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

Matchy J M Ma, Cho-Li Wang, Francis C M Lau, Zhiwei Xu
Department of Computer Science and Information System
The University of Hong Kong
Pokfulam, Hong Kong
{jmma, clwang, fcmlau, zwxu}@csis.hku.hk

Abstract
JESSICA stands for 'Java-Enabled Single-System-Image Computing Architecture'. It is a middle-ware running on top of the standard UNIX operating system that makes a cluster of computers to appear as a single, multi-processor machine to Java 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. Location transparency is upheld by redirecting any location dependent operations to the right machines for execution. The result is a parallel execution environment where threads can freely move across machine boundaries and execute simultaneously. A working prototype is implemented on a 12-node cluster and considerable speedup can be achieved on all the experimental applications devised, which stress on the performance of various aspects of the implementation.

1 Introduction

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 where the computers are running integration software for performing collaborative computation. Many promising results have been reported on using clusters of computers for load sharing and parallel computing [13, 18]. With the advent of high-speed networking and microprocessor technologies, Cluster Computing has emerged as a favorable alternative to Massively Parallel Machines for high performance and large-scale computing.

In 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. Consequently, 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 [16].

The Java programming language [9] 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. We envision Java to become the language of choice for most kinds of application development.

In order to span across multiple machines and achieve true parallelism, Java programmers currently have to tackle the coordination between processing nodes at the application level, through some IPC mechanisms such as socket. Since the introduction of JDK version 1.1, the burden on the programmers is alleviated by the provision of Object Serialization [11], Remote Method Invocation (RMI) [12] and the Object Request Broker (ORB) [10] supports. They allow coordination and cooperation of processes at the function call level, through some remote procedure call (RPC) like mechanisms as illustrated in Fig 1a. Nonetheless, programmers still have to worry about the availability of the processing nodes involved, as the usability of a distributed application depends on the nodes' availability. Moreover, parallel programming in this paradigm is still not as straightforward as one could do when using a multi-threaded model.

This paper gives an overview on JESSICA, our solution to the issues mentioned above by providing a Single-System-Image (SSI). JESSICA stands for 'Java-Enabled Single-System-Image Computing Architecture', a platform for executing multi-threaded Java applications [7] 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 and provides applications with an illusion that they are running on a single multi-processor computer. Because JESSICA conforms to the standard Java Virtual Machine Specification [14], the vast number of existing Java applications are readily runnable on JESSICA without any modification and be able to gain speedup.

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, as shown in Fig 1b. The Global Thread Space hides the physical boundaries between machines, Java threads can then move freely around the cluster from one machine to another. This movement is supported by a preemptive thread migration mechanism called Delta Execution. Migration is transparent to threads residing in the Global Thread Space because any location dependent operations are forwarded to the right machine for execution. Moreover, JESSICA provides a single and contiguous Global Object Space that allows memory objects to remain accessible by threads independent of their physical locations, even if the threads have been migrated to other machines. JESSICA supports the Global Object Space by deploying a Distributed Shared Memory (DSM) system over the cluster, so that memory consistency against concurrent accesses can be ensured. Consequently, the free movement of threads in the Global Thread Space provides an opportunity for optimizing utilization of shared resources in the cluster.

