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

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

Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam, Hong Kong
Email: {jmma, clwang, fcmlau}@csis.hku.hk

April 25, 2000

Proposed running head:

- JESSICA (first choice)
- JESSICA: Java-Enabled SSI Computing Architecture (second choice)

*A preliminary version of this paper appears in *Proc. of 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, June 1999. Correspondences should be sent to M.J.M. Ma, Department of Computer Science and Information Systems, The University of Hong Kong, Pokfulam, Hong Kong / Email: jmma@csis.hku.hk / Fax: (+852) 2559 8447.

Abstract

JESSICA stands for "Java-Enabled Single-System-Image Computing Architecture", a middleware that runs on top of the standard UNIX operating system to support parallel execution of multi-threaded Java applications in a cluster of computers. JESSICA hides the physical boundaries between machines and makes the cluster appear as a single computer to applications—a *single system image*. JESSICA supports preemptive thread migration which allows a thread to freely move between machines during its execution, and global object sharing through the help of a distributed shared-memory subsystem. JESSICA implements location-transparency through a message-redirection mechanism. The result is a parallel execution environment where threads are automatically redistributed across the cluster for achieving the maximal possible parallelism. A JESSICA prototype that runs on a Linux cluster has been implemented and considerable speedups have been obtained for all the experimental applications tested.

Keywords: cluster computing, single system image, dynamic load balancing, thread migration, Java Virtual Machine, JESSICA

1 Introduction

Cluster computing has been a subject for active research in recent years. A cluster of computers is a federation of computers linked by an interconnection network where the computers run integration software to support collaborative computations. Many positive results on using clusters of computers for load sharing and parallel computing have been reported [10, 13, 2].

The Java programming language [5] has received unprecedented acceptance for a programming language since its introduction in late 1994. Java provides a multi-threaded model for concurrent programming as a built-in feature, which could have great potentials for parallel processing. However, in the absence of specialized software at a lower level to support parallel processing, the programmer has to tackle the coordination problem between processing nodes at the application level using some interprocess communication mechanism such as sockets. Since the introduction of JDK version 1.1, the burden on the programmer is alleviated by the provision of Object Serialization [7], Remote Method Invocation (RMI) [8] and Common Object Request Broker Architecture (CORBA) support [6]. They allow process coordination and cooperation at the method-call level, using some remote procedure call mechanism as illustrated in Fig. 1a. The programmer, however, still has to worry about the availability and operation status of the processing nodes involved. Parallel programming based on these supports remains to be a nontrivial task.

We propose establishing a *single-system-image (SSI)* illusion over a cluster as a means to bridge cluster computing and Java's multi-threaded programming model. The idea behind SSI is

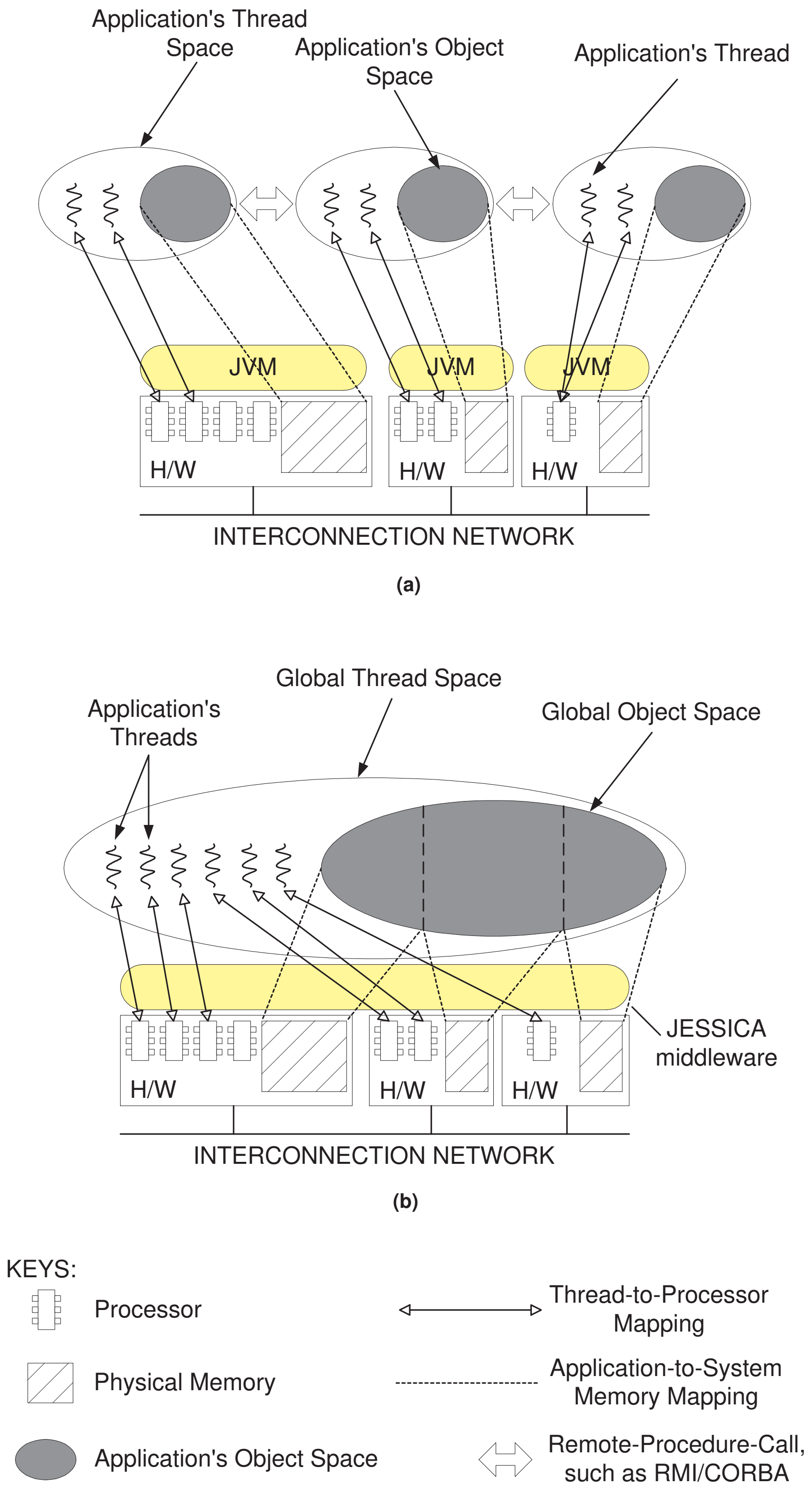


Figure 1: (a) A distributed Java application. (b) JESSICA transforms a cluster into a single multi-processor machine.

to encapsulate system resources distributed across the cluster in a layer of abstraction, such that components above the layer will see the encapsulated resources as a single, unified entity. Our approach is to establish an SSI illusion at the middleware level in the form of a distributed Java Virtual Machine (JVM). We favor this approach because it does not require any modification to the operating system or to the Java applications running on top. It guarantees portability over various popular operating systems and compatibility with existing Java applications.

JESSICA is our solution for supporting parallel application development in Java in a cluster using the multi-threaded model. JESSICA stands for "Java-Enabled Single-System-Image Computing Architecture". It is a middleware that hides the distributed nature of a cluster and provides multi-threaded Java applications with the illusion of a single multi-processor computer. The SSI illusion is realized through the provision of a *global thread space*. When an application is instantiated, the JESSICA system creates a logical thread space that spans the whole cluster for the execution of the application's threads, as shown in Fig. 1b. The global thread space hides the physical boundaries between machines and allows Java threads to freely move from one machine to another. The movement is supported by a preemptive thread migration mechanism called *delta execution*. Java objects are globally accessible within the *global object space*, a subspace of the global thread space (Fig. 1b) created through the support of a distributed shared-memory subsystem at a lower level. Migrated threads can continue to communicate with one another and location-dependent resources can still be accessed transparently with the help of a redirection mechanism.

With the SSI illusion in place, application programmers can create as many threads as needed as in a single execution environment, and rely on JESSICA to automatically redistribute them across the cluster to exploit the maximal parallelism obtainable from the cluster and to optimize the overall resource utilization.

The main characteristics of JESSICA include:

- Single-system encapsulation—The whole cluster is encapsulated into a single computing system. All Java threads created in a user program can be executed at any node in the cluster and the threads need not be aware of their physical location.

- Preemptive migration of Java threads—A Java thread can be preempted and migrated to another node at any time during its execution.

- Dynamic load balancing—Based on the thread migration technique, parallel execution of an application can be achieved by simply creating as many threads as needed. Threads are automatically redistributed across the cluster to exploit real parallelism.

- Migration and location transparency—Any location-dependent resources are transparently accessible by a migrated thread. The fact that the thread has migrated is not known to the thread itself and other objects in the system.

- Compatibility—The implementation of JESSICA is at the middleware level and is compatible with the standard JVM [11]. Existing applications therefore are readily runnable on this system without any modification.

- Portability—The JESSICA middleware, being a distributed version of the JVM, runs on top of the standard UNIX operating system as a distributed application. The implementation does not need any low-level or platform-specific supports. Thus it is portable across different hardware platforms.

