Contents

Building a Global Object Space for Supporting Single System Image on a Cluster

BENNY W.L. CHEUNG, CHO-LI WANG, FRANCIS C.M. LAU

Department of Computer Science and Information Systems
The University of Hong Kong
{*wlcheung, clwang, fcmlau*}*@csis.hku.hk*

1.1 Introduction

Advances in computer technologies have turned personal computers and workstations into commodity products. By connecting PCs or workstations through a high-speed network, *clusters* [1] have emerged to be a cost-effective computing platform, with power comparable to expensive high-end SMP servers or even mainframes.

Clusters differ from mainframes in that the composing units in a cluster are stand-alone computers, which do not share any physical memory, and they are built without the intention of co-operating with each other. Therefore, we need to add to them mechanisms to enable their communication with each other in order to jointly work on a task. Ideally, users should have the illusion that the cluster is but a powerful single machine. This leads to the pursuit of a *single system image* (SSI) [2] in a cluster.

SSI is a supportive layer above the operating system, which can provide a wide range of services. These services support each other to create a single-system illusion for the user applications. Machines within an SSI cluster need to be able to communicate with each other transparently, and cooperate with each other to execute the user tasks, so that the computing resources are fully utilized. An ideal SSI cluster therefore would have to provide for *dynamic thread/process migration* and *load balancing*. In addition, the SSI support should present a *single memory address space* to users' programs. This requires implementing the capability that a machine

can bring in the memory contents it needs to access from the other machines, which is not easily achievable because of *memory consistency* issues, as multiple copies of the same variable can reside in the memory of different computers.

To support an object-based system, the single memory address space is also known as the *global object space* (GOS). The GOS is a vital ingredient for achieving SSI in a cluster, and its design can affect the overall performance of the cluster. This chapter discusses the design and implementation issues of GOS. We shall use the *JESSICA* system [3] as a case in point, and its GOS design will be presented in detail.

1.2   Overview of JESSICA

A cluster having SSI support is convenient to application users. But SSI needs to couple with good programmability in order to be effective. It is well known that parallel or distributed programming is more difficult than sequential programming. Many special programming languages have been invented. Programmers need to learn these languages from ground up and existing code can hardly be reused. Other models introduce additional operations and primitives into existing programming languages. Programmers need to insert special statements at specific points of the program to perform explicit data partitioning and movement. A better approach than all this would be to let the programmer use an existing programming language as it is, with little or no modifications, and yet the programming language is powerful enough to allow harnessing the full computing power of the cluster.

Since its introduction in 1994, the *Java* programming language [4] has been receiving unprecedented acceptance and support. It has become a major paradigm in Internet computing for a wide range of users and application developers. It is also used as the programming language for numerous academic research projects and studies. Early versions of Java, however, targeted mainly at single machines. To make Java programs capable of spanning multiple machines to exploit true parallelism, programmers have to tackle co-ordination problems between processing nodes at the Java application level themselves, through some IPC mechanism such as sockets. Later versions of *JDK* (Version 1.1 and up) provide supports for parallelism such as *object serialization* [5], *remote method invocation* (RMI) [6], and *object request broker (ORB)* [7]. They deal with co-ordination and co-operation of processes at the function call level, through some form of remote procedure call (RPC) mechanism. These supports alleviate the programming burden to some degree, but programming as such is still not as straightforward as programming for a single machine.

The *Java Enabled Single System Image Computing Architecture*, abbreviated JESSICA, is therefore proposed [3]. It is a cluster-computing solution providing SSI for executing multi-threaded Java applications in a cluster of PCs or workstations. JESSICA allows applications to exploit the maximum parallelism derivable from the cluster. In particular, JESSICA allows multiple Java threads in a Java program to be executed in parallel among multiple cluster nodes, and these threads can freely

move from one machine to another during runtime to achieve load balancing.

1.2.1 Main Features of JESSICA

JESSICA provides a number of features to support the realization of an SSI in a cluster:

- *Single system encapsulation*: SSI is supported through the provision of a global thread space (GTS), as shown in Figure 1.1. When an application is instantiated, the JESSICA system creates a logical thread space that spans the whole cluster for thread execution. The GTS supports a cluster-wide thread naming scheme to hide the physical boundaries between machines. Since Java threads need to access data objects, a global object space (GOS) is built within the GTS, which creates the illusion that all the machines are sharing a single, global memory space.

- *Single-program-parallel-subsystem (SPPS) paradigm*: Parallel execution of an application is achieved by simply creating as many threads as needed, and these threads can be automatically mapped to different cluster nodes to exploit real execution parallelism. Application programmers no longer need to be cognizant of the physical topology of the underlying cluster, such as the number of processors available.

