJob()` method described above is supported by the Cooperative Semaphore that maintains the same thread synchronization semantics even after a thread is migrated to a remote node. Because this recursive ray-tracing program relies on the Cooperative Semaphore to implement the global job queue mechanism, operations provided by Cooperative Semaphore will be performed frequently as all the worker threads will execute the `getJob()` method for each line of pixels to compute, the program therefore serves as a good indicator on how significant distributed thread synchronization can affect the performance of JESSICA. Besides, as the 3D scene is stored in the DSM, the program can also demonstrate the performance of the DSM when the 3D scene is accessed as read-only.

The recursive ray-tracing program was tested on 1, 2, 4, 8 and 12 processors with 1, 2, 4, 8 and 12 worker threads created respectively. The 3D scene is a snowman composed of 2 light sources, 4 triangles, 5 spheres, and a 2D plane. In each case the program was asked to generate an image of dimension 480x640. The result is presented as follows:

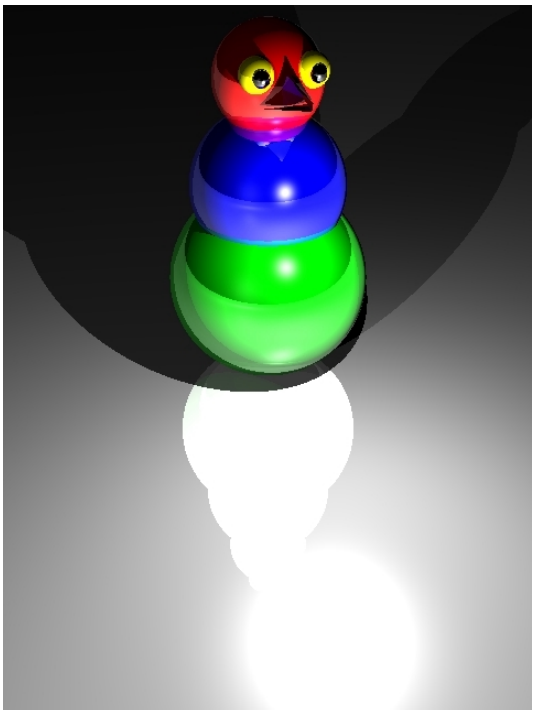

Fig 7.14: A snowman image produced by the recursive ray-tracer (480x640 pixels)