The rest of the paper is organized as follows. In Section 2, we present the system architecture of JESSICA. Section 3 discusses our solutions to achieve the SSI illusion through the global thread space. Section 4 evaluates the performance of our JESSICA prototype. Section 5 surveys other works that are related to JESSICA. We conclude by summarizing our experiences.

2 System Architecture

Fig. 2 shows an overview of the JESSICA system architecture. At the top is the programming and execution environment as seen by the application programmer—a single global thread space powered by multiple CPUs. This "illusion" layer is realized through the services of three important subsystems which handle redirection of system requests, sharing of distributed memory, and thread migration respectively. The actual implementation of these subsystems is by means of daemon processes running in the different nodes of the cluster. These daemon processes execute as user-level processes on top of the UNIX operating system.

In JESSICA, we classify a cluster node as either a *console* or a *worker* node, as follows.

- Console node—Java applications can be started in any node in the JESSICA cluster. The

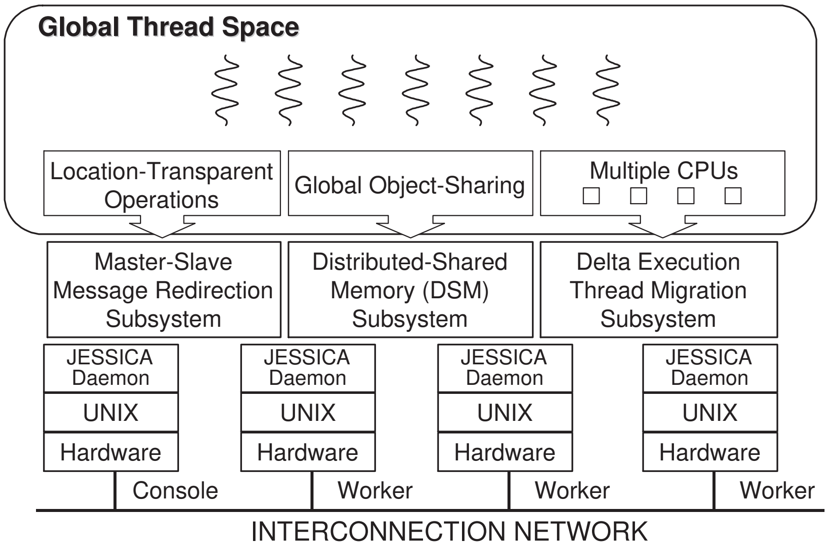


Figure 2: The global thread space creates an SSI illusion over a cluster of computers

console node of an application is the node in which the application is first instantiated—*i.e.*, the application's *home*.

- Worker node—Worker nodes are the other nodes that have one or more migrated threads of the application. These nodes play a subordinate role to the console node by serving requests directed from the console.

The console node is responsible for handling system service requests from a migrated thread that are location-dependent. An example of a location-dependent request is to get the current time maintained by the node. A request to read from a byte stream would be location-independent because the buffer holding the bytes is in the global object space. During execution, system service requests made by a migrated thread will be attended to by the concerned worker node. The worker node will determine whether a request is location-independent or not. If the request is location-independent, it is served locally; otherwise the request is forwarded to the console. The console, after receiving the request, will perform the necessary operations and return the result back to the migrated thread. The whole redirection process is carried out transparently.

2.1 JESSICA Daemon

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

- Bytecode Execution Engine (BEE)—It is responsible for binding an active thread and executing 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 for managing the memory resources in its local node and to cooperate with other DOMs on the other nodes to create a global object space. The physical locations of objects are transparent to the threads within the global object space.
- Thread Manager (TM)—It is responsible for thread creation, scheduling, and termination in the local node. During the course of migration, it coordinates with TMs on the other nodes to marshal, ship, and demarshal the execution contexts of migrating threads. TMs support distributed synchronization of migrated threads by forwarding semaphore operations back to the console TM.
- Migration Manager (MM)—It is responsible for collecting load information of the local node and exchanging those information with MMs running in other nodes in order to implement a migration policy.

Fig. 3 illustrates the interactions between the four system components of a JESSICA daemon. When the code which BEE is executing needs to create a new active thread t, BEE requests DOM to allocate a new thread object from the global object space, after which BEE will execute the constructor code to instantiate the new object. Next, BEE sends a request to TM to activate t. TM will bind t to a new execution context and insert t into the thread scheduling subsystem. Later on when t is scheduled to run, TM binds t to BEE for executing the thread's `start()` method. During t's execution, BEE can create new objects and will make sure that objects are updated consistently in the global object space through the `new()` and DSM `lock()/unlock()`

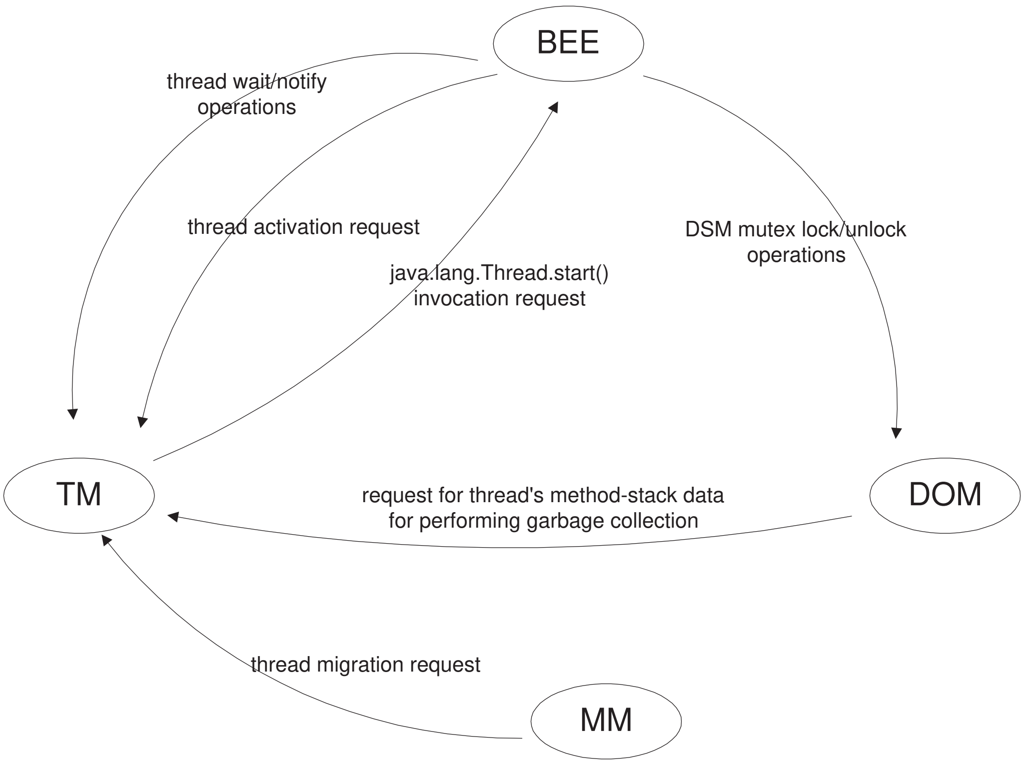


Figure 3: Interactions between system components in JESSICA

primitives provided by DOM. It can also let t communicate or synchronize with other threads by using the thread `wait()/notify()` and mutex `lock()/unlock()` primitives provided by TM.

Whenever necessary, DOM will perform garbage collection to reclaim unused objects. It asks TM to provide the runtime stacks of all the active threads and starts tracing from these stacks to locate any unreferenced objects.

At any time when the console MM detects sufficient load imbalance in the cluster and decides to migrate t to another node, it notifies TM to preempt the execution of t and migrates it to the selected node. The current JESSICA prototype employs a simple migration policy: MMs running in the worker nodes periodically submit their system loads to the console MM; the console MM will initiate a migration if the load imbalance between itself and one of the worker nodes exceeds a predefined threshold; the console MM will instruct the TM as to *which* thread to migrate and *where* to migrate the thread to.

3 Global Thread Space

The global thread space is a global execution environment for running threads which extends across the entire cluster. It is a key component in creating the desired SSI illusion. Threads running within this space would see the underlying cluster as a single computing system with multiple processors, a single memory space for object allocations, and location-transparent system resources. The global thread space is implemented via three important subsystems or mechanisms—the *delta execution* mechanism for supporting preemptive thread migration, the *master-slave message redirection* subsystem for supporting location-transparent operations, and a distributed shared-memory (DSM) subsystem that creates a *global object space* for supporting distributed object access.

3.1 Delta Execution

Delta execution is a preemptive thread migration mechanism for supporting transparent thread-to-processor mapping within the global thread space of JESSICA [9]. Migration granularity is per-bytecode-instruction where a thread can be preempted and migrated once execution of the current bytecode is completed. Parallel execution in JESSICA is realized by letting different JESSICA daemons execute multiple threads of an application in different nodes simultaneously.