- *Preemptive migration of Java threads*: A Java thread executing in JESSICA can be preempted and migrated to another node anytime to achieve dynamic load balancing and optimal resource utilization. This can be done without the user's involvement in deciding which thread to be migrated and when and where the selected thread is to be migrated.

- *Migration and location transparency*: In JESSICA, any location-dependent resource is transparently accessible by a migrated thread, hence achieving migration transparency. Even after thread migration, the location of the thread is also transparent to the thread itself, as well as to other objects in the system.

- *Compatibility*: The implementation of JESSICA is at the middleware level, between the operating system and the Java applications. In addition, conformance to the standard Java Virtual Machine (JVM) Specification [8] implies that no special programming language features need to be introduced. Thus, existing Java applications are ready to be executed on JESSICA without any modification. This enhances code reusability and reduces the programming effort.

- *Portability*: JESSICA is in essence a distributed version of the JVM, which runs on top of the standard UNIX operating system as a distributed application. The implementation does not require any low-level or platform-specific supports. Hence, it is portable across different hardware platforms.

(a)

Application's
Thread Space

Application's
Object Space

Application's
Thread

JVM   JVM   JVM

H/W   H/W   H/W

INTERCONNECTION NETWORK

(b)

Application's
Thread

Global Thread Space

Global Object Space

JESSICA
MIDDLEWARE

H/W   H/W   H/W

INTERCONNECTION NETWORK
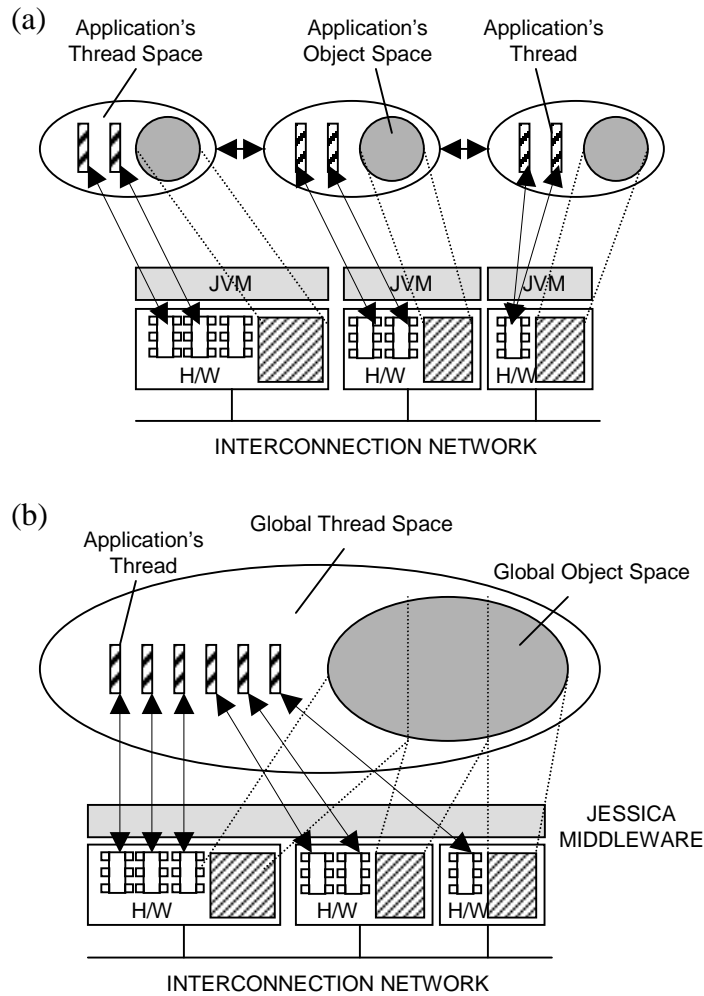
Figure 1.1. (a) A distributed Java application. (b) JESSICA transforms a cluster into a single multi-processor machine.

