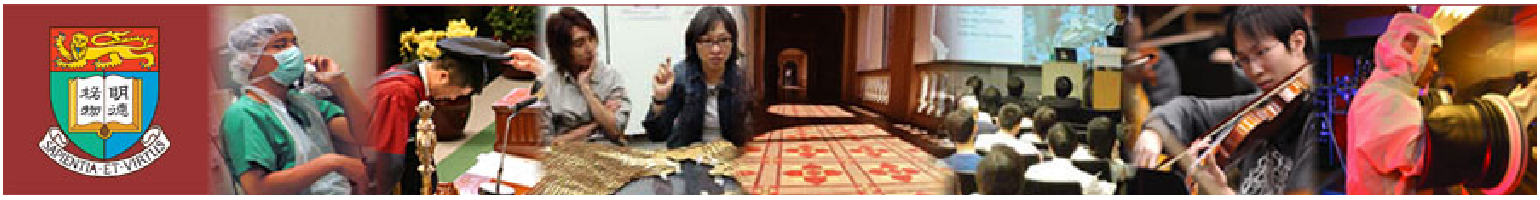


Cluster Computing with Single Thread Space

Francis Lau, Matchy Ma, Cho-Li Wang, and Benny Cheung

Abstract—**To achieve single system image (SSI) for cluster computing is a challenging task since SSI is a form of complete transparency that requires the integration and unification of all types of resources in a cluster. In this paper, we propose a new Java computing platform with the concept of *Single Thread Space*, which is a true parallel computing environment for performing a multi-threaded Java application on a cluster environment. Threads running within this space would share all the resources that each thread has created or allocated and they view the underlying cluster as a single computing system. We realize the single thread space based on a middleware developed at a virtual machine level, which allows application program to create as many threads as needed. The middleware can automatically distribute the executing threads across the cluster to exploit the maximal parallelism and to optimize the overall resource utilization. The implementation is compatible with the standard JVM and does not need any low-level or platform-specific supports. Thus it is portable across different hardware platforms.**

Index terms—**Cluster computing, single system image, dynamic load balancing, thread migration, Java Virtual Machine, JESSICA.**

I. Introduction

A cluster of computers is a federation of computers linked by an interconnection network where the computers run integration software to support collaborative computations [2,4,9,13]. The integration software provides an abstraction layer that hides the physical boundaries between machines and makes the cluster appear as a single computer to applications - a single system image (SSI).

SSI represents a complete form of transparency which is 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. By integrating distributed cluster resources and providing a unified naming scheme, the single system illusion can be achieved for different types of cluster applications.

For example, there are various job dispatch systems for cluster. Using a *global job scheduler*, a user job can be submitted from any node to request any number of host nodes to execute it. Concurrent job scheduling is possible either in batch, interactive, or parallel modes. The SSI can also be achieved in the access of I/O devices in the cluster [15]. A uniform device naming can be adopted so that user applications at different machines is able to view and access all the devices connected to the cluster as they access the local devices, even the devices are physically attached to a node different from the one on which the application is running. Several distributed operating systems support the concept of a *global process space* where all processes created in the cluster share a uniform process identification scheme. A process on any node can be created on (e.g., through a Unix fork) or communicate with any other processes (e.g., through signals, pipes, etc.) on any remote nodes.

In this paper, we propose establishing a single-system-image illusion over a cluster as a means to bridge cluster computing and Java's multi-threaded programming model. The SSI illusion is realized through the provision of a *single thread space*, which is a global execution environment for running threads that extends across the entire cluster. It supports parallel execution of multi-threaded applications. A multi-threaded Java application on any node can create threads to run at difference nodes. All the threads share a uniform thread identification scheme. In addition, threads running within this space could freely move between machines during its execution. They 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 single thread space illusion is established at the middleware level in the form of a distributed Java Virtual Machine (JVM). This approach does not require any modification to the underlying operating system or to the Java applications running on top. It guarantees portability over various popular operating systems and compatibility with existing Java applications.