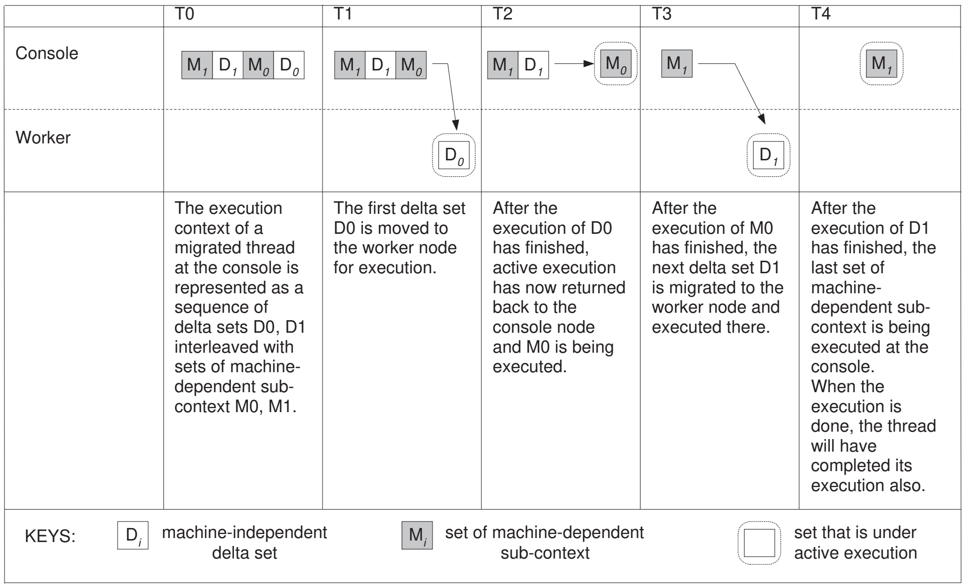


Figure 4: Delta execution in action

Delta execution aims at providing a high-level and portable implementation for Java thread migration that completely hides all the low-level or system-dependent details. Because the whole mechanism is implemented within the JESSICA middleware, migration is therefore transparent to Java applications running on top of JESSICA and no migration-specific code needs to be added to the applications.

The delta execution mechanism identifies and separates the machine-dependent sub-contexts from the machine-independent sub-contexts in the execution context of a migrating thread. Machine-independent sub-context refers to state information that can be expressed in terms of the execution state of the distributed virtual machine in JESSICA, such as data stored in the virtual machine's registers. Machine-dependent sub-context is state information that is part of the internal state of the JESSICA daemons, such as the hardware program counter which points to the current machine instruction when a daemon is executing a native method. As illustrated in Fig. 4, a thread's execution context consists of sets of machine-independent sub-context, also known as *delta sets*, which interleave with the sets of machine-dependent sub-context.

A master-slave model for thread migration. When a thread running in the console node migrates, it does not pack up its entire execution context in order to move to the destination worker node, which is the case in traditional process migration systems such as Sprite [4]. Instead, it is split into two cooperating entities, one running in the console node, called the *master*, and the other running in the worker node, called the *slave*. The slave thread is created anew at the worker node which will continue the execution at the point where the original thread stopped. The execution context is divided into machine-dependent sub-contexts and machine-independent delta sets as identified by the delta execution mechanism. All the machine-dependent sub-contexts are processed locally by the master thread while the machine-independent delta sets are transferred to the slave thread one by one for execution, as shown in Fig. 4.

Active execution of the migrated thread is seen as a sequence of executions, using the machine-dependent and the machine-independent sub-contexts, that switch back and forth between the console and the worker node. Since the migrated thread only *incrementally advances* its execution by a *delta* amount every time when control is switched to it, we therefore call this mechanism *delta execution*. Because of the master-slave design, the mechanism provides an opportunity for the implementation to isolate machine-dependent contexts from machine-independent contexts and process them in a manageable way.

Dynamic load balancing. JESSICA supports dynamic relocation of threads in order to achieve dynamic load balancing. After migrating a thread from the console node to a worker node, it is possible for the migrated thread to move to yet another worker node or to retreat back to the console node. When a migrated thread running in a worker node is required to further migrate, it will first retreat back to the console, after which the Migration Manager will select another worker node to migrate the thread to. The reason for this approach, as opposed to one that migrates the thread to the new worker node directly, is because if a migrated thread is allowed to directly migrate to another worker node without first retreating back to the console, residue dependency required for maintaining migration transparency will be left there with the first worker. Messages forwarded from the console will have to go through this first worker node before they can reach the new home of the migrated thread. This would result in one more level of redirection. If further migrations are made, there will be more levels of redirection through residue dependencies left behind in many worker nodes the thread has ever visited. Such a chain of residue dependencies would be difficult to manage. We therefore opted for the simple approach of first migrating the thread back to the console before performing another migration, and because of that, residue dependency in the first worker node will be removed together with the leaving thread.

The current implementation relies on Migration Managers running in all the worker nodes to provide load information across the cluster for making migration decisions. Such information is obtained from the process file system `</proc>` of each node. The Migration Manager at the console node queries its counterparts running in each worker node for load information every second. The percentage of time that a node spends in user mode between successive queries is the primary kind of load information used in migration decisions. If the Migration Manager at the console discovers that the percentage of time that a node is spending in user mode between successive queries is one-fifth or less of that of the console, it will go through the list of actively running threads to select a non-daemon thread to migrate to this underloaded node. Priority will be given to a running thread whose execution state does not contain any machine-dependent information. The Migration Manager may also trigger a redistribution of migrated threads if it comes across a worker node that is heavily loaded. A worker node is considered heavily loaded if the percentage of its user-mode time is more than double that of the console. When this happens, the Migration Manager will send a message to the identified worker node which will then select one of its actively running slave threads to retreat back to the console. If an underloaded node is found later on, the retreated thread could be migrated again.

3.2 Location-Transparency Support by Redirection

After a thread has been migrated, the master thread remaining at the console represents the original thread and is responsible for performing any location-dependent operations, such as I/O, for the slave. All the thread-level interactions, such as `wait()`/`notify()` and mutex-`lock()`/`unlock()` between the slave and other threads will go through the master. Redirections of service requests and responses make the master appear to other threads as the only thread they are interacting with. On the slave side, all the location-dependent operations are redirected transparently back to the console, while the remaining location-independent operations are carried out locally. With this design we are able to create a global thread space that has the same semantics and maintains the same relationships between objects in the execution environment as the case with no migration, as depicted in Fig. 5. We are also able to implement location-transparent services such as network and file I/O operations, and distributed thread sychronization. Consequently, the JESSICA execution environment as observed by a running thread is the same as that of a standard Java Virtual Machine (JVM). Multi-threaded applica-

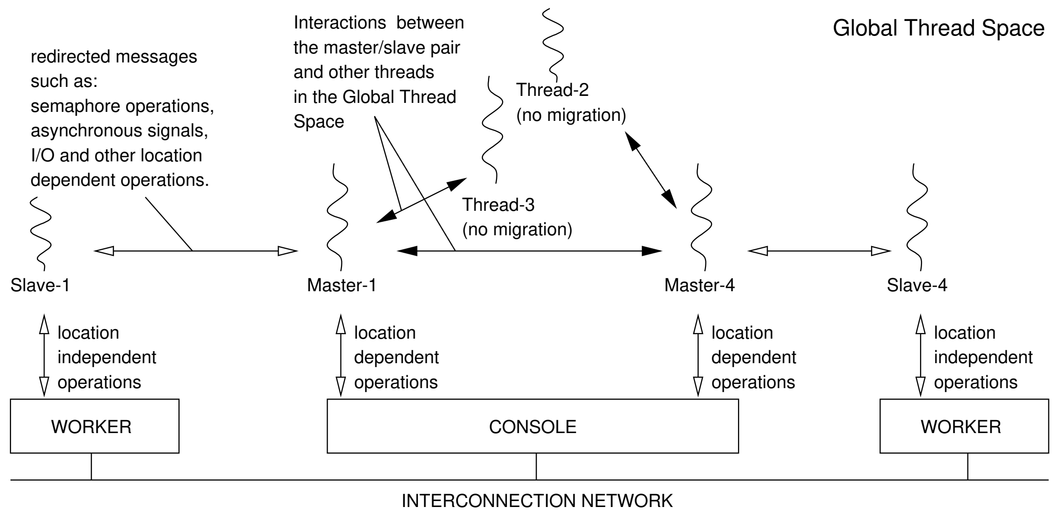


Figure 5: Interactions between the master and the slave threads

tions runnable on a standard JVM will also execute correctly on JESSICA.

I/O redirection. JESSICA ensures that all opened files and network communication channels following a migration will remain functional as they have been before the migration. Traditional process migration systems that follow the kernel-level approach store control information relating to opened files and network channel end-points inside the kernel space. Thus, migrating these resources transparently with the process execution context is difficult. In contrast, JESSICA adopts a middleware-level approach which makes it possible to support location-transparent I/O by extending the I/O subsystem implementation alone.

In JESSICA, location-transparent I/O support is achieved by redirecting I/O operations back to the console node and letting the master there to perform the operations on behalf of the slave. The redirection code is implemented within the `java.io` and the `java.net` class libraries for file and socket I/O redirection respectively. Their interface definitions are kept unchanged so that other classes relying on them do not need to be modified. To improve performance, a buffer cache allocated from the DSM is used to buffer I/O data for each opened file or socket. When a slave thread performs a `read()` operation on an opened file, for example, it checks whether the requested data has been loaded into the shared buffer already. If so, the data is retrieved from the buffer directly. Otherwise, the slave thread redirects the operation back to the console. The master thread will issue a `read()` operation to the underlying operating system to fill up the buffer. Eventually, the slave thread is notified and the requested data can then be obtained from the shared buffer.