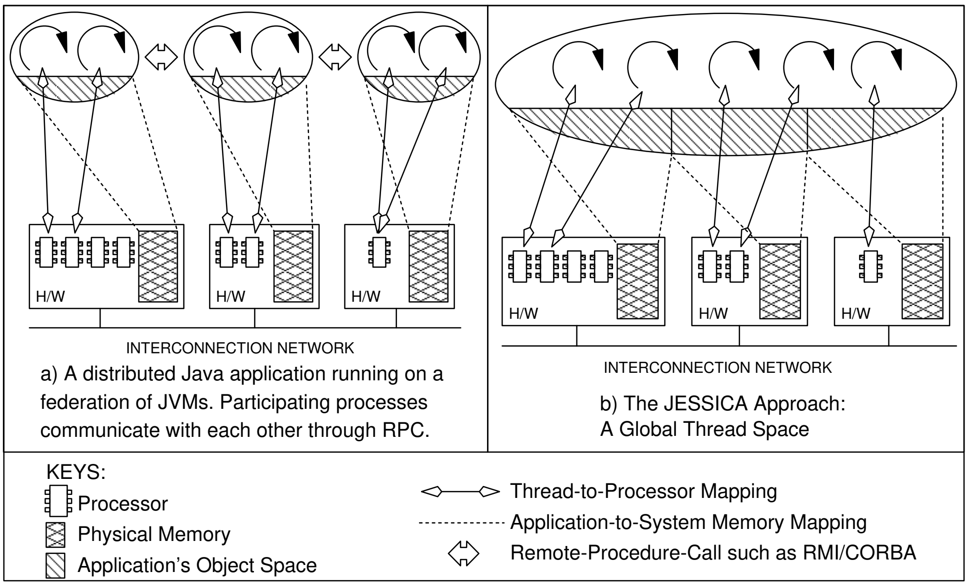


Fig 1: JESSICA encapsulates a cluster of computers into a single multi-processor machine

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

2 JESSICA System Architecture

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. In JESSICA, we classify cluster nodes into *console* node and *worker* nodes as follows:

- **Console Node –** Java applications can be started on any node in the JESSICA cluster. They are running on the distributed virtual machine that follows the *home model,* which is similar to that in Sprite [8] and MOSIX [1]. Any node at which the Java application is initiated will become the *home* of that application and is known as the *console node*. The console node is responsible to handle any system service requests made by a migrated thread that are location dependent. In general, when a thread is migrated from the console node to another node in the cluster, the system resource requests made by the migrated thread will now be served by the local daemon running on the destination node. However, when the daemon discovers a requested service call that is location dependent, the call will be redirected back to the console node and to be handled there.

- **Worker Nodes –** When an application is instantiated on the console node, the role of other nodes in the JESSICA cluster is to support the console to execute the application and they act as the slaves with respect to the console. They are responsible to stand by and serve any requests forwarded from the console, in order to help the console to execute the application to its completion. During its course of execution, the migrated thread will continue to make system service requests to the worker node as if it was running at the console. The worker node will have to differentiate whether the request is location independent or not. If it is independent the request will be served locally, otherwise, the request will be forwarded back to the console. The console, after receiving the request, will perform the necessary operations and return the result back to the thread. The whole redirection process is carried out transparently, and the thread is totally unaware of what goes on.

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

- **Bytecode Execution Engine (BEE)** – It is responsible to bind to an active thread and to execute its method code. Parallel execution of a multi-threaded application is realized by having multiple BEEs running on multiple machines to execute multiple threads simultaneously.
- **Distributed Object Manager (DOM)** – It is responsible to manage the memory resource in its local node and to cooperate with DOMs running on other nodes in the cluster to create the Global Object Space. They allocate new objects within the shared-space where access of objects can be made independent of their locations.
- **Thread Manager (TM)** – It is responsible to create, schedule and destroy threads running on the local node. Moreover, it works with TMs running on other nodes to perform thread migration. They marshal, ship and demarshal the execution states of migrating threads to resume their execution. They support distributed synchronization between migrated threads by forwarding synchronization operations back to the console TM.
- **Migration Manager (MM)** – It is responsible to collect load information of the local node and to exchange the information with MMs running on other nodes, in order to enforce a migration policy for load distribution around the cluster. When MM decides to perform thread migration, it notifies the TM about which thread should be migrated to where.