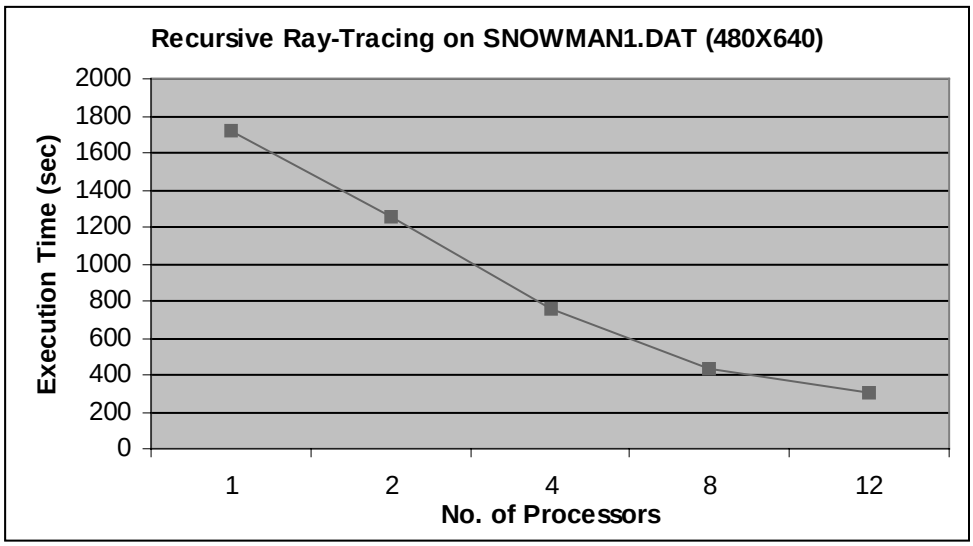

Fig 7.15: Total execution time against no. of processors

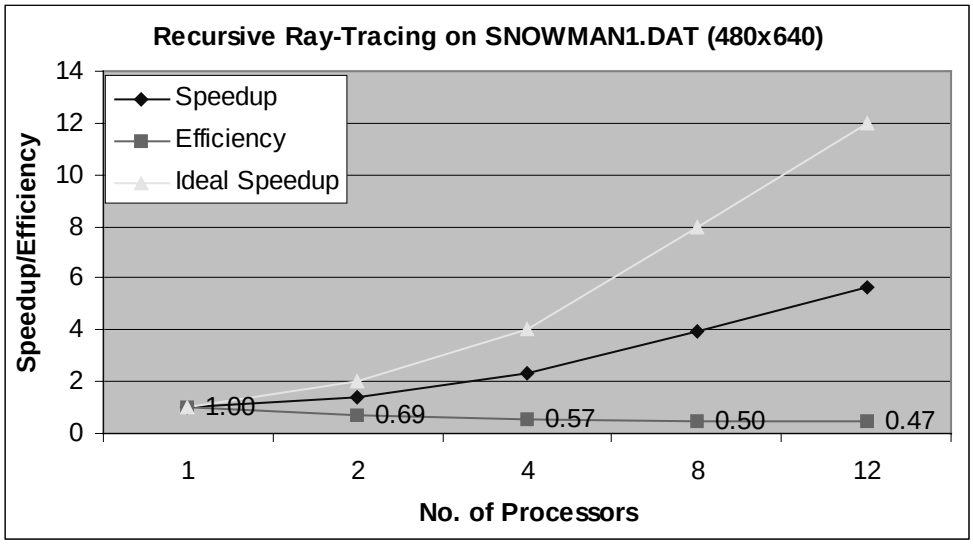

Fig 7.16: Speedup/efficiency against no. of processors

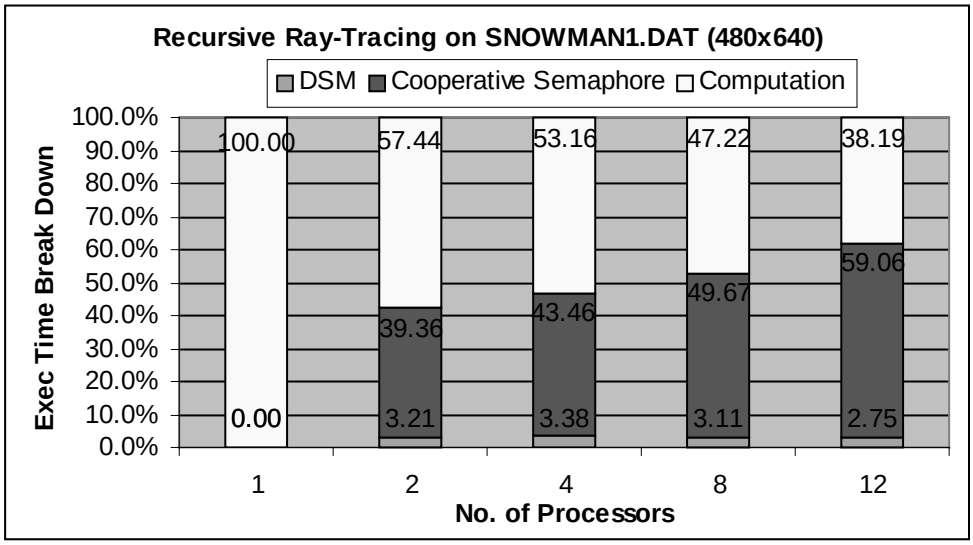

Fig 7.17: Percentage of execution time break down against no. of processors

From Fig 7.16 it can be found that the efficiency is less than optimal and it decreases moderately as more processors are used, the efficiency drops from 69% when using 2 processors to 47% when using 12 processors. This is because as indicated in Fig 7.17, the percentage of Cooperative Semaphore overhead contributes a significant amount to the total execution time. From the data collected it is found that altogether there were about 32,000 of Cooperative Semaphore acquire and release operations performed, and each one took on average 50 milliseconds to complete. When more processors are used, it is more likely that they are trying to acquire the Cooperative Semaphore at the same time when executing the

`getJob()` method, so more likely that they will be put to wait. By the same token, the console node is also likely to receiving more Cooperative Semaphore requests at a given time when more processors are used, therefore the response time for replying remote nodes will also increase. This is verified by the time to perform an acquire operation: the time increases from 27 milliseconds when using 2 processors to 84 milliseconds when using 12 processors. Therefore we conclude that the overhead of Cooperative Semaphore operations increases as more processors are used. On the other hand, the percentages of DSM overhead stay relatively constant when different number of processors are used. This is mainly because the 3D scene information are only accessed as read-only by all the threads. Hence once the data are cached locally, the access overhead does not change significantly when more processors are used.

7.4.3 Red-Black Successive Over-Relaxation on a Grid

The Red-Black Successive Over Relaxation (R/B-SOR) on a grid is another standard parallel application found in the literature. It is included as a programming example in the TreadMarks DSM package and we have adapted it as a multi-threaded Java Application. The application is used for solving partial differential equations. A huge matrix is created with the perimeter elements being initialized to be the boundary conditions of a given mathematical problem, each of the interior elements of the matrix is then computed as the average of its top, bottom, left, and right neighbors. The interior elements are repeatedly computed in the described manner until the computed values are sufficient close to the values computed in the last iteration. When the interior elements are successfully set to their neighbors' average, they satisfy a simple approximation to the 2D LaPlace equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \qquad \text{(Eq. 2)}$$

though an interpretation of the above equation is beyond the scope of this chapter.

In the R/B-SOR implementation, the huge matrix is divided into two sub-matrices, the Red matrix and the Black matrix. At any time the elements are read from one sub-matrix and the computed averages are written to the other. After finishing all the elements in the sub-matrix, their roles are swapped and the process is repeated. The R/B-SOR program creates multiple threads to compute the matrix elements in parallel. The Red and Black matrices are divided into roughly equal size bands of rows, with each band assigned to a different threads. The threads are synchronized using barrier every time before the Red and the Black matrices swap

their roles. The following code segment shows the implementation of barrier synchronization between threads and the main execution loop of each worker thread.

```
00 class SorWorkerThread extends Thread {
01 static int no_of_threads = 0;
02 static Object barrierObj;
03 static int barrierCount;
04
05 static {
06    barrierObj = new Object();
07    barrierCount = -1;
08 }
09
10 public void barrier() {
11   try {
12    synchronized(barrierObj) {
13          if (barrierCount == -1)
14                barrierCount = no_of_threads;
15          if (--barrierCount > 0)
16                barrierObj.wait();
17          else {
18                barrierCount = -1;
19                barrierObj.notifyAll();
20          }
21    }
22  }
23 }
```

Fig 7.18: Implementation of barrier synchronization between threads

```
00 private void sor_first_row_odd(int first_row, int end) {
01     int  i, j, k;
02     for (i = 0; i < iterations; i++) {
03       for (j = first_row; j <= end; j++) {
04          for (k = 0; k < N; k++) {
05            black[j][k] = (red[j-1][k] + red[j+1][k] + red[j][k]
06                        + red[j][k+1])/(float) 4.0;
07          }
08          if ((j += 1) > end)
09            break;
10          for (k = 1; k <= N; k++) {
11            black[j][k] = (red[j-1][k] + red[j+1][k] + red[j][k-1]
12                        + red[j][k])/(float) 4.0;
13          }
14       }
15       barrier();
16       for (j = first_row; j <= end; j++) {
17          for (k = 1; k <= N; k++) {
18            red[j][k] = (black[j-1][k] + black[j+1][k] +
19                black[j][k-1] + black[j][k])/(float) 4.0;
20          }
21          if ((j += 1) > end)
22            break;
23          for (k = 0; k < N; k++) {
24            red[j][k] = (black[j-1][k] + black[j+1][k] +
25                black[j][k] + black[j][k+1])/(float) 4.0;
26          }
27       }
28       barrier();
29     }
30 }
```