1.2.2   JESSICA System Architecture

Figure 1.2 shows the system architecture of JESSICA. JESSICA exists as a middleware between the operating system and the JAVA applications. It supports the *global thread space* (GTS) atop multiple cluster nodes. The GTS is an SSI layer over the cluster, and it is the programming and execution environment as seen by application programmers. The layer is supported by three subsystems:
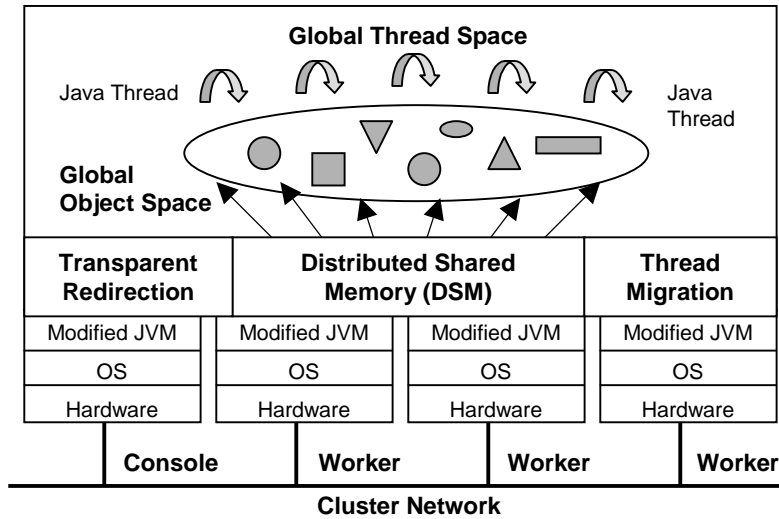
Figure 1.2. JESSICA system architecture.

- *Transparent redirection subsystem*, which handles the redirection of system requests issued by the migrated threads;

- *Distributed shared memory (DSM) subsystem*, which handles remote data object accesses and maintains data consistency so that the GOS can be established.

- *Thread migration subsystem*, which is responsible for thread migration for dynamic load balancing.

These three subsystems are implemented as system daemons, in the form of user-level processes running in different nodes of the cluster. They make use of the underlying operating system services to support the realization of the GTS.

The cluster nodes running JESSICA are referred to as either *console nodes* or *worker nodes*. A console node is where a Java application is started, one for each application. Other nodes in the cluster, which contain one or more migrated threads of the application, are worker nodes.

The console node is responsible for handling location-dependent system service requests from a migrated thread, such as getting the current time in the console node. A location-dependent request from a migrated thread will be redirected from the corresponding worker node to the console node. The console node performs the necessary operations and returns the result to the migrated thread. This is all transparent to the user. On the other hand, location-independent requests such as reading a byte stream in the GOS are served at the worker node where the thread is located.

Another function of the console node in JESSICA is to co-ordinate the migration of threads. Load balancing information is collected at the console node and migration decisions are performed there. To migrate a thread from one remote machine to another remote machine, the thread "retreats" to the console node before migrating to the destination node.

More details of the JESSICA system architecture and implementation can be found in [3]. For the sake of completeness, the transparent redirection subsystem and the thread migration subsystem are discussed in Sections 1.2.3 and 1.2.4. The design issues of and our solutions for supporting efficient global object sharing based on a new page-based DSM model will be discussed in Sections 1.3 and 1.4 respectively.

1.2.3   Preemtpive Thread Migration

In JESSICA, threads can be moved from one node to another transparently through a *preemptive thread migration* mechanism known as *delta execution* [9]. With this mechanism, the execution context of a migrating thread is separated into two sub-components: the *machine-dependent sub-contexts* and the *machine-independent sub-contexts*. The former contains state information that is part of the internal state of the JESSICA daemons handling the thread, such as the hardware program counter which points to the current machine instruction (when a daemon is executing a native method). The latter contains state information that can be expressed in terms of the execution state of the distributed virtual machine of JESSICA, such as data stored in the virtual machine's registers. The machine-dependent sub-contexts in a thread are known as *delta sets*.

When a thread migrates away from the console node, it does not take with it its entire execution context to the destination node. Instead, the thread is split into two co-operating threads. One thread, called the *master*, will run on the console node, executing all the machine-dependent sub-contexts on behalf of the original thread. The other one, called the *slave*, will be created on the worker node. The worker thread continues the execution of the original thread at the point of migration, and executes all the machine-independent sub-contexts of the thread. The slave and worker threads switch control alternatively to execute the thread, advancing incrementally by a small amount at a time; hence the name *delta execution*. With delta execution, a thread being migrated multiple times is possible in order to achieve dynamic load balancing. Based on collected workload information of the cluster, a migration manager in the console node can call back a migrated slave thread running in a worker node, and then migrate it to another worker. This approach is simpler than directly migrating the slave thread from one worker node to another, as there will not be any residue dependency resulting in the first worker node.

1.2.4   Transparent Redirection

Another important service provided in JESSICA is the *transparent redirection* mechanism, which is for achieving *location transparency* of migrated threads. As mentioned in the previous sub-section, after a thread has been migrated, the master thread at the console node is responsible for performing location-dependent operations for the slave, such as I/O, wait and notify signaling as well as mutex lock and unlock operations. In JESSICA, these operations are redirected from the slave thread to the master thread for execution.