The concept of single thread space was realized in our JESSICA system. 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. With the single thread space support, 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 and to optimize the overall resource utilization. JESSICA supports preemptive thread migration which allows a thread to freely move between machines during its execution, and global

The authors are with the Department of Computer Science and Information Systems, the University of Hong Kong, Hong Kong. E-mail: {fcmlau+jmma+clwang+wlcheung}@csis.hku.hk. This paper is an extended version of [18].

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.

The rest of the paper is organized as follows. Section 2 presents the concept of single thread space. Section 3 discusses the design and implementation of the single thread space. Section 4 evaluates the performance of our prototype. Section 5 surveys other works related to our work. We conclude by summarizing our experiences in Section 6.

II. SINGLE THREAD SPACE

In recent years, the multi-threaded programming model has grown increasingly popular because of the availability of SMPs and the wide spreading of the Web-based applications such as the Web browsers and Web servers [6]. However, due to the limited scalability of SMP architecture, it hinders the development of a large-scale application that need scalable computing power. An ideal solution is to use the cluster as an new execution environment for the multi-threaded application, where the application program can create as many threads as possible and these threads are able to map to different processors in the cluster for true parallel execution. In addition, threads could freely move between machines during its execution to achieve fault tolerance or load balancing.

We define the *single thread space* as a cluster computing environment for performing multi-threaded application that can extend its execution across the entire cluster. Threads running within this space would share all the resources that each thread has created or allocated and they see the underlying cluster as a single computing system with multiple processors – single system image. Figure 1 shows the concept of single thread space that provides a single system illusion among all the threads created by a process.

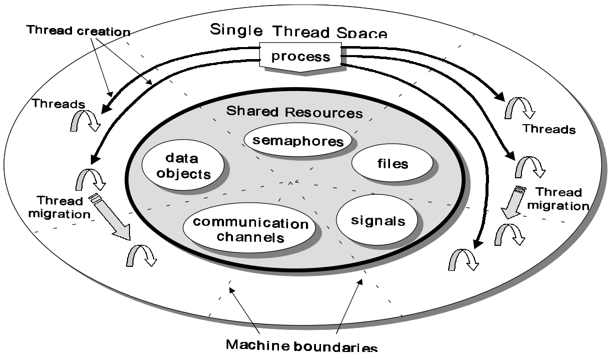


Fig. 1. Single thread space provides an SSI illusion over a cluster.

In order to move a thread to a different processor, it is necessary for the transferred thread to correctly access all related resources and let all the threads share a uniform thread identification scheme. Data local to the thread (i.e., stack and thread local heap) may be copied to the destination. However, since the addresses on the target machine may be different from the original addresses, internal data references may no longer be valid. A system with thread migration requires integration solution that allows all threads to share all the resources (such as files, communication channels, data objects, etc.) as if they are not migrated. In addition, a thread may access data shared by multiple threads such as synchronization objects. After the migration, it is necessary for all the threads to locate the object and provide synchronization mechanism (e.g., signals, semaphores) to allow correct access.

A. Our Design

In our design, the single thread space illusion is established at the user level in the form of a middleware that enables the execution of multiple Java threads among cluster nodes. A single global thread space is constructed through the services of three important subsystems (1) the *Delta Execution* subsystem for supporting preemptive thread migration, (2) the *Master-Slave Redirection* subsystem for supporting location-transparent operations, and (3) a distributed shared-memory (DSM) subsystem that creates a global object space for supporting remote object access. Figure 2 shows an overview of the design of the single thread space architecture.

We classify a cluster node as either a *console* or a *worker* node. The console node of an application is the node in which the application is first instantiated, i.e., the application's home. Worker nodes are the other nodes that house one or more migrated threads created by the application.