The object-oriented nature of JESSICA has helped simplify the implementation of I/O redirection, since class hierarchies of both the `java.io` and the `java.net` libraries are sufficiently well organized. There are base classes located towards the top of the hierarchies that are responsible for performing the raw I/O operations through the underlying operating system. All of their child classes that specialize in I/O operations for specific data types inherit the functionality directly from the base classes. These specialized classes therefore can simply invoke the inherited methods to access the raw I/O channels. As a result, when the base classes in the hierarchies are extended to support the required I/O redirection, the rest of the child classes can inherit the feature immediately without any further modification.

Cooperative semaphore. The distinctive feature of multi-threaded computing is that threads can share resources and execute concurrently in order to multiplex computations, or even communications. In such a multi-threaded runtime environment, a mechanism for providing mutual exclusion is necessary for ensuring coordinated access to shared resources so that data integrity is maintained. The Java programming language uses semaphores at the virtual machine level to implement mutual exclusion control. A semaphore is associated with every shared object so that application programs can avoid any race condition by waiting on the associated semaphore before updating a shared object. Thread synchronization is a direct consequence of using semaphores.

JESSICA adopts a decentralized approach to implementing distributed semaphore. A Thread Manager need not be aware of the existence of other Thread Managers; each of them can perform their own local thread scheduling without affecting the others. In addition, all the semaphore operations are performed in the console node by the master thread, and so the same semaphore semantics is enforced as if there is no migration. The implementation of distributed semaphore in JESSICA is called *cooperative semaphore*.

The implementation of cooperative semaphores ties closely to the Thread Manager for handling blocking and resuming of active threads. A cooperative semaphore is created the first time a mutex-lock is applied to the object. If the lock operation is initiated by a slave thread, the corresponding cooperative semaphore will be created by its master thread at the console. Cooperative semaphores are part of the internal control structure of JESSICA and do not live in the DSM. A cooperative semaphore maintains a count and a queue of blocking threads that try to perform a mutex-lock on the corresponding Java object. If the count is zero, a thread can immediately lock the object and increment the count; otherwise the thread is scheduled out by the Thread Manager and appended to the queue. A thread unlocking the object decrements the count. When the count reaches zero, the first blocked thread from the queue will be scheduled to run by the Thread Manager. Note that the mechanism just described applies to master threads as well as normal threads that have not been migrated; for a migrated slave threads, the lock and unlock operations are redirected back to the console as described below.

Observe that a running thread will be blocked and forced to leave the ready queue if

- it tries to lock a semaphore which is currently held by another thread,
- 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.

A blocked thread T will be rescheduled back to the ready queue when

- the semaphore that T has previously requested is unlocked by another thread and it is now T's turn to lock the semaphore,
- the I/O channel that T previously tried to operate on is now ready, or
- another thread has issued a *notify* operation on an object 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 where data have not yet arrived. When a slave thread tries to lock a semaphore S, instead of directly operating on S, it sends a message to its master, asking the master to lock S on its behalf. After that the slave thread will be blocked waiting for the master's reply. At the console node, when the master thread receives the semaphore lock request for S from its slave, it will try to lock the semaphore S. When eventually the master has successfully locked the semaphore S, it will then send a success message back to its slave so that the slave can continue its execution, as if the slave has successfully locked the semaphore itself. Similarly, when the slave thread later tries to unlock S, it again sends a message to the master asking it to unlock S on its behalf. After the master has received the message and unlocked the semaphore, the local thread manager at the console can then reschedule some other thread that has previously issued a lock request for the semaphore.

With the above design the observed effect for the slave thread is that a semaphore lock operation will block until the semaphore is unlocked, and a semaphore unlock operation will cause other threads that are also trying to lock the semaphore to be rescheduled. The effect is the same as if the semaphore operations are performed locally. At the same time, for the other threads that are running in the console node, what they observe is that it is the master thread that performs all the semaphore lock and unlock operations. Hence, cooperative semaphore can transparently hide the fact of migration from the rest of the system.

Distributed thread synchronization. Java threads rely on simple wait-notify signals for inter-thread communications. By a design similar to that for cooperative semaphores, we have implemented a remote thread signaling mechanism where the master is responsible for transparently forwarding any *wait* and *notify* signals between its slave and the other threads running in the console node. With the cooperative semaphore and the remote signaling mechanisms installed, we are able to implement the distributed thread synchronization mechanism in a decentralized manner.

3.3 Global Object Space

The global object space is the part of the global thread space which provides location-independent object access. In each node of the cluster there is a *distributed object manager* (DOM) responsible for managing the local memory resources as well as cooperating with DOMs running on other nodes to create a globally accessible object space. This is achieved by implementing the DOMs on top of a distributed shared-memory (DSM) subsystem. With the help of the DSM subsystem, discrete memory regions belonging to various cluster nodes are unified to form a single and contiguous memory space for global object sharing. As a result, objects remain to be accessible by a thread even after the thread has migrated to another node in the cluster. The location where an object resides is transparent to a thread.

The global object space is for the containment of all Java objects. It constitutes a portion of the entire address space at the virtual machine level, where the stack and the local heap of each JESSICA daemon process are used to store other internal state of the runtime system. Contents of objects that are located in a remote node are cached by the DSM subsystem, and JESSICA relies on the cache coherent protocol provided by the DSM subsystem to maintain the consistency of the cached data. The global object space is established by allocating a large chunk of shared memory from the DSM. JESSICA employs a decentralized approach for memory management, where the DOM running on each node is responsible for managing its own share of the global shared memory. Object allocation requests made by the threads in a node are always satisfied locally—the DOM will allocate the required space from its share instead of forwarding the request back to console. Because it is possible to have two or more nodes updating the same object at the same time, the `lock()` and `unlock()` primitives provided by the DSM subsystem are used to ensure consistency.

Distributed garbage collection. Garbage collection is the activity to locate all the unused objects and to reclaim their space. As the Java Virtual Machine relies on garbage collection to reclaim unused memory objects, a distributed mark-and-sweep garbage collection mechanism is installed in the global object space of JESSICA. Distributed garbage collection is a nontrivial problem still under active research [3]; our approach is to adopt a simple solution for the JESSICA prototype. Since DOM handles local object allocation requests by allocating requested memory from its own subspace, it is not difficult to deduce which DOM is the owner of any given object by looking at the object's address. During the marking phase of garbage collection, each DOM will form lists of all traceable objects that belong to other DOMs. The lists of objects are forwarded to their respective owners at the end of the marking phase. Consequently, a DOM will be able to sweep and reclaim only unreferenced objects, as any object that is referenced remotely can be identified from lists coming from the other nodes.

Interaction with the DSM subsystem. The following example, in which a synchronzied method is invoked to update a field of a shared object, illustrates how the implementation interacts with the DSM subsystem. The DSM implementation relies on the *lazy release consistency* model.

When a worker thread invokes a synchronized method to update some field of an object, the thread will first try to mutex-lock the object using the corresponding cooperative semaphore associated with the object. The worker thread may be required to block if the object is being locked by another thread. Once the cooperative semaphore has been successfully locked, the system will perform a DSM-lock on the shared memory page that contains the object. To do this, the system sends a message to the DSM lock manager requesting the lock; the lock manager will reply with a successful message later on, and information about which DSM pages need to be invalidated will be attached to the reply message. The worker thread will then try to access the object, resulting in a page fault being generated. The DSM subsystem will update the content of the page by applying "page diffs" received from other nodes, if there are any. After the update the worker will then be allowed to perform a write on the object's field. The DSM subsystem will create a twin of the page being modified before the write is performed on the original copy. The twin is required to produce a diff to describe the changes made to the page later when another node performs a DSM-lock on the page. When the worker thread finishes updating the object and exits the synchronized method, the system will perform a DSM-unlock on the shared page. An unlock message and the IDs of shared pages that have been modified will be sent to the lock manager as a result. Finally, a mutex-unlock will be applied to the object, completing the process.

4 Performance Evaluation

The first JESSICA prototype ran on the Solaris platform [9]. We have since ported it to the Linux platform. Our Linux cluster consists of 8 Linux PCs connected to a 100Mbps Fast Ethernet switch. Each PC is equipped with a 300MHz Intel Celeron processor and 128MB main memory, and is running Linux Kernel 2.2.1. The JESSICA implementation is based on version 0.9.1 of the *Kaffe virtual machine* [12] and uses version 1.0.3.2 of the *Treadmarks* DSM package [1].

We had to make some major modifications to the Kaffe implementation in order to support the SSI-enabling features. For example, in order to facilitate the extraction of a thread's execution context, the method invocation mechanism in Kaffe's bytecode execution engine was changed so that it would allocate the method stack from the local heap instead of from the process runtime stack. The set of bytecode instructions that are responsible for method invocation were also adjusted in order to support the delta execution mechanism. In addition, the Distributed Object Manager has been incorporated into the memory management subsystem for creating the global object space. All the bytecode instructions that access the global object space have been augmented to use the `lock()/unlock()` primitives provided by the DSM whenever necessary. Moreover, the thread subsystem has been extended to become the Thread Manager for supporting thread migration, cooperative semaphore, and the remote signaling mechanism. Finally, the Migration Manager responsible for enforcing a load balancing policy has also been incorporated into the system.

All communications between the JESSICA daemons are conducted through the BSD socket interface provided by the Linux operating system.