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

3 Delta Execution

Delta Execution is a preemptive thread migration mechanism devised for JESSICA, which allows a cluster to move threads from the console node to the worker nodes. In this approach, a structural expression is formulated for representing the current execution states of a running thread. The expression enables the system 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, a high-level migration mechanism can be developed so that only the machine independent states of a migrating thread are extracted and are moved to a destination node for execution. The manipulation of any machine dependent states is avoided by leaving them behind at the source node so that any execution that will involve 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 at the destination and then followed by the execution of non-migratable machine dependent instructions back at the source. This process is repeated until all the migratable *delta sets* and non-migratable machine dependent instructions are exhausted, as illustrated in Fig 2.

Since 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, as only the machine independent states are migrated, it is possible to extend the mechanism for supporting heterogeneous migration, where a thread can be migrated to a computer that is of different hardware architecture.

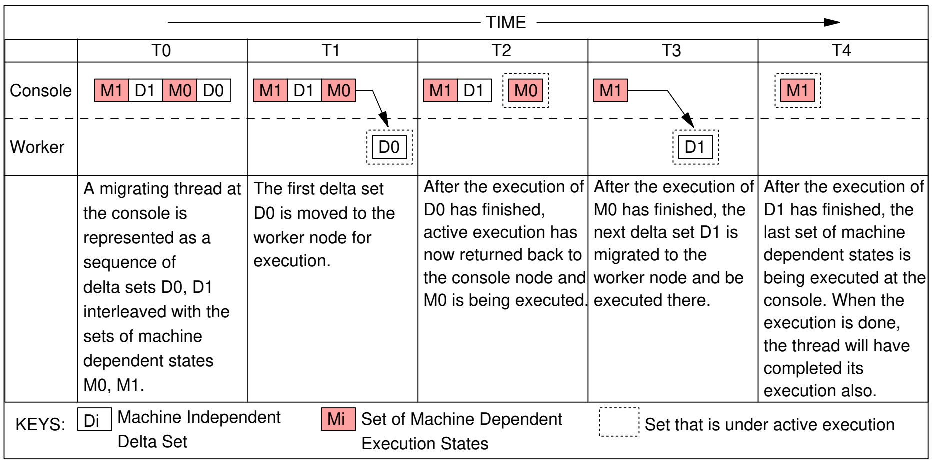


Fig 2: Delta Execution in action

4 Master-Slave Model for Supporting Location Transparency

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

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, called the *master;* and the other running at the destination, called the *slave*. The slave thread is in fact created anew at the destination and acts as the migrated image to continue the execution of the original thread. The master thread remaining at the console is actually the original migrating thread, which is now responsible to perform any location dependent operations like I/O on behalf of the slave, in addition to forwarding messages between the slave and the rest of the system.

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

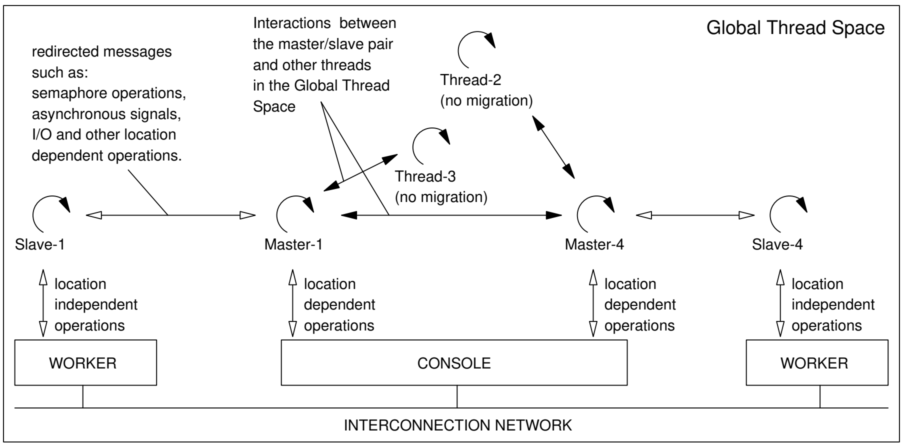


Fig 3: Interactions between the master and the slave threads that hide migration from the rest of the system