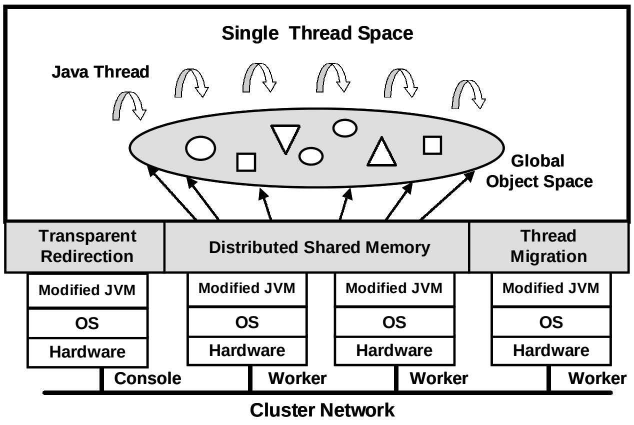


Fig. 2. Single thread space is supported by three subsystems that are provided by a set of cooperative JVMs.

These worker nodes play a subordinate role to the console node by serving requests directed from the console. The core design of single thread space is based on a *master-slave* approach for thread migration, where the cooperation between the *master* thread running on the *console* node and the *slave* thread running on the *worker* node together produce the required transparency. With this master-slave design we are able to implement transparent network communication and file operations, distributed thread synchronization and remote exception.

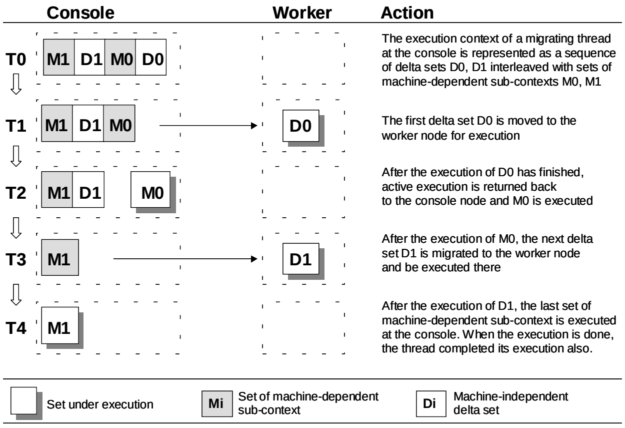
The single thread space extends the parallelism of a JVM that spans over a cluster without changing the semantics of runtime interactions between objects, therefore making all existing multi-threaded Java programs able to run on the cluster. Three main subsystems to support single thread space are discussed in the following sections.

B. Thread Migration: Delta Execution

Thread migration is usually established as a mechanism for achieving dynamic load sharing. However, such a fine-grain migration (as compared with process migration) has not been used due to the high thread and messaging overheads [7].

Delta execution [5] is a preemptive thread migration mechanism for supporting transparent thread-to-processor mapping within the single thread space. 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 virtual machine level, migration is therefore transparent to Java applications and no migration-specific code needs to be added to the applications.

In general, the execution context of a Java thread consists of both machine-independent and machine-dependent sub-contexts. Machine-independent sub-context refers to the migratable state information that can be expressed in terms of the high-level execution state of a JVM, such as data stored in the virtual machine's registers. Machine-dependent sub-context is the non-migratable state information that is part of the low-level execution state of a JVM implementation, such as the hardware return address stored in the execution stack of an internal function invoked that implements the `iadd` bytecode instruction. As illustrated in figure 3, a thread's execution context consists of sets of machine-independent sub-contexts, also known as delta sets, which interleave with the sets of machine-dependent sub-contexts. In our design, migration granularity is per-bytecode-instruction where a thread can be preempted and migrated once execution of the current bytecode is completed. 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.



The execution context of a migrating thread at the console is represented as a sequence of delta sets D0, D1 interleaved with sets of machine-dependent sub-contexts M0, M1

The first delta set D0 is moved to the worker node for execution

After the execution of D0 has finished, active execution is returned back to the console node and M0 is executed

After the execution of M0, the next delta set D1 is migrated to the worker node and be executed there

After the execution of D1, the last set of machine-dependent sub-context is executed at the console. When the execution is done, the thread completed its execution also.