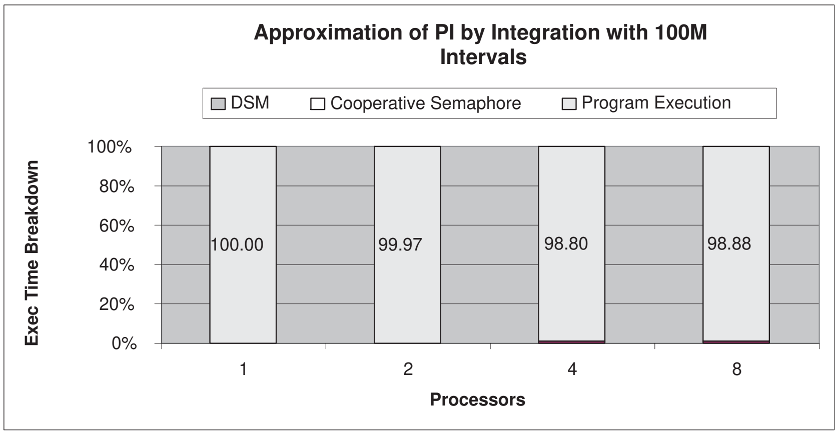
4.1 Applications Performance

We have implemented the following three multi-threaded Java applications in order to measure the performance of our JESSICA prototype. The prototype was set to utilize the maximal parallelism obtainable from the cluster—newly created worker threads are automatically migrated to worker nodes by our load-balancing mechanism until there is a worker thread running in each worker node. Since the applications are designed to create the same number of worker threads as the number of processors available, we are able to study the effect on the execution time when all the threads are running in parallel.

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

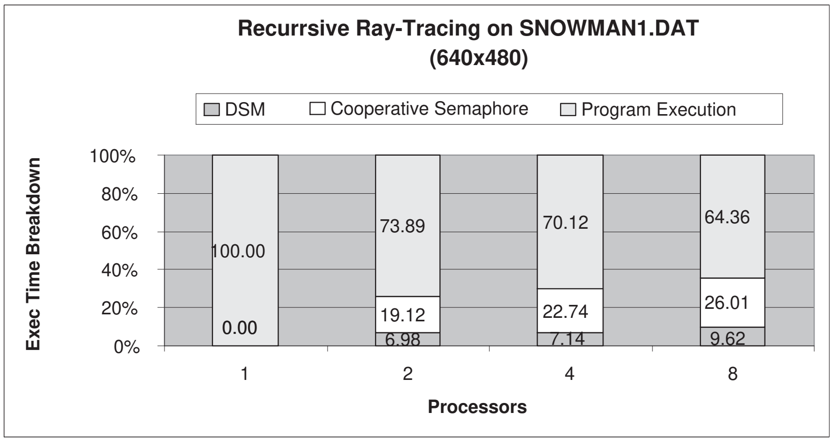
- Recursive Ray-Tracing—In this recursive ray-tracer written in Java, worker threads render the pixels of a projected 2D image by shooting rays into a given 3D scene. The threads would fetch the next line of pixels to compute from a globally shared job queue, leading to a load-balancing effect at the application level. All worker threads are tightly synchronized among themselves when they access the job queue in order to maintain consistency. This application demonstrates how distributed thread synchronization affects the performance of the worker threads that are distributed across the cluster.

- Red-Black Successive Over-Relaxation (R/B-SOR) on a Grid—This program creates multiple threads to compute matrix elements in parallel. A large 512×512 matrix is divided into two sub-matrices, the Red and the Black matrix, which in turn are divided into roughly equal-size bands of rows, with each band being assigned to a different thread. The threads repeatedly retrieve values from one matrix, compute the average, and write the result to the other matrix. Since the input matrix is allocated from the globally shared DSM space,



No of Processors	SpeedUp
1	1
2	2.00
4	4.00
8	7.99

Figure 6: Performance results of π approximation with 100M intervals



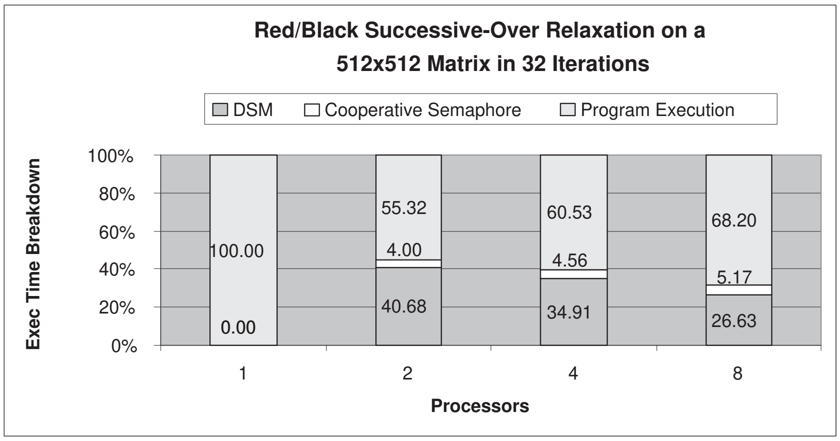
No of Processors	SpeedUp
1	1
2	1.66
4	3.13
8	5.44

Figure 7: Performance results of the recursive ray-tracer to produce a 480×640 image

the execution imposes a significant amount of loading on the DSM subsystem. Hence, this is a good candidate for studying how the DSM overhead due to migration contributes to the overall execution time.

The results are presented in Fig. 6, Fig. 7, and Fig. 8. The execution times presented 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 was started after the sequential initialization phase so as to arrive at a more accurate estimation on the performance improvement.

According to Fig. 6, it can be seen that almost ideal speedup and efficiency are achieved in the π approximation application, due to the fact that there is no communication or coordination between worker threads until all the computations are completed. The recursive ray-tracing



No of Processors	SpeedUp
1	1
2	1.50
4	2.51
8	3.36

Figure 8: Performance results of the red/black SOR application over a 512 × 512 matrix

Execution time	π Approximation	Ray-tracing	R/B-SOR Matrix Iteration
JESSICA	400 sec	385 sec	247 sec
Kaffe	385 sec	264 sec	107 sec
Percentage Slowdown	4%	46%	131%

Table 1: Comparison of application performance when using a single worker thread

experiment shows that the efficiency is less than optimal and drops slightly as more processors are used. The efficiency decreases from 83% when using two processors to 68% when using eight processors. This is because, as indicated in Fig. 7, the distributed synchronization overhead contributes a larger amount to the total execution time as more nodes are used. As shown in Fig. 8, the R/B-SOR application can achieve moderate speedups as more processors are used. The efficiency drops from 75% when running with two processors to 42% when running with eight processors. The result shows that DSM overheads contribute a significant portion to the execution time and make the application less scalable.

4.2 JESSICA versus Unmodified Kaffe

In this section, we compare the application performance of JESSICA using one worker thread with the performance of the original, unmodified Kaffe. Table 1 shows the results.

Both JESSICA and Kaffe exhibit almost identical performance for the π approximation application because the application only relies on the stack to store intermediate results, rather than using Java objects, and hence there is virtually no extra overhead generated by the DSM. This shows that the BEE implementation in JESSICA is as efficient as that of Kaffe. The moderate slowdown shown in the ray-tracing application for JESSICA is due mainly to the DSM overheads when reading the 3D scene objects which are Java objects stored in the DSM. Similar to the π approximation application, because most of the intermediate results are stored in the method stack, the application does not generate an excessive amount of DSM overhead from data updates. Finally, the R/B-SOR application shows the slowest performance for JESSICA when compared to Kaffe. This is because the application had a large number interactions with the DSM as it needed to update the R/B matrix, which contains 0.25 million elements stored in the DSM, many times repeatedly.

From the results we can see the current JESSICA implementation performs the worst for applications that require a huge amount of data updates to be performed through the DSM. The major overhead comes from the Treadmarks DSM because of its lazy release consistency, where every object access would require invoking a pair of DSM-lock and unlock operations for the sake of data consistency. Future versions of JESSICA will look for better substitute for the current DSM subsystem. An ideal DSM should provide an access semantics that is identical to ordinary memory, which can free the system from having to perform any DSM-lock and unlock operations in order to obtain the up-to-date contents of an object.

4.3 Dynamic Load Balancing

This section studies the effect of our simple migration policy and the dynamic load balancing capability of JESSICA. We used our π approximation program which computes the value of π by integration. In the experiments, the program would create two worker threads with each thread evaluating an assigned area under a graph concurrently.

In the simplest case where no artificial load was introduced into the system, thread migration occurred at the beginning of the program execution as the Migration Manager detected load imbalance between the console and the rest of the worker nodes. Each of the worker threads was assigned to a different node of the cluster, and the two threads executed in real parallelism. This was in effect a *static assignment*. The execution time in this case was 193 seconds.