For location-transparent I/O support, the redirection code is implemented within the java.io and the java.net class libraries for file and I/O redirection respectively. Their interface definitions are kept unchanged so that other classes relying on them do not need to be modified.

For mutex-lock and unlock operations, JESSICA relies on a decentralized approach known as *co-operative semaphores* to implement distributed semaphores. Semaphore operations in the slave thread are redirected to the console node, and the slave will block-wait for a reply from the master thread. Upon receiving the request, the master thread operates on the co-operative semaphore on behalf of the worker. The master thread will join the waiting queue if the locking operation is unsuccessful, or continue execution otherwise. Wait and notify signaling is implemented similarly.

With this transparent redirection service, the GTS can maintain a relationship between objects in the execution environment, which is the same as the case where no migration is performed. Location-transparent services and distributed thread synchronization can be achieved, and a running thread in the JESSICA execution environment has exactly the same view as one in the standard JVM.

1.3   Design Issues of the Global Object Space

The *global object space* (GOS) provides for location-independent object accesses. It creates an SSI illusion of a single and unified shared object space for all distributed threads, thus easing the implementation of the global thread space. In this section, we discuss the design of the GOS in detail.

1.3.1   The GOS Architecture

Figure 1.3 shows the layered design of the GOS which is a sub-space of the GTS. The GOS is supported by JESSICA's *distributed shared memory subsystem* for the memory sharing service which is carried out by the *distributed object manager* (DOM). DOMs running on different nodes co-operate to implement a virtual shared memory layer. The memory sharing service includes initiating communication when one node accesses an object (a memory location) which is not in the local memory. The node would initiate a request to the remote node containing the most updated copy of that object, to obtain the contents of the object. The DOMs are also responsible for maintaining memory consistency when multiple copies of the same object exist

in different nodes, so that the clean (most up-to-date) copy can always be identified. Moreover, the DOMs need to perform *garbage collection* from time to time, which gathers unused memory to reclaim space for the GOS.
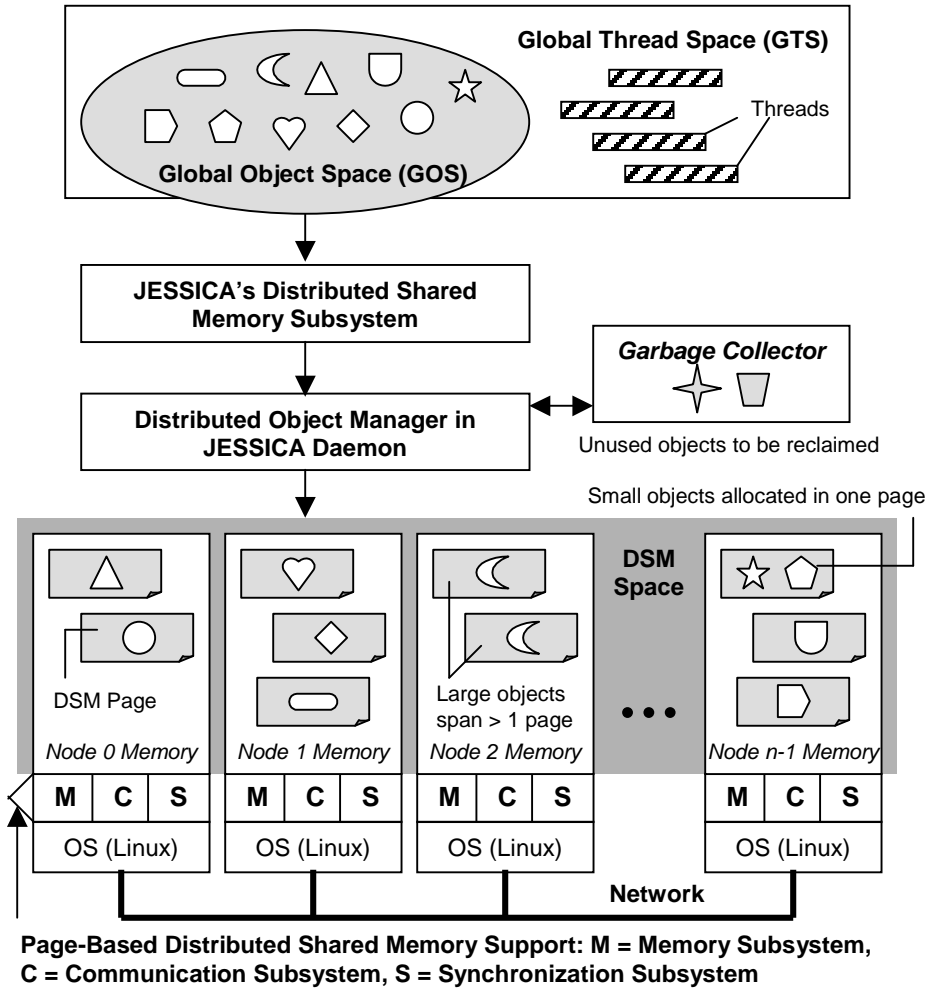

Figure 1.3. The Global Object Space architecture in JESSICA.