Fig 7.19: The main execution loop of a worker thread

The purpose of `barrierCount` variable in (Fig 7.18, line 03) is to remember the number of threads that have not yet entered the barrier. Therefore when its value is decremented to zero, the current thread is the last thread to enter the barrier and hence all the waiting threads can be awaken (Fig 7.18, line 19). On the other hand, if the current thread is the first thread that enters the barrier, the `barrierCount` variable is set to be number of worker threads in the system (Fig 7.18, line 14). The value of `no_of_threads` remembers how many worker threads there are in the system. Its value is incremented by one whenever a new worker thread is created and decremented by one when a worker thread terminates.

According to the code shown in Fig 7.19, the main loop of each thread repeatedly retrieves values from one matrix, computes the average, and writes the result back to another matrix. Since matrices are complex data type, their space is allocated from the DSM instead of the local method stack. Hence the execution will impose a significant amount of loading onto the DSM system, and the application is therefore a good candidate for studying how the DSM

overhead contributes to the parallel execution performance of JESSICA. Another point to note is that for simplicity, our implementation only iterates for a given number of times (Fig 7.19, line 02), instead of keep iterating until the newly computed average is sufficiently close to the last iteration by a given epsilon. Since a barrier-synchronization is executed before the Red and the Black matrix swap their role (Fig 7.19, line 15 and line 28), barrier synchronization is therefore performed twice per iteration. As a result, total number of barrier synchronization performed is equal to two times the number of iterations. The number is independent from the number of threads deployed and the matrix size.

The R/B-SOR program was again tested on 1, 2, 4, 8 and 12 processors with 1, 2, 4, 8 and 12 worker threads created respectively. The size of the matrix we used was 1024x1024 and the perimeter elements were initialized to alternate zeros and ones. We have performed 10 iterations for each run and the result is presented as follows:

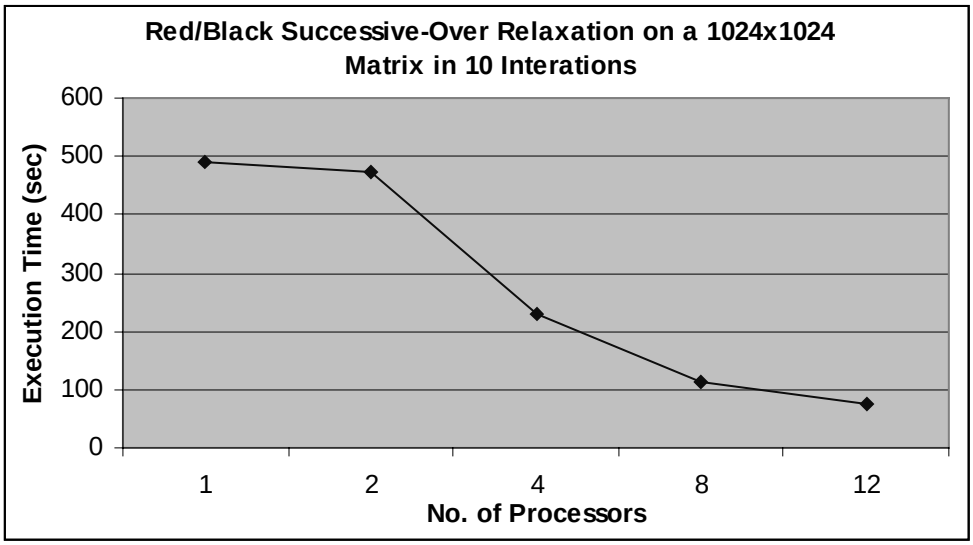

Fig 7.20: Total execution time against no. of processors

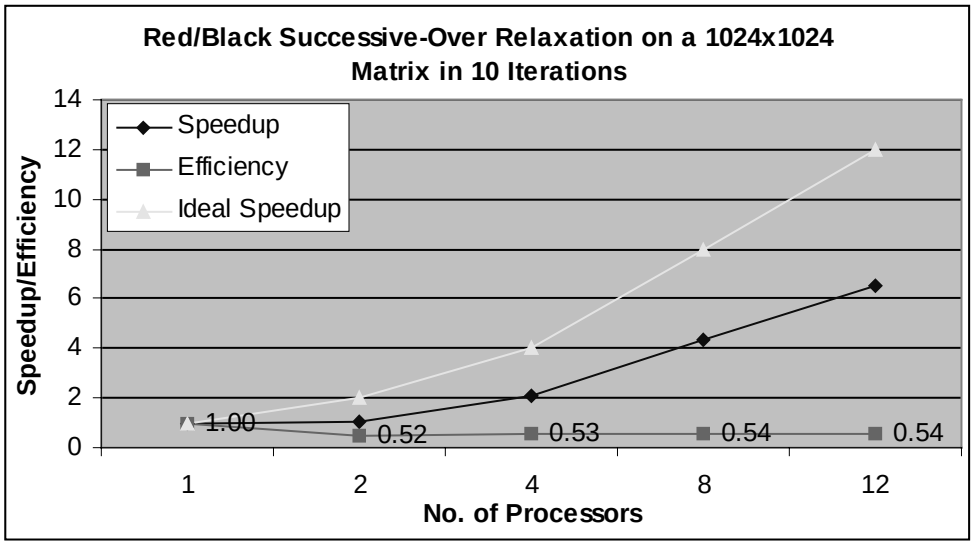

Fig 7.21: Speedup/efficiency against no. of processors

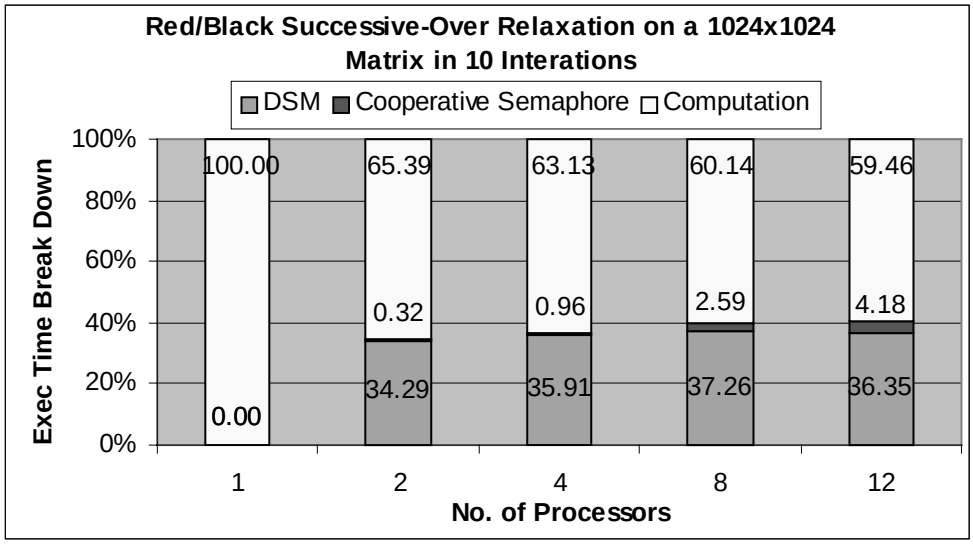

Fig 7.22: Percentage of execution time break down against no. of processors

The graph of Fig 7.21 shows that the application can gain moderate speedup when running with 4 or more processors. When using two processors only, 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 execution time (Fig 7.22). Since the barrier synchronization is implemented using the `synchronized` keyword, the `wait()`, and the `notifyAll()` methods,

which are in turn implemented using Cooperative Semaphore, overheads from Cooperative Semaphore thus also appears in the graph of Execution Time Break Down (Fig 7.22). As explained in the previous paragraphs, the number of barrier synchronization is equal to twice the number of iterations, therefore there are only 20 barrier-synchronization performed during the execution of the program. The total time taken to perform the 20 barriers only constitute a relatively small percentage to the total execution time. Finally, we conclude that when the running threads are sharing a huge amount of data, the overhead due to the DSM can be significant (about 36% in our example), though the relative percentage decreases as more processors are used (Fig 7.22).

7.5 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 7.2 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.

7.5.1 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 7.23.

```
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    }
```

Fig 7.23: Class `Bar` that can minimize the number of distributed object accesses when migrated

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 7.23, 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 7.23, 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 elements directly into the `sum` variable every time the `for` loop iterates, the result is stored in a local `temp_sum` variable first (Fig 7.23, line 10). The final value is only written to the sum variable when the for loop exit (Fig 7.23, 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 7.23, the local `myArray` variable is used to cached the reference to the object's array (Fig 7.23, line 06), so that its elements can be immediately accessed in the `for` loop (Fig 7.23, line 10).

7.5.2 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 7.4.2. In general, there is no perfect solution to tackle the granularity problem, usually careful tuning is required to find the optimal size.

Chapter 8

Conclusions and Future Work

8.1 To Migrate Or Not To Migrate

JESSICA takes a novel approach in dealing with thread migration when compared to traditional approaches like Millipede [18] and MOSIX [2]. Instead of moving the whole execution context to the destination all at one time, the context is separated into machine dependent and machine independent parts, and only the machine independent states are transported in a regulated manner. This design imposes no limitation on the type of thread that can migrate, whether they own location-dependent resources or not. Moreover, the design also prepares substantial ground for further development of thread migration in a heterogeneous environment as all the states migrated are hardware independent. Hence, the JESSICA approach provides maximum flexibility, portability, and transparency for thread migration. This JESSICA advantage is a direct consequence of using the Java Programming Language, its virtual machine approach provides an extra layer of abstraction. The approach allows the implementation of JESSICA to be carried out entirely at the user level, without having to deal with any operating system specific or hardware specific issues. On the other hand, in the traditional cases like Millipede and MOSIX, their operating systems are usually customized in order to support migration. For JESSICA, the basic requirement is to maintain the same interface and execution semantics that are visible to Java applications as defined by the Java Virtual Machine Specification [36]. On account of this unique advantage, we strongly believe implementing thread migration in JESSICA is highly feasible and beneficial. It is verified by our functional prototype that has demonstrated improved performance and considerable speedup for multi-threaded applications.

8.2 The Home Model

JESSICA follows the home model to support the required SSI transparency, any location-dependent operations are redirected back to the console node. Although this approach offers simplicity and modularity for implementing the transparency, some performance is sacrificed. It is because the current implementation of message redirections is built on top of the signal handling and the socket I/O mechanisms provided by the underlying UNIX operating system. To perform a location-dependent operation, it will involve the invocations of multiple system

calls like <setjmp()>, <longjmp()>, <select()>, <read()>, and <write()>, these system calls as well as the UNIX signal handling overhead are by no means lightweight as they cause many context-switchings between the kernel and the JESSICA process. In addition, it can overload the console node when the migrated threads try to send too many redirection requests back to the console at the same time. This is shown in our recursive ray-tracing experiment in section 7.4.2 in which threads are tightly synchronized, the time to perform distributed thread synchronization increases sharply when more migrated threads are introduced into the system. However, this is the tradeoff we have opted when implementing JESSICA. We are willing to tolerate some decrease in performance in order to achieve a high degree of transparency, better modularity and simplicity.