5 Performance Evaluation

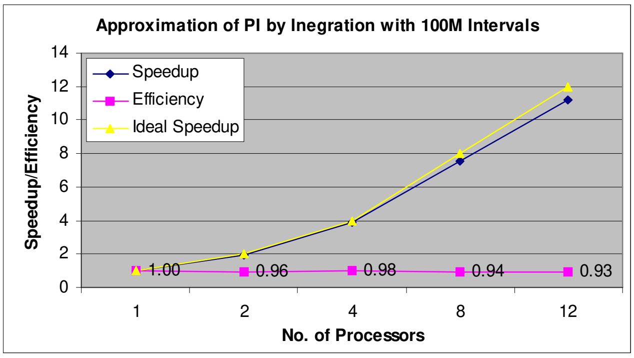
We have implemented a JESSICA prototype that runs on a cluster of 12 SUN Ultra-1 machines interconnected by a 155Mbps ATM switch. The implementation is based on Wilkinson's work on the KAFFE virtual machine [15] and the Treadmarks package [5] for distributed shared-memory support. The following three multi-threaded Java applications are devised to evaluate the performance of the system:

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

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

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

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



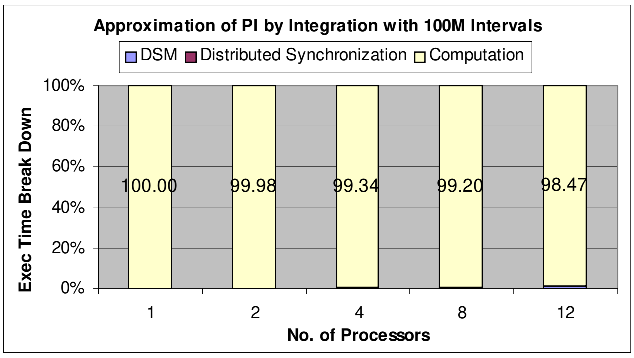
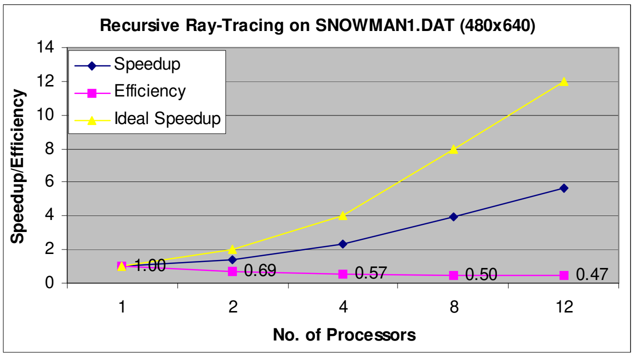


Fig 4: Performance results of the approximation of the value π (PI) with 100M intervals



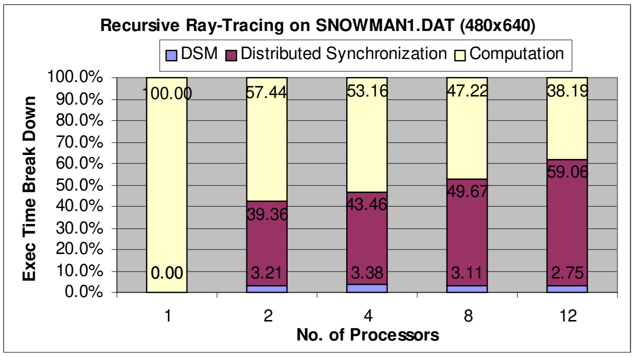


Fig 5: Performance results of the recursive ray-tracer to produce a 480x640 image