In the next experiment, the same program was run. This time, however, the retreat feature of the thread migration mechanism was left disabled so that a migrated thread would continue

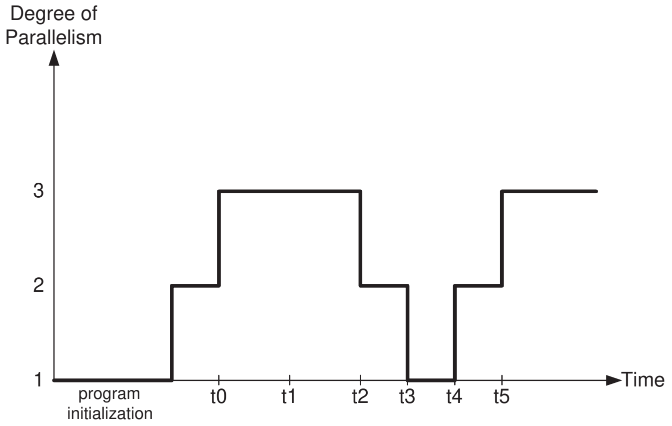


Figure 9: Change in degree of parallelism against time in the dynamic load balancing experiment

to execute on a worker node regardless of any change in CPU loading. Some artificial load was then inserted (one second after the threads began to execute) into the system to overload the worker nodes on which the threads were running. The extra load slowed down the execution of the migrated threads and the execution time increased to 328 seconds.

In the final experiment, the same artificial load was used, but the retreat feature of thread migration was enabled this time. As shown in Fig. 9, after the program began its execution, the Migration Manager migrated the threads to two separate worker nodes. At $t0$, the degree of parallelism reached three as there was a Migration Manager running at the console and two migrated threads were running at two worker nodes. After the artificial load was introduced one second later at $t1$, the Migration Manager detected a sharp increase in system load on the worker nodes, causing the Migration Manager to trigger a retreat operation at each worker node, and the migrated threads were then sent back to the console at time $t2$ and $t3$ respectively. The degree of parallelism was reduced to one at this time. Soon after the Migration Manager discovered there were in fact five other worker nodes sitting idle, it triggered another migration and caused the two worker threads to migrate to two other worker nodes at $t4$. Eventually, the degree of parallelism returned to three again at $t5$. The execution time was 207 seconds.

From the experiment results we can see the artificial load caused the execution time to increase by 70% if the retreat feature is disabled. With the retreat feature enabled the increase in execution time dropped to a mere 7%. Since during the period between $t2$ and $t5$, the program could not execute in full parallelism, which also contributed to the 7% slowdown, we conclude that the the migration and the retreat operations are reasonably efficient.

4.4 Primitive Operations Overheads

This section studies the overheads that are incurred as a result of allowing threads to be distributed across the cluster for parallel execution. The overheads are mainly due to remote object accesses and distributed thread synchronization.

Remote object access overhead. Because it is possible to have two or more threads to update the same object at the same time, the DSM's *lock* and *unlock* primitives are used for data consistency control. According to our observation, the overhead introduced by these primitives can be substantial if object updates are frequent. Besides, when a thread accesses an object that is in a dirty memory page, or if the object is not already cached, the DSM subsystem will have to fetch the page from a remote node, introducing access delay.

In general, there are three types of memory access in JESSICA:

- Local stack data access—The variable involved is local to a method or a block of code. It is allocated from the Java method stack rather than from the DSM. For example, the `iload_0` instruction, which loads an integer onto the top of the method stack, is an access to the local stack data.

- Local object data access—The variable involved is a field of a local object. The field variable is allocated from the DSM and the data concerned resides in the same machine as the thread that is making the access. The bytecode execution engine uses the `GETFIELD` and the `PUTFIELD` instructions to access the object field. This kind of access is indirect as the memory location of the data field has to be computed first by the bytecode execution engine. This is done by adding the object address to the offset at which the field variable is stored.

- Remote object data access—This is similar to local object data access except the thread that is making the access is located in a node different from where the object data is stored.

To study the performance differences of various types of memory access, we have performed a series of experiments to measure the time required to *update* some selected elements of a very large array that spans 4096 shared memory pages. The elements are selected in such a way that in the DSM they are 4K bytes apart pairwise, *i.e.*, the size of a shared page. As a result,

```
Class Foo {
  native void startTimer(); // a timer with microsecond resolution
  native void endTimer();

  int a[];
  public void run() {
    startTimer();
    a[0] = 1;
    a[1024] = 1;
    a[2048] = 1;
    a[3072] = 1;
    a[4096] = 1;
    a[5120] = 1;
    a[6144] = 1;
        ...

    endTimer();
  }
  Foo() {
    a = new int [1024*4096];
  }
....
```

Figure 10: Sample code of class `Foo` for measuring object access overhead

every remote update to each of the elements will need to fetch a new shared page containing the element from the remote node. The sample code of the program is shown in Fig. 10.

The ratio of access overhead is found to be:

$$\begin{array}{lclclcl} \text{remote object data} & & \text{local object data} & & \text{local stack data} & & \\ \text{access time} & : & \text{access time} & : & \text{access time} & = & 2322 : 23 : 1 \end{array}$$

In other words, the overhead of remote object access is about 100 times that of local access. The difference is due to the transmission of DSM pages from remote nodes through the network. Note that this is a worst-case result as the update will cause the whole page of data to be received as a DSM diff. In general, the size of diff varies and not all updates will trigger the transmission of diffs. It is possible that the current thread is the only one to make the update between successive invocations of DSM-unlock so that no diff will be generated by other nodes. The 23 times difference between the access time for local object access and that for local method stack access is because of the overhead produced by the DSM-lock and unlock operations as performed by the `iastore` instruction, although no diffs will be received in this local case.

Cooperative semaphore overhead.

We compare the time for a migrated thread to perform cooperative semaphore operations with one without migration. Consider when a slave thread tries to acquire a cooperative semaphore, as shown in Fig. 11. It sends a *semaphore acquire message* to its master ($T0$).

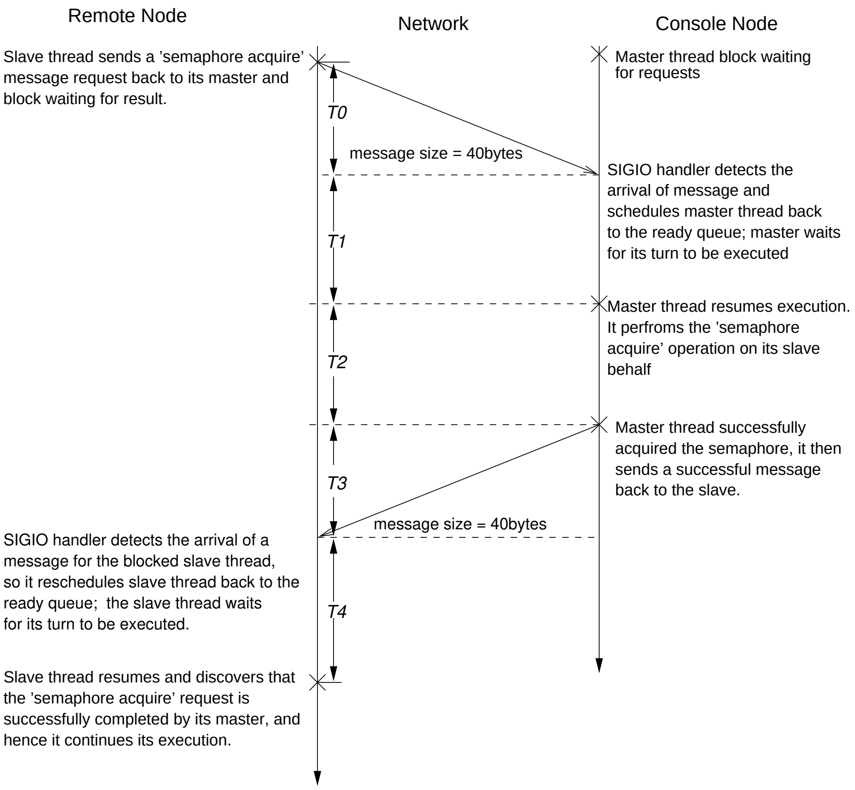


Figure 11: Cooperative semaphore in action: a migrated thread performs an acquire operation

The message will trigger a `SIGIO` signal when it arrives at the console. With the help of a `SIGIO` handler, JESSICA will then notify the TM that some data is ready for the master thread to read. As a result, the master thread is rescheduled back to the ready queue. Notice that the master thread may not be able to resume execution immediately because there may be other threads currently waiting in the ready queue in front of it. Assume that after a while ($T1$) it is the master's turn to execute, and the master acquires the semaphore ($T2$). After the semaphore is acquired, the master sends a *success* message to the slave, prompting the slave thread to resume ($T3 + T4$) its execution. From Fig. 11, it can be seen that the total time for a slave thread to acquire a cooperative semaphore is equal to $T0 + T1 + T2 + T3 + T4$, while that for a local thread to acquire a semaphore is simply $T2$. In other words, the extra overhead in this case is $T0 + T1 + T3 + T4$.

We have conducted a series of experiments to measure the time taken for a migrated thread to acquire a free cooperative semaphore remotely and the time taken for a local thread to acquire a free semaphore locally. A free semaphore is a semaphore that is not currently held by anyone and so a thread can acquire it immediately. In this case, the value for $T2$ will be the smallest. By our design, both the master and the slave thread are the only active threads running in their respective nodes; therefore the time to wait before resuming execution, *i.e.*, $T1$ and $T4$, would be zero. It is found that the time it took to acquire a remote cooperative semaphore for a slave thread this way is about 261 microseconds. For the case of a local, non-migrated thread, the time is approximately 7.78 microseconds. Hence, the ratio of the time required to acquire a free semaphore remotely to that for the local case is about 34 : 1. By similar arrangement, we were able to determine the time for releasing a semaphore both remotely and locally. The result shows the corresponding times are about the same: it took about 258 microseconds to remotely release a cooperative semaphore and 7.81 microseconds to release a local one.