JESSICA opts to use a *page-based* software DSM system [10] for providing the necessary services of the DOM. A page-based DSM system implements a shared memory space through the use of the virtual memory management mechanism of the underlying operating system. It consists of three main components: (1) the

memory management subsystem, which ensures memory consistency in different nodes; (2) the *communication subsystem*, which transfers control information and page updates from one node to the other, and also fetches page copies in serving remote page requests; and (3) the *synchronization subsystem*, which provides the necessary synchronization interfaces for application programmers. Through the services provided by the underlying DSM, the DOM of a JESSICA daemon is able to supply most of the functions needed for realizing the GOS. The only exception is the garbage collection function, which most page-based DSM does not support and hence must be implemented in the DOM. Also, the DOM is responsible for transforming the page view of the underlying DSM system to the object view as seen by Java programmers.

1.3.2   GOS Initialization and Object Allocation

The GOS is initialized at the JESSICA initialization stage as shown in Figure 1.4. When a Java application is run on JESSICA, the system will call the underlying DSM support to create a large piece of shared memory. Users can adjust the size of this shared space at the command prompt when invoking the application. All the objects declared during the execution of the Java application, shared or otherwise, are allocated in this piece of shared memory. When an object needs to be created (due to a static declaration or the new() function in Java), the local DOM in JESSICA will handle the request and create the object locally in the GOS. The logical memory address to which the object is mapped will be passed to other nodes, so that they know which address to refer to when they access the object later on.
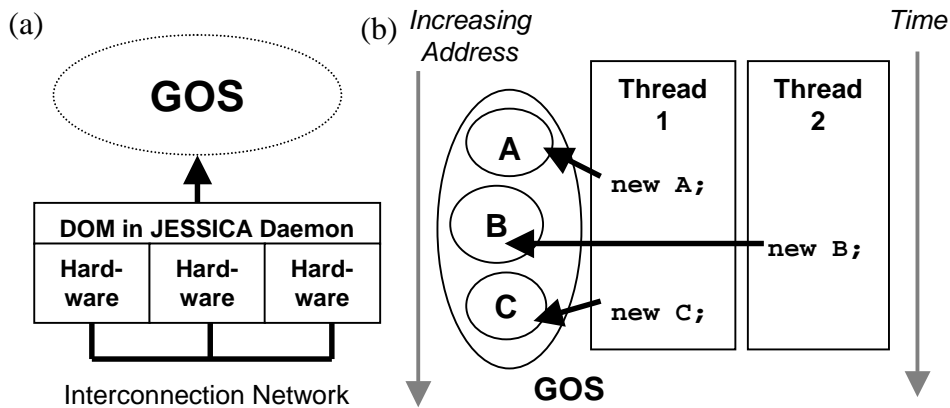
Figure 1.4. GOS creation and object allocation in JESSICA. (a) Creating a GOS in the startup phase. (b) Objects are created during execution on a first-come-first-serve basis.

Some time later, if another node tries to access the object, a segmentation fault will occur, and the local DOM will check the address of the object. If the address is valid (i.e., within the logical address space of the GOS), it will request a copy of the object from the DOM of the node holding the most updated copy of the object. If the access is a write, certain memory consistency policies must be observed (which vary with different kinds of DSM supports; to be discussed later), so that the system is able to identify the most up-to-date copy.

A question arises which is whether allocating non-shared objects in the GOS would be a waste of resources. The answer is yes in some cases. In JESSICA, however, since Java application threads are able to migrate from one node to another during execution, objects local to a thread may not be accessed by the same node throughout the thread's execution. Placing these objects out of the GOS can create problems when the thread is migrated. In fact there is no easy way for JESSICA to know which objects are shared or not given just the Java bytecode, unless additional information can be supplied via compiler analysis.