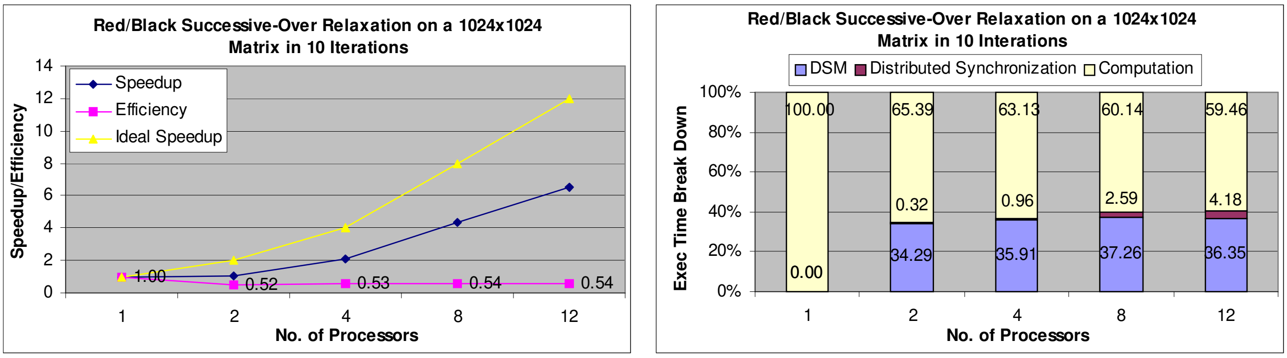


Fig 6: Performance results of the red/black successive-over-relaxation application on a 1024x1024 matrix

According to Fig 4, it can be seen that almost ideal speedup and efficiency are achieved in the π (PI) approximation application, since there is no communication or coordination between worker threads until all the computations are completed. The recursive ray-tracing experiment shows the efficiency is less than optimal and drops moderately as more processors are used. The efficiency decreases from 69% when using 2 processors to 47% when using 12 processors. This is because as indicated in Fig 5, the percentage of distributed synchronization overhead contributes a significant amount to the total execution time. As shown in Fig 6, the R/B-SOR application can gain moderate speedup when running with 4 or more processors. When running with 2 processors, the speed gained by overlapping the computation is offset by the extra overhead incurred due to remote memory access. The efficiency stays at around 53% and improves slightly when the number of processors is progressively increased from 2 to 12. This can be due to the fact that the amount of data shared, i.e. the sizes of the Red and the Black matrices, remain the same when executed by any number of processors, therefore the DSM overheads constitute roughly the same percentage of the execution time.

6 Related Work

Java/DSM [17] is a distributed Java Virtual Machine that runs on a cluster of heterogeneous computers. It provides an illusion to Java applications that they are running on a single JVM with multiple processors attached. Parallel execution of a multi-threaded application is possible by having multiple threads running on multiple nodes in the cluster.

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

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

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

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

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

7 Conclusions and Further Work

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

We are in the process of porting the JESSICA prototype to the SRG-1 cluster, which is a 24-node Intel-Pentium cluster running the Linux operating system. The SRG cluster is equipped with the Direct-Point fast communication subsystem [6] and the JUMP DSM subsystem [3], they enable us to investigate the effect of using different shared-memory consistency models on the performance of JESSICA.

References

[1] A. Barak, O. Laden and Y. Yarom, 'The NOW MOSIX and its Preemptive Process Migration Scheme', *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, Vol. 7, No. 2, pp. 5-11, 1995.

[2] A. 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.

[3] B. Cheung and C.L. Wang, 'A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations', Technical Report, Department of Computer Science and Information System, The University of Hong Kong, 1999.

[4] B. Dimitrov and V. 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.

[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, 'TreadMarks: Shared Memory Computing on Networks of Workstations', *IEEE Computer*, Vol. 29, No. 2, pp. 18-28, Feb 1996.

[6] C.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.

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

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

[9] J. Gosling, B. Joy and G. Steele, 'The Java Programming Language Specification', Addison Wesley, 1996.

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

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

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

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

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

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

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

[17] W. Yu and A.L. Cox, 'Java/DSM: A Platform for Heterogeneous Computing', ACM 1997 Workshop on Java for Science and Engineering Computation, Jun 1997.

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