Fig. 3. Delta Execution in action.

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

As illustrated in figure 3, 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. With the support of delta execution in a single thread space, it is possible to dynamically relocate the 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. Another worker node will be selected 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 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.

C. Global Object Space

After thread migration, the migrated thread should be accessible by the same name and mechanisms as if migration never occurred. The same applies to all resources used by the Java threads created in a program, such as the shared objects. The *global object space* (GOS) provides location-independent object access. It is yet an SSI layer created for all distributed threads to view a single and unified shared object space to ease the implementation of the single thread space.

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 only. Other internal state of a JVM runtime is stored locally in the corresponding UNIX process' stack and local-heap memory. Contents of objects that are located in a remote node are cached by the DSM subsystem, and GOS 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. GOS 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 mutual exclusion control primitives provided by the DSM subsystem are used to ensure object consistency. Distributed garbage collection is built on top of the DSM layer to locate all the unused objects and to reclaim their space. In current GOS, a distributed mark-and-sweep garbage collection mechanism is installed [3].

D. Transparent Redirection

In this section, we discuss various redirection mechanisms developed in the single thread space to support post-migration services, including the *cooperative semaphore, remote thread signaling, remote exception, and location-transparent I/O*.

The distinctive feature of multi-threaded Java 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 the shared resources. 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. In the single thread space, to achieve efficient thread synchronization and data integrity, a *distributed semaphore* and *remote thread signaling* should be supported. Besides, we need to support *location-transparent I/O* so that all opened files and network communication channels following a migration will remain functional.

We adopted a master-slave approach to achieve the above functions. After a thread has been migrated, the master thread remaining at the console represents the original thread. All the thread-level synchronization 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 I/O and thread synchronization operations are redirected back to the console. 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 without thread migration. Consequently, such an execution environment as observed by a running thread is the same as that of a standard JVM.

A decentralized approach is used to implement distributed semaphore. In each cluster node, a Thread Manager (TM) is created and it is responsible for thread creation, scheduling, and termination. TMs support distributed synchronization of migrated threads by having the runtime system to forward semaphore operations back to the console TM. 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. We called this *cooperative semaphore*.

Java threads rely on simple wait-notify signals for inter-thread communications. By a design similar to that for cooperative semaphores, we have designed 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.

The Java programming language supports a language exception construct where a method can include a block of code, called the *catch block*, for handling any specified exceptions that may be generated as the method executes. The idea of language exception is that when the current method execution has generated an anticipated error, the execution can be aborted and the context be rewind back to a point where the program has pre-defined a specific catch block to deal with the error. As a result, to implement language exception, the method invocation sequence of a thread's execution context is scanned from the tail towards the head to locate the nearest called method which has implemented the corresponding catch block to handle the exception. When this method is located, the execution context of the current thread is rewind up to this method so that execution can continue from the catch block.

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

In our design, location-transparent I/O support is also 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. Implementation details are discussed in Section III.D. The object-oriented nature of our design 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.

III. System Architecture and Implementation

We have realized the single thread space in the JESSICA system which was running on a Linux PC cluster. The experimental platform 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.

A. The Building Components

The implementation of the JESSICA system is in the form of a distributed virtual machine and there will be a JVM daemon process running on each node of the cluster. These daemon processes execute as user-level processes on top of the UNIX operating system. A JESSICA daemon is composed of the following four components:

- *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.