8.3 An Effective DSM Subsystem

From our implementation experience with JESSICA, an efficient and reliable distributed shared memory (DSM) system is one of the most important components that can determine the success of the JESSICA approach. With the help of a DSM, we are able to move threads around the cluster freely without worrying if any object references will become invalid for the DSM will transparently handle the migration of memory pages when a migrated thread tries to access any remote objects. This can simplify the implementation of JESSICA substantially. However, DSM overhead can account for about 35% of the total execution time and the DSM access penalty can be about 9 times higher than that for local memory access. Therefore, an effective DSM subsystem is indispensable.

In the JESSICA implementation, we are using a commercial package called Treadmarks [1] for DSM support. The package is based on the release consistency model. In this model, any updates made to the shared memory are considered possible to produce race conditions and therefore they have to be protected by using the semaphore acquire and release operations provided by the DSM subsystem. When it is deployed in JESSICA, every DSM update performed by the application has to be protected by the semaphore operations because the runtime system cannot anticipate when and where the next memory location that the running application is going to access. As a result, many semaphore operations, sometimes redundant, are being executed and they will account for the significant portion that DSM overhead is contributing to the execution time. Another setback is the amount of DSM semaphore operations being executed will sometimes exceed the DSM subsystem limit. For example, if a thread running on one of the computers is making more DSM updates than other threads running on other computers, as in the case when the thread is following a different execution flow or if its computer is simply more powerful, the faster thread will cause the DSM

subsystem to generate many update messages that are sent to the other computers so as to maintain the DSM consistency. The DSM messages so generated can reach a size that exceeds the maximum message size or the amount of messages exceeds the DSM limit, this will lead to the malfunction of the DSM subsystem. In the end, the overall reliability of the system is compromised. Nevertheless, Treadmarks is a commercial package that and we do not have any access to its source code, there is little control in this respect.

The ideal DSM subsystem should be able to maintain consistency by itself without too much attention from the JESSICA runtime. It should also be able to determine the optimal strategy for data replications and update propagation. It would be the most favorable if hardware DSM can be available as part of the built-in feature in a processor similar to that of the virtual memory, this would greatly facilitate the implementation of systems like JESSICA that supports Single-System-Image.

8.4 User level Thread System

The internal thread system in the JESSICA prototype is implemented at the user level, and is based on the `<setjmp()>/<longjmp()>` system calls of the UNIX operating system. There are both advantages and disadvantages with this user-level approach when compared to that using kernel thread. On one hand, special care has to be taken in order to keep the JESSICA process alive with user level threads. This is because if a user level thread is blocked by a blocking system call such as the network I/O `<read()>`, the whole JESSICA process will also be blocked, and this will prevent other ready threads from being scheduled to run. In order to keep the process alive, extra checking has to be done to see if a thread will block before it tries to execute any blockable system calls. For example, when a thread tries to read from a socket, the system has to first execute the `<select()>` system call to see if data is ready at the socket. If the system finds the socket not ready, it will schedule out the current thread and let other ready threads to run. When data arrives at the socket later, a SIGIO signal will be generated and this will cause the system to move the de-scheduled thread back to the ready queue. As a result, the required checking and signal handling will introduce extra overhead into the JESSICA runtime system. On the other hand, user level threads offer better portability and makes JESSICA runnable on systems where kernel thread is not available. Moreover, the context-switching overhead of user level threads is about an order of magnitude smaller than that of kernel threads [33].

8.5 Transparent I/O Redirection

Because I/O operations are location-dependent, they are redirected back to the console when a migrated thread tries to perform such operations in order to maintain the SSI transparency as discussed in section 6.5. The interfaces to file and network I/O are provided by the `java.io` and the `java.net` class libraries respectively. SUN provides the Java implementation of these two class libraries where applications that need file or network I/O support can link to them directly. Methods in the two libraries will eventually invoke the corresponding native methods provided by a JVM to perform the real work. In other words, a JVM is free to implement the native methods that perform the low-level file and network I/O operations, as long as the interfaces exporting to the SUN's `java.io` and `java.net` class libraries are not changed.

Notice that when a thread is migrated, any subsequent I/O operations will be forwarded back to the console in order to maintain the migration transparency. However, because of the significant message redirection overhead incurred between the master and the slave thread as discussed in Chapter 7, remote I/O operations can be a source of inefficiency especially if the migrated thread performs remote I/O operations frequently with very short messages.

A possible solution to reduce this inefficiency is for the migrated thread to cache the bytes to be written in a buffer temporarily. A write redirection request will not be sent to the console until the buffer is full, or after a certain amount of time has passed so that the data will not be cached forever at the worker node. At this time all the data that is cached in the buffer will be shipped together to the console by a single operation. This technique is similar to the buffering of I/O data in the UNIX kernel.

8.6 Conclusions

Our implementation experience with the JESSICA prototype and the experiment results we obtained 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 the design and implementation of JESSICA. On the other hand, it is the characteristics of the language; namely, the bytecode execution, the pointer-less semantics, the simple model of inter-thread signaling and synchronization and any additional system services are offered as modular class libraries, that have simplified the implementation of JESSICA. Consequently, we are able to arrive at a design that does not introduce any changes to the Java Programming Language and still be able to support thread migration and to achieve the required SSI transparency. The only element that needs to be changed in order to facilitate thread migration is the implementation of the `java.io` and the `java.net` package so that it is possible to implement I/O redirection more efficiently.

The idea of execution by means of a virtual machine also provides the simplicity for implementing the system as a user-level process on top of the operating system. As a result, the implementation does not require dealing with any lower-level implementation of the operating system as it can be tedious. On the other hand, running on top of the operating system as a user process makes the JESSICA runtime system less efficient since JESSICA has to rely on system calls in order to provide the required system services to applications such as thread scheduling and I/O operations. The ideal case is to be able to by-pass the operating system or to implement JESSICA as the sole operating system running on the hardware. Following the microkernel approach there can be a JESSICA kernel running on each node in the cluster and it is only responsible for performing thread scheduling and message redirections between the kernels. The Global Object Space is supported by DSM servers running at the user level across the cluster. Similarly, addition system services such as file I/O and graphical window system are also supported by user level servers.