It can be seen that a major portion of the cooperative semaphore overhead comes from the need to send control messages between nodes and from the operating system invoking the `SIGIO` handler. A point to note is that the overheads measured here are minimum values. In general, it will take some time for a thread to resume execution after it is rescheduled since there could be other threads, with either the same or higher scheduling priorities, already running in the same node. Moreover, a semaphore may not always be available immediately when a thread tries to acquire it. Hence, $T1$ and $T2$ could be larger. For example, in our recursive ray-tracing application, where threads are tightly synchronized, it is found that the time to acquire a cooperative semaphore increases from 5.49 to 10.29 milliseconds when more worker threads are migrated.

4.5 Analysis of Migration Latency

Fig. 12 shows in detail the interactions between the console and the worker node when a thread is migrated from the former to the latter. The migration latency is the time between the moment the migrating thread is frozen by the console and the moment it is restarted later as a slave thread at the worker node. Let $T0$ be the time taken to notify the destination node and to have the destination node prepare itself for the migration. The value of $T0$ is relatively constant. Let $T1$ be the time taken to marshal a delta set at the console node, to send the marshaled data across the network, and eventually to de-marshal the received data at the destination node. The

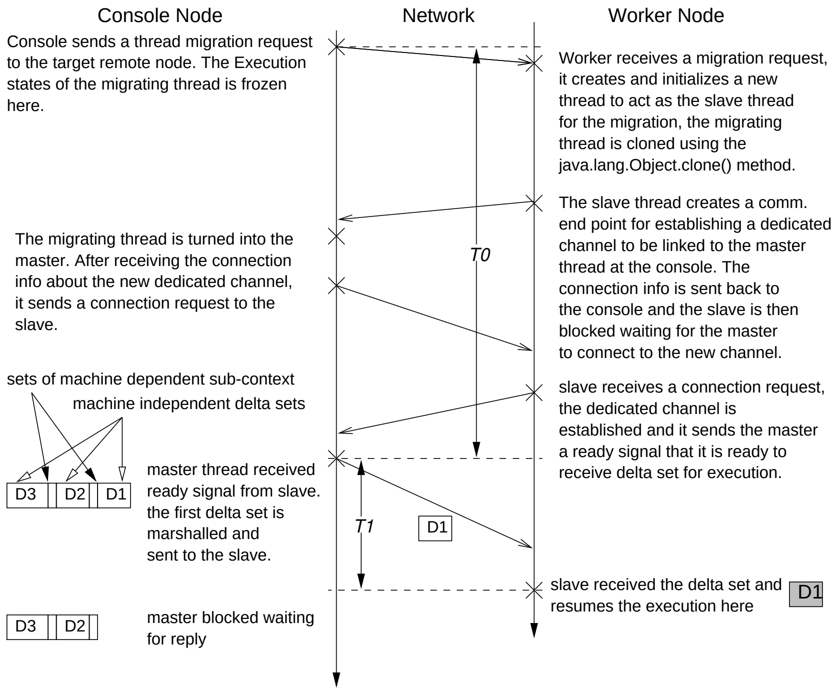


Figure 12: Interactions between the console and the worker during migration

value of $T1$ is therefore proportional to the size of the transferring delta set. The migration latencies, *i.e.*, $T0 + T1$, for different sizes of the delta sets are measured.

According to the data collected, when the size of the delta set is zero, the migration latency is about 27.91 milliseconds. $T0$ includes the time taken to execute the `java.lang.Object.clone()` method in the worker node as well as the time for sending the four handshake messages between the console and the worker node, as shown in Fig. 12.

The purpose of the `clone()` method is to create an image of the migrating thread at the destination node, which will then become the *slave* thread. Notice that in general, a thread is not allowed to invoke its `clone()` method unless the thread implements the `java.lang.Cloneable` interface. However, since the `clone()` method is invoked directly within the virtual machine, JESSICA would bypass the checking of whether a thread implements the `Cloneable` interface or not. As a result the invocation will always be successful and the SSI illusion will not be compromised. If a thread does in fact implement the `Cloneable` interface, then it will be duplicated in a way according to its `clone()` method; otherwise, the default implementation of the `clone()` method in JESSICA is to perform a `memcpy()` to duplicate the thread object byte-by-byte.

```
class DeltaE extends Thread {
  int i;
  DeltaE() { i = 100; }

  public void run() {
    g();
  }

  void g() {
    i--;
    if (i > 0) f(i);
    else autoMigrate(); // trigger a migration here ...
  }
  native void f(int i); // a native function that invoke g()
  native void autoMigrate(); // a native function that tells
                             // Migration Manager to migate
                             // the currentThread to a worker
                             // node
}
```

Figure 13: Implementation of class `DeltaE` in Java

A further breakdown of this $T0$ value reveals that the time required to invoke the `clone()` method is about 6.76 milliseconds. The time required to set up a TCP connection between the master and the migrated slave thread and that for sending the four handshake messages make up the remaining time. Now consider the case when a thread is migrated just before it starts executing the first instruction; the size of the smallest possible delta set, which contains no local variable or stack data, is 208 bytes. Consequently, the minimum migration latency of the delta execution mechanism is measured to be about 29.79 milliseconds.

The cost of delta execution. We have devised a test program based on the `DeltaE` class as shown in Fig. 13 and Fig. 14 to study the effect of machine-dependent code on thread migration and the cost of delta execution. There are two methods `f()` and `g()` defined in class `DeltaE`, which recursively call each other until the counter `i` reaches zero. In addition, the native method `f()` would print the level of recursion to `stdout` before returning. The function `autoMigrate()` is a special native function defined in JESSICA which will cause the Migration Manager to migrate the current thread to a worker node.

When an instance of `DeltaE` is instantiated, the thread will recursively invoke method `g()` and method `f()` until `i` reaches zero. `autoMigrate()` will then cause the `DetalE` thread to be migrated to a worker node. At this point the execution context of the thread should contain a chain of delta sets interleaved by sets of machine-dependent state, as shown in Fig. 15. By

```
void DeltaE_f(struct HDeltaE* this, int i) {
    do_execute_java_method(0, (Hjava_lang_Object*) this, "g",
                           "()V", 0, 0); // invoke g again

    fprintf(stdout, "hello: %d\n", i); // print the current
                                       // level of recursion
}
```

Figure 14: Implementation of native method `DeltaE.f()` in C

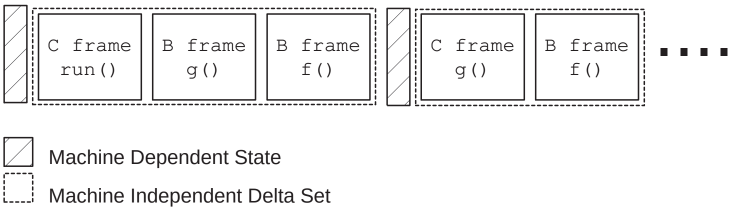


Figure 15: Execution context of an instance of the `DeltaE` thread

the time the migrated thread resumes its execution at the worker node, it will continue from the point of return of `autoMigrate()`, which is also the point of return of method `g()`. From this point onwards, the effect of delta execution will cause the execution control to bounce back and forth between the console and the worker node. At the console node the current level of recursion will be printed to `stdout` as a set of machine-dependent states is executed, while at the worker node the control will complete the execution of method `g()` as the next *delta set* is shipped there.

In our experiment the number of recursion was set to 100 and it took 2037 milliseconds to execute the test program. The time spent was mainly on the shipping of delta sets as well as the bouncing of execution control between the console and the worker node for a 100 times. In the case where migration was disabled, the time spent to execute the test program was found to be 18 milliseconds. Hence, the round-trip overhead for each bouncing of control between the console and the worker node due to delta execution is about 20.19 milliseconds.

5 Related Work

5.1 Solaris MC

Solaris MC [16] is a prototype distributed operating system for running in a cluster of computers. It extends the existing Solaris operating system to provide a single-system view. Thus, a cluster 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.

The SSI boundary created by Solaris MC can be considered a *global process space* that spans the entire 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 of JESSICA which hides the physical boundaries between machines. The global thread space of JESSICA supports remote creation of threads, and system services are also transparently redirected to the appropriate location. While threads living in the global thread space of JESSICA can freely migrate node to node within the cluster, processes in the global process space of Solaris MC cannot.

Solaris MC leaves a shadow virtual process behind when a process migrates. It is similar to JESSICA which leaves a master thread behind in the console node. They both provide migration transparency by redirecting location-dependent operations and messages. Other Solaris MC ideas can provide good input to future versions of JESSICA; for example, the idea of a global network subsystem and the use of a packet filter for network data redirection could be incorporated into the `java.net` class library.