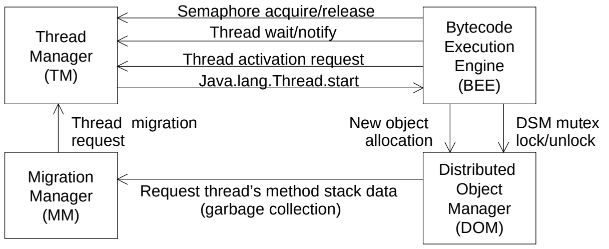

Thread Manager (TM)
Semaphore acquire/release
Thread wait/notify
Thread activation request
Java.lang.Thread.start
Bytecode Execution Engine (BEE)
Thread migration request
New object allocation
DSM mutex lock/unlock
Migration Manager (MM)
Request thread's method stack data (garbage collection)
Distributed Object Manager (DOM)

Fig. 4. Interaction between system components in JESSICA

Figure 4 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, 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 the thread. TM will bind the thread to a new execution context and insert it into the thread scheduling subsystem. Later on when the thread is scheduled to run, TM binds it to BEE for executing the thread's start() method. During it'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 mutual exclusion control primitives provided by DOM. It can also let the thread 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.

The current implementation of JESSICA is based on version 0.9.1 of the Kaffe virtual machine [8] 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 features. For example, in order to facilitate the extraction of a thread's execution context, the method invocation mechanism in Kaffe's BEE 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, DOM 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 mutual exclusion control primitives provided by the DSM whenever necessary. Moreover, the thread subsystem has been extended to become the TM for supporting thread migration, cooperative semaphore, and the remote signaling mechanism. Finally, MM responsible for enforcing a load balancing policy has also been incorporated into the system. MM obtains load information from the process file system `</proc>` of each node and interacts with other MMs of other cluster nodes to make migration decision. All communications between the JESSICA daemons are through the BSD socket interface provided by the Linux operating system.

B. *Preemptive Thread Migration*

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

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

Note that the 7th step of the above scheme is the critical step for the system to maintain migration transparency, it is in this step where redirections take place. While the slave thread is executing the delta set at the destination node, the master thread is also responsible to monitor the communication channel and see if there are any messages of redirection requests sent from the slave thread. These messages can be network channel read/write operations or mutex `lock()/unlock()` operations where the master has to perform on behalf of the slave back at the console node, so as to maintain the network and location transparency. In addition, the master is also responsible to redirect any asynchronous signals sending from other running threads on the console node, so that the original thread appears to other threads as if it was still running on the console node and the migration has never took place.

C. Dynamic Load Balancing

The current implementation relies on MMs running in all the worker nodes to provide load information across the cluster for making migration decision. Load information is obtained from the process file system `</proc>` of each node. MM 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 load information used in migration decision. If MM 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 sub-context. The MM 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, MM 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.

D. Transparent Redirection

The I/O 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, 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 implementation of cooperative semaphores ties closely to TM 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. 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 TM and appended to the queue. A thread unlocking the object later will decrement the count. When the count reaches zero, the first blocked thread from the queue will be scheduled to run by TM. 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 mutex `lock()` and `unlock()` operations are redirected back to the console as described below.

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, TM at the console can then reschedule some other thread that has previously issued a lock request for the semaphore.

IV. Performance Evaluation

In this section, we report performance results of the JESSICA system through the tests of various programs. We focus on the analyses of overheads incurred in basic operations, such as remote object access, cooperative semaphore, migration latency, and the cost of delta execution. Details related to the performance results of various applications can be found in [18].

A. Remote Object Access Overhead

We evaluate the efficiency of the global object space layer by analyzing the overheads that are incurred in remote object accesses and distributed thread synchronization. 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 global object space and accessed through the DSM.
- *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.
- *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 with different stride distance. The ratio of access overhead is found to be:

	remote object data data access time :		local object data access time :		local stack access time
=	2322	:	23	:	1

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 4KB page of data to be moved or the page `diff` to be calculated and shifted in the subsequent remote object accesses. 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. Note that 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.

B. 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, it sends a semaphore acquire message to its master which spends T0 time. 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 T1 time, it is the master's turn to execute, and the master acquires the semaphore which spends T2 time. After the semaphore is acquired, the master sends a success message to the slave, prompting the slave thread to resume (all together T3 + T4) its execution. T3 is the time to send the message to the slave, and T4 is the delay until the slave thread is rescheduled after the message has arrived. It can be estimated 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 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.