1.3.3   Garbage Collection

Garbage collection [11] is the activity to locate all the unused objects and reclaim their space. The JVM relies on garbage collection to reclaim unused memory objects. As JESSICA follows the JVM standard, it needs to implement a garbage collection mechanism for the GOS. The implementation is at the JESSICA middleware level, not at the DSM level. A *distributed mark-and-sweep garbage collection* mechanism is installed in the GOS of JESSICA, which will be invoked by the DOM when the total size of all allocated objects reaches a certain threshold. Distributed garbage collection is a non-trivial problem still under active research. JESSICA opts for a simple but fairly efficient solution. Since the GOS is divided into sections for each thread to allocate their objects created through the DOM, the DOM is the owner of any given object. During the marking phase of garbage collection, each DOM will form a list for each of the other DOMs. The list stores all traceable objects that belong to a specific DOM. After that, the lists of objects are forwarded to their respective owners. Consequently, a DOM will be able to sweep unreferenced objects and reclaim their space, as any object that is referenced remotely can be identified from lists coming from the other nodes.

1.3.4   Criteria for an Efficient GOS

The method for allocating GOS memory to objects in JESSICA described in the previous section is quite different from traditional memory allocation under DSM support alone. In applications that directly employ support from the DSM, such as TreadMarks [12] and JUMP [13], shared memory is allocated by calling an explicit function call provided by the DSM. This call can be invoked anywhere in the program. The shared variable (or object) is declared as a pointer, and is made pointing to the address of the head of the shared memory space claimed by the explicit function call. Hence, programmers take control on the use and the access

patterns of the shared memory space, even though they may not have any idea about the underlying mechanisms used by the DSM. In JESSICA, however, programmers may not have such a control. This is because a large piece of the GOS is initialized at the beginning, and it is the responsibility of the DOM in JESSICA to allocate the memory space in the GOS to the objects declared in the Java application. In particular, the following situations could happen, which programmers may not expect:
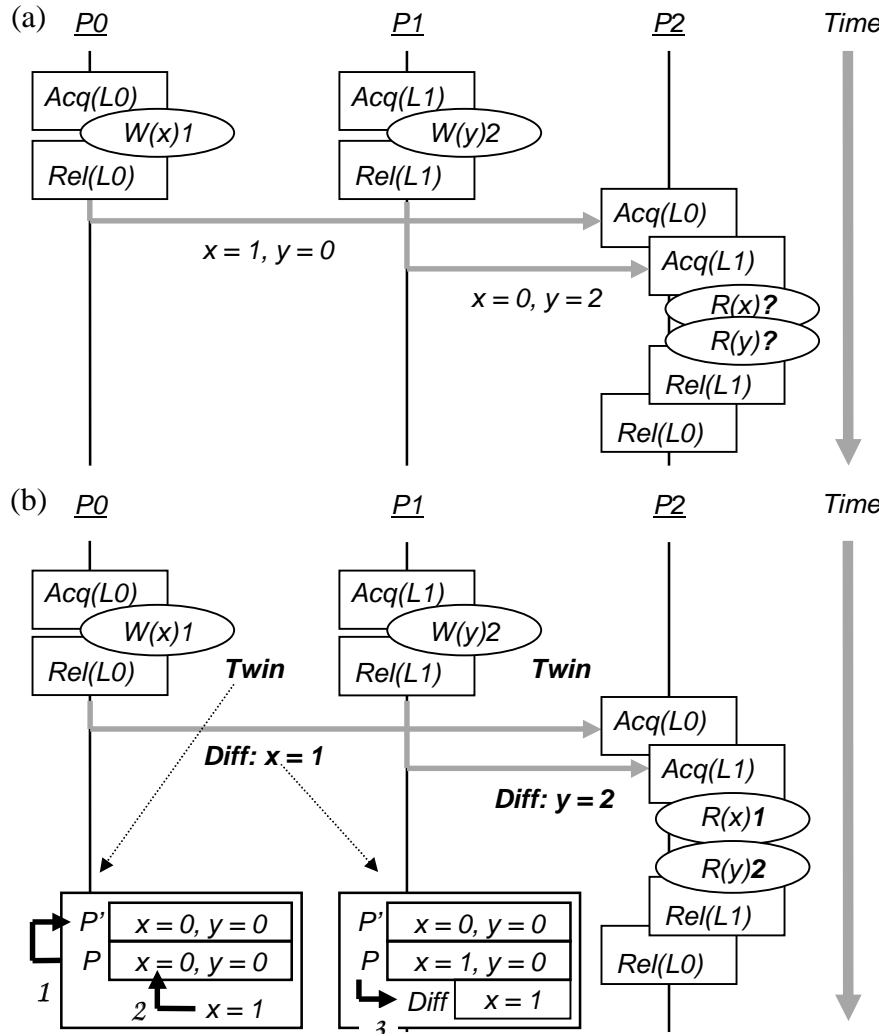
- *False sharing*: Although JESSICA views all the Java classes and variables as objects, the DSM we use to support the GOS is page-based. The GOS is therefore nothing more than a shared memory space consisting of logical pages. False sharing, which refers to the phenomenon that two or more threads accessing different locations of a shared page simultaneously (and being treated as conflicting), is therefore unavoidable.