5.2 cJVM

cJVM [15] is a cluster-aware Java Virtual Machine implementation which provides a single-system-image illusion over a cluster of computers. The cJVM approach focuses on Java semantic information obtainable from the virtual machine level, which is used as a basis for exploiting possible optimization strategies in the implementation. Instead of using a cluster-enabled infrastructure, such as a distributed shared-memory (DSM), cJVM maintains a distributed heap by using the master-proxy model for object creation and the method shipping technique for transparent remote object access. cJVM has its own particular distributed thread model for parallel execution of threads across the cluster. For a Java object which is passed as reference to a remote node, a proxy of that object will be created at the target node. When the remote node accesses the proxy later, the proxy will be responsible for forwarding the execution flow back to the original node in which the master object resides, where operations on the object can then be performed. Load balancing over the cluster is conducted by means of remote thread creation. The master-proxy object model offers an opportunity for providing *smart proxy* on a per-object-instance basis. By employing code analysis, an object proxy can decide how to handle a remote object access request in the most efficient manner, such as forwarding the execution flow back to the node where the master object resides or caching the data so that the request is handled locally, depending on the nature of the access.

The JESSICA approach is different from that of cJVM in that it employs a DSM subsystem to establish a global object space for object sharing across the cluster. Object consistency is transparently handled by the cache coherent protocol as implemented in the DSM subsystem. However, the possibility of false sharing and the extra overhead of synchronization primitives can have a considerable impact on the performance of the JESSICA prototype, as has been demonstrated in the matrix iteration example. On the other hand, although the master-proxy object model used in cJVM saves the use of explicit synchronization primitives when accessing objects, the cJVM implementation is still required to send remote access requests when a bytecode instruction tries to access heap data that is located in a remote node. The remote access and data caching granularity in the cJVM case is an object while that for JESSICA is a page. Hence, the effectiveness of the two approaches will depend on the pattern and frequency of remote object accesses for cJVM, and the cache coherent protocol for JESSICA.

Because the method shipping technique of cJVM forwards execution flow of an active thread to the node where the target master object is located, load distribution across the cluster is therefore largely dependent on the placement of distributed objects across the cluster. Consequently, load distribution across the cluster can fluctuate as the execution flow moves from one node to another. Thus, an effective load balancing strategy is necessary for cJVM to decide on which node a newly created thread should be placed in order to maintain an evenly distributed load across the cluster. Whereas in the case of JESSICA, the thread migration capability provides a more flexible means for performing load balancing. For example, when the migration manager detects heavy DSM traffic, it is possible to improve locality by migrating threads towards their target objects to which accesses are frequent, after taking other system load factors into account.

5.3 Java/DSM

Java/DSM [14] is a distributed Java Virtual Machine that runs on a cluster of heterogeneous computers. It provides an illusion to Java applications as if 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. With the help of the strong-typing characteristic of Java, Java/DSM is able to make use of a translation mechanism so that when data of the same type are retrieved from computers of different hardware platforms, they will be converted to a common format before interpretation.

Both Java/DSM and JESSICA follow a similar approach by implementing a distributed virtual machine at the middleware 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.

Table 2 provides a comparison between JESSICA and the related systems discussed in this section.

	Level of Approach	Method of Load Distribution	Implementation Techniques	Characteristics	Support Migration/SSI Transparency
JESSICA	Middleware	Thread migration	delta execution + DSM + Message redirection by helper threads	Global thread space offers a compatible and parallel execution environment where applications can run without modification and gain speedup	Yes
Solaris MC	Monolithic kernel	Remote execution of process	Message redirection by vproc + CORBA + packet filtering	Commercial OS with contemporary software technique	Yes
cJVM	Middleware	Remote execution of thread	Master-proxy object model + method shipping	Use Java semantic available from the virtual machine to exploit optimal implementation strategy	Yes
Java/DSM	Middleware	Remote execution of thread	DSM + data translation	A distributed Java Virtual Machine runs on a heterogeneous cluster	No

Table 2: Comparison of characteristics between JESSICA and related systems

6 Conclusion

The JESSICA prototype provided a parallel execution platform with respectable speedup for all the experimental applications tried. It is able to support dynamic thread migration, to achieve SSI, and has been shown to be a parallel execution environment with good efficiency. Efficiency we measured ranges from 41.95% to 100% when using two to eight nodes in a Linux cluster.

JESSICA takes a novel approach in dealing with thread migration when compared to other systems. Instead of moving the whole execution context to the destination all at the same time, the execution context is separated into machine-dependent and machine-independent parts, and only the machine-independent parts are migrated in a regulated manner. This design imposes no limitation on the type of thread that can migrate, and whether they own location-dependent resources or not.

Although it is true that the master-slave design for supporting migration transparency can make the console node a potential bottleneck, the centralized design allows control state to be maintained at a single location which reduces implementation complexity. On the contrary, a decentralized approach would require the control state to be distributed across the cluster, requiring a much more complicated implementation. The complexity stems from the need to coordinate all the nodes to support distributed thread scheduling, migration of communication endpoints, forwarding of inter-thread signals, etc., which all translate into significant amounts of messages to be sent across the cluster in order to maintain the consistency of the distributed control state.

Our experiments have shown that the major overheads had come from remote object accesses made by the migrated threads as well as distributed thread synchronization. This is because the DSM subsystem we used in our current implementation follows the lazy release consistency model, which would require a pair of DSM `lock()` and DSM `unlock()` operations to be performed for every object access. At the BEE level, the lack of knowledge about the application has made it difficult for the execution engine to intelligently organize object accesses so that the number of `lock()/unlock()` operations may be reduced. Using a DSM that follows the lazy release consistency model is clearly not the best choice for providing location-transparent object access to threads in JESSICA. Work is underway to replace the current DSM by a more efficient object-based DSM.

Our experiences with the JESSICA prototype have led us to conclude that establishing an SSI illusion using the middleware approach to support parallel execution of multi-threaded Java programs in a cluster environment is feasible and beneficial. The design of the Java programming language has not included any restriction that would hinder the use of the middleware approach. On the contrary, it is the characteristics of the language, such as bytecode execution, threads as first-class citizens, and the simple model of inter-thread signaling and synchronization that have simplified the implementation of JESSICA. Consequently, the JESSICA implementation has not added any new features to the Java programming language. Moreover, the design of delta execution opens up a substantial ground for further development of thread migration in a heterogeneous environment as all the states migrated using delta execution are platform-independent. Overall, the JESSICA approach has proved to be a simple, flexible, and portable solution for realizing the goal of single-system-image.

Acknowledgements

This research was supported in part by Hong Kong RGC grants (HKU 7032/98E and HKU 7025/97E) and a HKU equipment grant (10003.01991001.00000.14200.100.01). The authors are grateful to the anonymous referees for the very useful comments they made.

References

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

[2] R. Buyya (ed.), *High Performance Cluster Computing: Programming and Applications*, Vol. 2, Prentice-Hall, 1999.

[3] D. Plainfosse and M. Shapiro, "A Survey of Distributed Garbage Collection Techniques", *Proc. of 1995 International Workshop on Memory Management*, Sep 1995.

[4] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Software Practice and Experience*, Vol. 21, No. 8, Aug 1991.

[5] K. Arnold and J. Gosling, *The Java Programming Language*, Addison Wesley, 1996.

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

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

[8] Javasoft, Java Remote Method Invocation—Distributed Computing for Java, white paper, *http://java.sun.com/marketing/collateral/javarmi.html.*

[9] M.J.M. Ma, C.L. Wang, and F.C.M. Lau, "Delta Execution: A Preemptive Java Thread Migration Mechanism", *Cluster Computing*, to appear.

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

[11] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1996.

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

[13] 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.

[14] W. Yu and A.L. Cox, "Java/DSM: A Platform for Heterogeneous Computing", *Proc. of ACM 1997 Workshop on Java for Science and Engineering Computation*, Jun 1997.

[15] Y. Aridor, M. Factor, and A. Teperman, "cJVM: a Single System Image of a JVM on a Cluster", *Proc. of 1999 International Conference on Parallel Processing*, Sep 1999.

[16] Y.A. Khalidi, J.M. Bernadbeu, V. Matena, K. Shirrif, and M. Thadani, "Solaris MC: A Multi-Computer OS", *Proc. of 1996 USENIX Annual Technical Conference*, pp. 191–294.

Authors' Biographies

Matchy J.M. Ma received his B.Sc. and M.Phil. degrees in Computer Science from the University of Hong Kong in 1994 and 1999 respectively. He is currently a software engineer at the Advanced Academic Java Campus of the University of Hong Kong. His research interests include Internet computing, object-oriented and distributed systems, and cluster computing. He can be reached by email at jmma@csis.hku.hk.

Cho-Li Wang received his B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1985. He obtained his M.S. and Ph.D. degrees in Computer Engineering from University of Southern California in 1990 and 1995 respectively. Currently, he is an assistant professor in the Department of Computer Science and Information Systems at the University of Hong Kong. His research interests include high-speed networking, cluster computing, and distributed Web server systems. Wang is a member of the advisory committee of the IEEE TFCC (Task Force on Cluster Computing). He can be reached by email at clwang@csis.hku.hk.

Francis C.M. Lau is the head of the Department of Computer Science and Information Systems at the University of Hong Kong. He has a B.Sc. degree from Acadia University, and an M.Math. and a Ph.D. degree from University of Waterloo. His major research interests include parallel and distributed systems, operating systems, and Internet and Web computing. He is an active volunteer of the IEEE Computer Society, serving as a vice president in 1999. He can be reached by email at fcmlau@csis.hku.hk