C. Migration Latency

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. The migration latencies, i.e., T0 + T1, for different sizes of the delta sets are measured, where:

- T0 is the time taken to notify the destination node and to have the destination node prepare itself for the migration. The value of T0 is relatively constant.
- T1 is the time taken to marshal a delta set at the console node, to send the marshaled data across the network, and eventually to de-marshal the received data at the destination node. The value of T1 is therefore proportional to the size of the transferring delta set.

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 handshake messages between the console and the worker node. 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. The default implementation of the `clone()` method in JESSICA is to perform a `memcpy()` to duplicate the thread object byte-by-byte.

A further breakdown of this T0 value reveals that the time required to invoke the `clone()` method is about 6.76 milliseconds. This includes the time to set up a TCP connection between the master and the migrated slave thread and that for sending the handshake messages. 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. The minimum migration latency is measured to be about 29.79 milliseconds. Further analyses of the delta execution overhead were studied in the next section.

D. Cost of Delta Execution

We have devised a test program based on a `DeltaE` class 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`. The native method `f()` would print the level of recursion to `stdout` before returning. The function `autoMigrate()` is a special native function 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, where *i* represents the number of recursion. `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 sub-contexts. By 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 sub-context 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 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, which is considerably efficient.

V. Related Works

In this section, we shall briefly discuss some other works related to thread migration, and the realization of a single system image.

cJVM [11] is a cluster-enabled implementation of a Java Virtual Machine that provides a single-system image of a traditional JVM while executing on a cluster. cJVM supports distributed access to objects using a master-proxy model. The node where an object is created contains the master copy, while proxies are used to other nodes to access the object. *Smart-proxy* is employed to allow multiple proxy implementations for a given class while using the most efficient implementation on a per instance basis. A single system image is maintained in some degree that applications are unable to distinguish between accessing the master of an object or its proxy. This is achieved by the use of a distributed heap, which also gives the application an illusion that the system is using a monolithic heap like traditional JVM does. Instead of thread migration like JESSICA, remote objects are accessed transparently through the method shipping technique, in which the proxy redirects the flow of execution to the node where the master copy of the object is located. Load balancing can be achieved in cJVM through remote thread creation, which distributes newly created threads according to a pluggable load balancing routine. It determines the best node on which the new instance of the runnable class is created. Because of the method shipping technique, load distribution across the cluster is therefore largely dependent on the placement of distributed objects across the cluster.

JavaParty [14] is built on top of Java RMI which extends Java as minimally and transparently as possible with a pre-processor and a run-time system for distributed parallel programming in heterogeneous environment. JavaParty provides a single system image by using a shared address space to support location-transparent remote access. Since JavaParty regards threads as objects of a thread class, remote threads can be created as objects of a remote thread class and migration of threads is also possible. Therefore, programmers need not deal with the mapping of remote objects or threads to specific nodes of the network. In object migration, if a remote object that implements an instance part is moving to a different host, a proxy is left behind. If a method call arrives at the proxy, a *MovedException* is thrown back to the caller. With the exception, the caller is informed about the new location of the moved object. The JavaParty compiler and run-time system deals with the locality and communication optimization. The runtime system offers distribution strategies, which are used when new objects are created. It can also schedule object migration by moving an object to the position of the caller or to the location of a different remote object. JavaParty introduces the new class identifier *remote* for parallel execution of Java threads. Program code modifications are needed to turn a multi-threaded Java program into a distributed JavaParty program by identifying the classes and threads that should be spread across the distributed environment.

Solaris MC [12] provides the same ABI/API as Solaris, running Solaris applications without modifications. Solaris MC provides a *global process space* for global process management, so that the location of a process is transparent to the user. 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 single thread space of JESSICA. While threads living in the single thread space of JESSICA can freely migrate node to node within the cluster, processes in the global process space of Solaris MC cannot. Process migration in Solaris MC is intended only for planned shutdown of nodes to achieve high fault-tolerance. Both Solaris MC and JESSICA provide migration transparency by redirecting location-dependent operations and messages.