The DSM subsystem that the JESSICA prototype is using does not support dynamic addition or removal of nodes to and from the cluster. Consequently, the Global Object Space cannot expand when more nodes are attached to the cluster or contract when some nodes in the cluster fail. To support a dynamically reconfigurable JESSICA cluster, future DSM systems should allow dynamic addition/removal of participating nodes.

We believe our work has laid a concrete foundation for future research in the JESSICA direction. It is a step forward towards the ultimate goal of Global Computing, where in the future every computer will be connected by the Internet, the physical boundaries between computers are blurred and they can organize themselves together to function as single, gigantic, super-servers with dedicated functionality.

The JESSICA idea was originally sparked by the story of the RSA-129 Challenge. We would therefore like to present the incidence here as a compliment:

> *"The RSA-129 challenge was made by Gardner in 1977, where Rivest, Shimar, and Adleman, the inventors of the RSA cryptosystem [31] offered 100 US Dollars to anyone who can decode a given encrypted message. A 129-digit RSA key, randomly picked by Rivest encoded the secret message. At that time Rivest estimated the decryption process would take a dedicated supercomputer to run for 40 quadrillion years! However, it turned out that it took merely 8 months to crack the message. In 1994, 17 years after the announcement of the challenge, Atkins, an electrical engineering student at MIT, with the help from about 600 volunteers from more than 20 countries around the world, solved the problem using more than 1,600 machines. The computers they used ranged from PC compatibles to a 16,000-processor MasPar supercomputer. The sieving step in the process took approximately 5,000 MIPS years, but was done in just 8 months by distributing the computation over the 1,600 computers involved. A MIPS year is the work done by a million-instructions-per-second machine running for a solid year. The successful crackers later donated the 100-dollar reward to the Free Software Foundation [32, 4]."*

This story has not only revealed how powerful a network of computers can become when they work together in a coordinated manner, but it has also proved the goal of Global Computing is a certainty.

To conclude, the RSA-129 Challenge as well as our proposed JESSICA system, have both fulfilled the famous credo of the SUN Microsystems Inc [35], which says – "The Network is the Computer™".

Recently it has been brought to our attention that the Microsoft's Millennium project [40] has also developed a prototype called Borg [27]. Borg is a distributed Java virtual machine that makes a cluster of computers appears as a single large computer for Java application. The Borg prototype is very similar to our JESSICA as they both follow the SSI approach.

Although we are not able to obtain any further information on the Borg prototype from Microsoft, our JESSICA belief is further reinforced by the news.

8.7 Executive Summary

This thesis presents our proposed JESSICA system which can turn a cluster of computers into an instant supercomputer. JESSICA runs as a middle-ware on top of the operating system that in turn runs on each computer. It encapsulates the cluster as a single system with multiple processors, one single and contiguous memory space and other unified resources.

JESSICA provides a *Global Thread Space* that hides the physical boundaries between machines; threads can freely move around the cluster to bind to less-loaded processors for execution. Thread migration is supported by our novel technique called *Delta Execution* where a thread can be preempted at the end of any bytecode instructions and be migrated to any other node in the cluster. The machine dependent and the machine independent execution states of the migrating thread are identified and isolated. Only the machine independent states called *Delta Sets* are migrated and executed on a remote node, all the machine dependent executions are still performed back at the home node. As a result, no machine dependent state information is transported across machines and hence we are able to arrive at a clean and portable implementation. The implementation does not need to consider any low-level information that is hardware specific, consequently, it can be done entirely in the high-level C/C++ language.

In addition, our *Master/Slave Design* for migrated thread further facilitates the provision of transparent thread migration. In practice, a thread does not actually move to another machine when migrating, but a new thread is created anew at the destination machine to act as the migrated image; that is, the slave of the original migrating thread. The original thread at the home node is transformed into a master where it performs any location-dependent operations on behalf of its migrated slave and redirects any messages to and from its slave. Moreover, the master/slave design enables the correct implementation of *Cooperative Semaphore*, which guards against any inconsistent accesses to critical sections where multiple threads are possible to update the same shared data at the same time. Since the Cooperative Semaphore maintains the same semaphore semantics even after threads are migrated, threads can synchronize themselves in a distributed fashion even if they are running on different nodes of the cluster. Consequently, transparent thread migration can be achieved because other threads in the system can never be aware of the fact that a thread has been moved to run on another node, as all the interactions between the threads are still performed at the home node.

Furthermore, with the help of a distributed memory system (DSM), allocated objects remain to be accessible by threads independent of their physical locations. This is necessary because a fundamental characteristic of multi-thread programming is that threads do share data between them. The DSM supports a *Global Object Space* where threads can share objects even after they have migrated to different nodes on the cluster. In conclusion, it is the *Global Object Space*, the *Cooperative Semaphore*, the *Master/Slave Design* for message redirections and the *Delta Execution* mechanism together that hide the physical boundaries between machines and make the *Global Thread Space* realizable.

With the Global Thread Space established, JESSICA offers a parallel execution environment for multi-threaded Java applications. The mapping between threads to processors can be automatic because JESSICA supports the Serial-Program-Parallel-Subsystem (SPPS) computing paradigm. Application programmers do not need to worry about the number of processors available but instead they can create as many threads as needed. The system can automatically migrate threads to any idle processors and maximize the parallelism.

As JESSICA is compatible to the standard Java Virtual Machine, the vast number of existing Java applications can run on the system immediately and gain speedup. Furthermore, application programmers can now use the shared memory model with thread support for developing parallel applications, which is considered simpler and less tedious to use than the message-passing model. As a result, it is possible for JESSICA to turn the Java Programming Language into the most favorable language for parallel application development in the future.