  JESSICA targets at executing multi-threaded Java applications. Each thread can declare new objects during execution. The DOM handles the memory allocation of newly created objects in a first-come-first-serve manner. Hence, each page in the shared memory space can contain many small objects, each of them being accessed by different threads in different nodes. False sharing therefore could easily result. The underlying DSM support must be able to handle false sharing carefully (as the example in Figure 1.5 illustrates), so as to ensure correct memory behavior. In general, the higher the degree of false sharing (i.e., more processors sharing one page), the less efficient the GOS would become.

- *Unexpected memory alignment*: With multiple threads of an application executing in different nodes, the memory allocation and access pattern become difficult to predict and are non-deterministic. This is shown in Figure 1.6. If thread 1 in node 1 executes much faster than thread 2 in node 2, the objects *A* and *B* will be allocated to the same page. If however thread 2 executes the `new()` statement just before thread 1 tries to allocate memory to object *B*, object *B* will span two pages. This misalignment can hinder performance, since every time object *B* is accessed, two pages need to be brought in instead of one. Hence, the memory alignment pattern is affected by the sequence of events happening in each thread, which can be different in different executions.

To make the GOS more efficient, the following are essential considerations.

- *Reducing false sharing*: False sharing can significantly hinder the performance of DSM applications, even when the underlying DSM support employs multiple-writer protocols such as the diffing technique used by TreadMarks, JIAJIA [14], and JUMP. Reducing false sharing can therefore improve performance in general. With the unpredictable memory allocation sequence as described above, however, it is not easy for JESSICA to find a way to reduce false sharing. One of the solutions is to divide the underlying page-based DSM

(a) <u>P0</u>   <u>P1</u>   <u>P2</u>   *Time*

| Acq(L0) |
| W(x)1 |
| Rel(L0) |

| Acq(L1) |
| W(y)2 |
| Rel(L1) |

x = 1, y = 0

| Acq(L0) |
| Acq(L1) |
| R(x)? |
| R(y)? |
| Rel(L1) |
| Rel(L0) |

x = 0, y = 2

(b) <u>P0</u>   <u>P1</u>   <u>P2</u>   *Time*

| Acq(L0) |
| W(x)1 |
| Rel(L0) |

| Acq(L1) |
| W(y)2 |
| Rel(L1) |

Twin   **Twin**

| Acq(L0) |
| Acq(L1) |
| R(x)**1** |
| R(y)**2** |
| Rel(L1) |
| Rel(L0) |

Diff: x = 1

Diff: y = 2

| P' | x = 0, y = 0 |
| P | x = 0, y = 0 |
1   **2** ⌐ x = 1

| P' | x = 0, y = 0 |
| P | x = 1, y = 0 |
3 ⌐ Diff | x = 1 |

Figure 1.5. Illustrating the effect of false sharing. (a) Writing two locations x and y on a same page P by two processors causes two sets of different updated values x and y. (b) Using the diffing technique: (1) Making a twin of page P to P'; (2) writing to one copy of page P; (3) calculating the diff.

space that forms the GOS into different regions, each spanning a contiguous logical address space. Each thread is then assigned a unique region, and the objects created by a particular thread will be put into the corresponding re-

(a)

Node 1   Node 2

Time

Thread 1	Thread 2
new A;	
new B;	
	new C;

Page X   Page X+1

A		C
B		

(b)

Time

Thread 1	Thread 2
new A;	
	new C;
new B;	