The objective of Java/DSM [10] is to support heterogeneous parallel computing that hides users from the difference in architecture and data format for different machines. The use of the DSM system transparently handles message passing details such as data replication, remote interface design and reference marshalling. Java/DSM allows a multithreaded Java program written for a single machine to run on a parallel platform with fewer changes than a system using Java RMI. Java/DSM provides a shared memory space for object allocation which is similar to the global object space in JESSICA. However, the single-system-image offered by Java/DSM is incomplete, as a thread's location is not transparent to the programmer, and thread migration is not supported.

VI. Conclusions

To achieve SSI for cluster computing is a challenging task since SSI is a form of complete transparency that requires the integration and unification of all types of resources in a cluster.

The single thread space approach is our solution to achieve single system image. With the support of single thread space, the whole cluster is encapsulated into a single computing system from the view of a multi-threaded Java application. 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. With the support of preemptive thread migration (delta execution), a Java thread can be preempted and migrated to another node at any time during its execution. Based on this thread migration technique, parallel execution of a Java application can be achieved by simply creating as many threads as needed. Threads are automatically redistributed across the cluster to exploit real parallelism. The global object space provides location-independent object access. The proposed redirection mechanism allows any location-dependent resources to be transparently accessible by a migrated thread. 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.

The implementation of JESSICA is at the middleware level and is compatible with the standard JVM. The implementation does not need any low-level or platform-specific supports. Thus it is portable across different hardware platforms. 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. Work is underway to replace the current DSM by a more efficient DSM which adapts better to the access patterns [16].

Overall, 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 single thread space approach has proved to be a simple, flexible, and portable solution for realizing the goal of single system image. In our future work [19], we plan to port the W3C's Jigsaw Web server on JESSICA, where a cluster of four 4-way SMP servers with Gigabit

Ethernet network [20] will be employed to support the execution. Additional SSI features such as the idea of a global network subsystem and more efficient network data redirection as supported in Solaris MC can be employed to let JESSICA achieve the ultimate goal of SSI – complete transparency.

VII. Acknowledgements

This research was supported in part by Hong Kong RGC grants (HKU 7032/98E and HKU 7025/97E).

VIII. 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] K. Hwang and X. Xu, "Scalable Parallel Computing: Technology, architecture, Programming", McGraw-Hill Book Company, Feb. 1998.

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

[6] Jigsaw - The W3C's Web Server, http://www.w3.org/Jigsaw/

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] G. Pfister. In Search of Clusters: The Coming Battle in Lowly Parallel Computing. Prentice-Hall, 1995.

[14] JavaParty project, University of Karlsruhe, http://wwwipd.ira.uka.de/JavaParty/

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

[16] B. Cheung, C.L. Wang, Kai Hwang; ``JUMP-DP: A Software DSM System with Low-Latency Communication Support,’’ the *2000 International Workshop on Cluster Computing - Technologies, Environments, and Applications* (CC-TEA'2000) June 26 – 29, 2000, Las Vegas, Nevada, USA.

[17] B. Alpern *et. al.*, “The Jalapeño Virtual Machine,” *IBM System Journal*, Vol. 39, No. 1, February 2000.

[18] M.J.M. Ma, C.L. Wang, F.C.M. Lau, ``JESSICA: Java-Enabled Single-System-Image Computing Architecture,’’ to appear in *Journal of Parallel and Distributed Computing*.

[19] The JESSICA 2 Project, The System Research Group, The University of Hong Kong, http://www.srg.csis.hku.hk/jessica.htm

[20] Wenzhang Zhu, David Lee, and C.L. Wang, “High Performance Communication Subsystem for Clustering Standard High-Volume Servers using Gigabit Ethernet”, *HPC Asia 2000*, May, 2000, Beijing, China.