To prove our concepts, we have implemented a JESSICA prototype that runs on a cluster of 12 SUN Ultra 1 machines interconnected by a 155Mbps ATM switch. Experiments show that Delta Execution is able to provide fast Java thread migration between machines. The minimum migration latency is about 28 milliseconds. The shuttling mechanism that allows machine dependent operations to be performed back at the source node is also proved to be reliable and effective. In addition, the Master/Slave design for message redirections and the Cooperative Semaphore can guarantee the correct interactions between threads when a number of them have migrated to other nodes; together with the Global Object Space supported by the DSM, they can successfully uphold the SSI and migration transparency requirement in JESSICA. Consequently, the JESSICA prototype offers a functional execution environment for parallel execution of multi-threaded applications. We have devised several applications that stress on the performance of various migration-related components of the system. Experiments show we are able to obtain considerable speedup in all of them. For the integration application that requires minimal communication and

coordination between threads, we are able to obtain close-to-optimal efficiency of over 93% when running on 12 nodes. For the recursive ray-tracing application whose threads are tightly synchronized, the efficiency ranges from 69% to 47% when running from 2 to 12 nodes. Finally, for the Red/Black Successive-Over-Relaxation application that shares huge amount of data between threads, the efficiency stays relatively constant at around 54% when different number of nodes are used.

8.8 Future Work

The following are possible areas of research that we have identified for the future development of JESSICA.

8.8.1 Heterogeneous Migration

Extending the system to support heterogeneous migration can be the next immediate step for the JESSICA development. The design and implementation of the current prototype has already made way for heterogeneous migration as the implementation of Delta Execution only requires the shipping of machine independent execution context across machines. In addition, all the interactions and messages redirected between the computers in the cluster are implemented on top of the TCP/IP protocol, there is no assumption made on any specific hardware architecture. What is missing here in the current version for supporting heterogeneous migration is a heterogeneous DSM subsystem which allows threads to share objects residing on different hardware. Although the current release of the Treadmarks DSM package we are using is a homogeneous one, the next version is planned to support heterogeneous DSM on machines with the same byte-ordering. When a heterogeneous DSM package like the next planned release of Treadmarks is available, we should be able to proceed with heterogeneous migration in JESSICA.

8.8.2 Thread Migration with JIT Execution Support

The current version of JESSICA only supports thread migration when the Bytecode Execution Engine (BEE) is executing under the interpreter mode. Under this mode the BEE interprets each bytecode instruction and updates the execution context of the current thread. This allows the execution context of a thread to be readily extractable at any time and the states are represented in a machine independent manner to facilitate thread migration. A problem with this design is that the execution speed is compromised. The execution speed can be much faster when the BEE is executing under the Just-in-Time (JIT) mode where bytecode are first complied into native machine instructions before executing. However, this would imply the execution states of a thread is now encoded in machine dependent values,

and makes thread migration difficult. A possible solution to this problem is checkpointing. A challenge in this approach is how to insert checkpoints into the Just-in-Time-compiled native code so that the execution states at each checkpoint can map to an equivalent execution states when executing under the interpreter mode, the states can then be represented in a machine independent manner.

8.8.3 A Java Shell

The current version of JESSICA resembles the standard Java Virtual Machine that only supports the execution of one Java application at a time. To achieve optimal resource utilization in the cluster, the JESSCIA system should allow multiple applications to execute at the same time, like a multi-computer. Hence, a Java shell utility with similar functionality provided by common UNIX shells is necessary for initiating applications in the JESSICA cluster. A research challenge for this is how to define and implement a protection domain for each of the Java applications that is co-existing in the JESSICA runtime system. Threads belonging to different application instances should not interfere with one another. In addition, the objects created by threads in one application instance should not be accessible by threads running within other application instances. The system should be able to detect and prevent any illegal accesses that are made across the protection domains.

8.8.4 A Distributed Java OS in the Micro-kernel Approach

As we have discussed in the background study in chapter 2, the micro-kernel approach is more flexible and effective for implementing process and thread migration. JESSICA can be implemented at the micro-kernel level as a distributed Java OS, whose performance can then be improved when compared to the current implementation as it executes as user processes on top of the standard UNIX operating system. Experiments show the major overhead in the current implementation comes from the DSM subsystem and the synchronization of distributed threads, both of them rely heavily on sending messages between computers to perform their tasks. Therefore when following the distributed operating system approach where there is a micro-kernel running on each node, the communications performed between the nodes can be carried out at the kernel level. In addition, the DSM subsystem can be implemented in the form of an external pager as in the case of Mach NORMA [7], which offers transparency, flexibility and modularity. The Flux OS Toolkit [13] can be a good starting point for this approach.

References

[1] 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, Vol. 29, No. 2, pp. 18-28, Feb 1996.

[2] Barak, O. Laden, Y. Yarom, 'The NOW MOSIX and its Preemptive Process Migration Scheme', Institute of Computer Science, The Hebrew University, Israel.

[3] Chan, Lee, Kramer, 'The Java Class Libraries', 2nd edition, Addison Wesley, 1998.

[4] D. Atkins, 'Factor RSA-129', http://www.mit.edu:8001/people/warlord/RSA129-announce.txt.

[5] D. Lea, 'Concurrent Programming In Java: Design Principles and Patterns', Addison Wesley, 1997.

[6] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, 'Process Migration, Technical Report', TOG Research Institute, Oct 1996.

[7] D.S. Milojicic et al., 'Task Migration on Top of the Mach Microkernel', In 3rd USENIX Mach Symposium, pp. 273-289, Santa Fe, Apr 1993.

[8] D.I. Bevan, 'Distributed Garbage Collection using Reference Counting,' PARLE Parallel Architectures and Language Europe, pp. 176-187, Springer-Verlag LNCS 259, June 1987.

[9] Dimitrov and Rego, 'Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms', IEEE Transactions on Parallel and Distributed Systems Vol. 9, No. 5, pp 459-469, May 1998.

[10] E. Jul, H. Levy, N. Hutchinson, and A. Black, 'Fine Grained Mobility in the Emerald system', ACM Transcations on Computer Systems, 6(1):109-133, Feb 1988.

[11] E. Sahin, 'RayWizard', Department of Computer Science, Simon Fraser University, http://www.cs.sfu.ca/~esahin/personal/java/RayWizard.

[12] F. Douglis, J. Ousterhout, 'Transparent Process Migration: Design Alternatives and the Sprite Implementation', Software Practice and Experience, vol. 21(8), Aug 1991.

[13] Ford, G. Back, G. Benson, J. Lepreau, A. Lin, O. Shivers, 'The Flux OSKit: A Substrate for Kernel and Language Research', Proceedings of the 16th ACM Symposium on Operating Systems Principles, Oct 1997.

[14] G.F. Pfister, 'In Search of Clusters: the Coming Battle in Lowly Parallel Computing', Prentice Hall PTR, 1995.

[15] General Magic, 'Introduction to the Odyssey API', http://www.genmagic.com/agents/odysseyIntro.ps.

[16] Gosling, B. Joy, G Steele, ‘The Java Programming Language Specification’, Addison Wesley, 1996.

[17] Grimshaw, W. Wulf, and the Legion Team, ‘The Legion Vision of a Worldwide Virtual Computer’, Communications of the ACM, 40(1), Jan 1997.

[18] Itzkovitz, A. Schuster, and L. Shalev, ‘Thread Migration and its Applications in Distributed Shared Memory Systems’, The Journal of Systems and Software, vol 42(1), pp. 71--87, Jul 1998.

[19] J. White, ‘Telescript Technology: An Introduction to the Language, General Magic White Paper’, General Magic, 1995.

[20] Javasoft, ‘Java Interface Definition Language’, http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html.

[21] Javasoft, ‘Java Object Serialization’, http://java.sun.com/products/jdk/1.1/docs/guide/serialization/index.html.

[22] Javasoft, ‘Java Remote Method Invocation - Distributed Computing for Java, a White Paper’, http://java.sun.com/marketing/collateral/javarmi.html.

[23] M. Deshpande et al., ‘Application of a Distributed Network in Computational Fluid Dynamic Simulation’, Supercomputing Application High Performance Computing, 8, (1), 64-67 (1994).

[24] M. Lee, A. Tam, and C.L. Wang, ‘Directed Point: An Efficient Communication Subsystem for Cluster Computing’, The International Conference on Parallel and Distributed Computing Systems (IASTED), Oct 1998.

[25] M. Livny. ‘The Condor Distributed Processing System’, Dr. Dobbs Journal, pp. 40-58, Feb 1995.

[26] M.M. Theimer, K.A. Lantz, and D.R. Cheriton, ‘Preemptable Remote Execution Facilities for the V-System’, In Proc. 10th ACM Symp. On Operating System Principles, Dec 1985.

[27] Microsoft Research, ‘Borg, a Prototype for the Millennium Project’, http://research.microsoft.com/sn/Millennium/Borg.html.

[28] P. Smith, N. Hutchison, ‘Heterogeneous Process Migration: The Tui System’, Technical Report, 96-04 (Revised), University of British Columbia, Mar 1997.

[29] R. Fatoohi and S. Weeratunga, ‘Performance Evaluation of Three Distributed Computing Environments for Scientific Applications’, Supercomputing ‘94, pp.400-408.

[30] R.E. Ewing et al.,’Distributed Computation of Wave Propagation Models Using PVM’, IEEE Parallel Distributed Technology, 2, (1), 26-31 (1994).

[31] RSA Data Security Inc., ‘High-Speed RSA Implementation’, TR 201, ftp://ftp.rsa.com/pub/ps/201.ps.

[32] S. Levy, Wisecrackers, 'Wired Magazine', 4.03, Mar 1996, http://www.wired.com/wired/archive/4.03/crackers_pr.html.

[33] S. Tanebaum, 'Distributed Operating System', Prentice Hall, 1995.

[34] S.J. Mullender, G. van Rossum, R. van Rensse, and H. van Staveren, 'Amoeba - a Distributed Operating System for the 1990s', IEEE Computer, 23(5):44-53, May 1990.

[35] Sun Microsystems Inc., 'A Timeline of Sun's History', http://www.sun.com/corporateoverview/who/html_history.html.

[36] T. Lindholm, F. Yellin, 'The Java Virtual Machine Specification', Addison Wesley, 1996.

[37] Transvirtual Technologies Inc, 'Kaffe Open VM', http://www.transvirtual.com.

[38] W. Gropp, E. Lusk, and A. Skjellum, 'Using MPI', MIT Press, 1994.

[39] W. Yu, A. Cox, 'Java/DSM: A Platform for Heterogeneous Computing', Department of Computer Science, Rice University.

[40] W.J. Bolosky, R.P. Draves, R.P. Fitzgerald, C.W. Fraser, M.B. Jones, T.B. Knoblock, R.Rashid, 'Operating System Directions for the Next Millennium', Microsoft Research, http://research.microsoft.com/research/os/Millennium/mgoals.html.

[41] W.T.C. Kramer et al., 'Clustered Workstations and Their Potential Role as High Speed Compute Processors', RNS-94-003, NASA Ames Research Centre, 1994.

[42] Y.A. Khalidi, J.M. Bernabeu, V. Matena, K. Shiriff, M. Thadani, 'Solaris MC: A Multi-Computer OS', Proceedings of the USENIX 1996 Annual Technical Conference, pp. 191-294.

[43] K. Hwang, H. Jin, E. Chow, C.L. Wang, and Z. Xu; 'Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space,' IEEE Concurrency Magazine, Vol. 7, No. 1, pp. 60-69, Jan-Mar., 1999.

[44] Zukowski, 'Prepare Yourself for What's New and Different in the Forthcoming JDK 1.2 Release', Javaworld, Nov 1998, http://www.javaworld.com/javaworld/jw-11-1998/jw-11-jdk12.html.