Page X   Page X+1

A		B
C		
B		
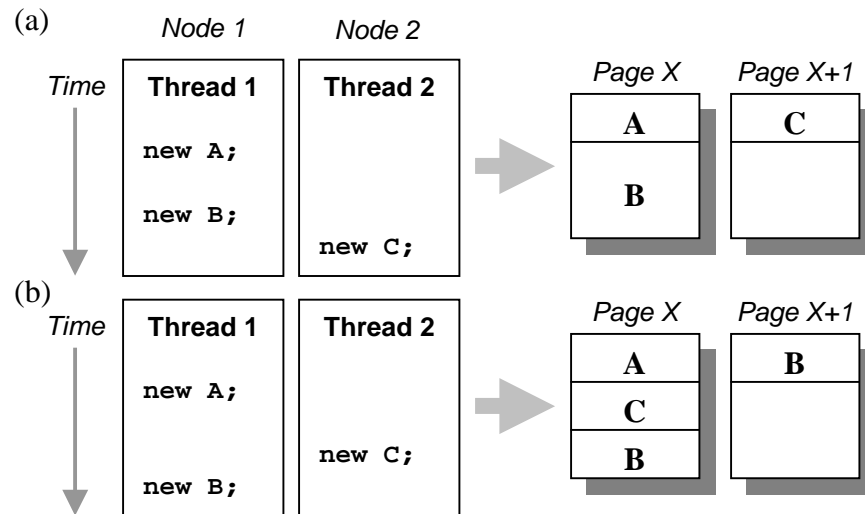
Figure 1.6. Illustrating the non-deterministic memory misalignment problem: (a) Object B fits in a page. (b) Object B has to span two pages in a different execution sequence.

gion, as illustrated in Figure 1.7. With this strategy, objects created by the same thread will be able to share a page, and with access locality, false sharing can be reduced.

A more efficient solution is to differentiate objects local to a thread from shared objects. Local objects are accessed only by one thread and therefore by one node before migration takes place. After migration, these objects will still be accessed by a single node, namely the node the migrated thread is now at. False sharing will not occur if we are able to allocate local objects accessed by the same thread to the same page. This requires runtime analysis of the Java application, which current JESSICA implementation is not equipped to do.

- *Reducing data communication*: Network communication is the major source of performance bottleneck in many distributed applications. JESSICA is no exception. Reducing false sharing is only one of many ways to reduce data communication through the network. The memory consistency model and the coherence protocol used by the underlying DSM also affect the volume of data transmitted between nodes. The mechanisms used in synchronizing different threads can also have an effect. These factors will be discussed one by one in the next section.
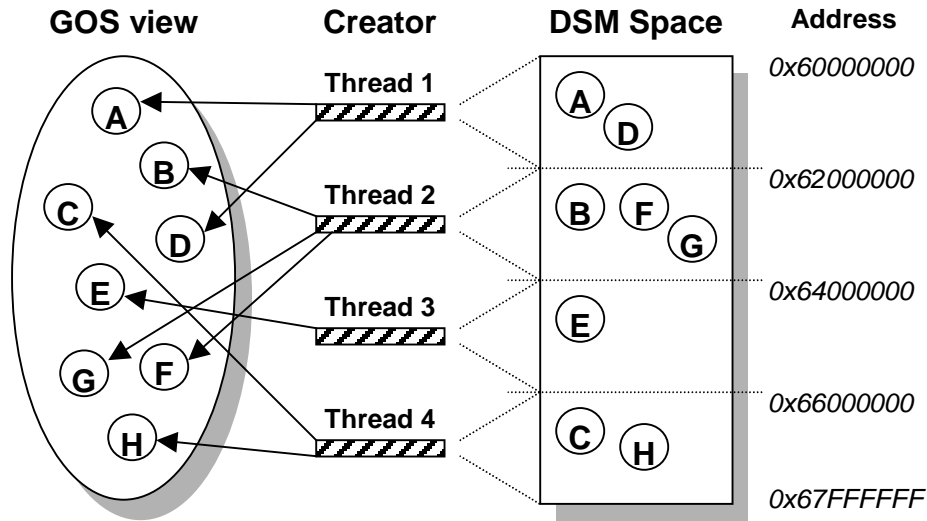
Figure 1.7. Partitioning of the underlying DSM space for different threads.

- *Efficient implementation*: The data structures used and the actual coding of JESSICA and the underlying DSM support can produce dramatic effects on the overall efficiency of the GOS, since the code could be called millions of times during the execution of an application. For example, if there are many objects of the same category but only a few of them need special treatment, using a linked list data structure to store the identities of these special objects can be more efficient, in terms of both memory utilization and execution performance, than using a bitmap to store the status of each object.

1.4   Factors Contributing to GOS Efficiency

In the previous section, we have mentioned the different factors contributing to reduced data communication in the network so that the GOS can be more efficient. We discuss these factors in detail in this section.

1.4.1   Memory Consistency Models

A *memory consistency model* (or simply *memory model*) [15] defines the behavior of memory as viewed by the applications. It is a set of rules, such that application programs abiding by these rules will guarantee to run correctly (meaning that the programs will run with the expected behavior, unless the application code itself contains errors). The most intuitive memory model is called *sequential consistency* (SC) [16], which guarantees that an update on a shared object made by any node

will be known by all the other nodes as soon as the update is completely performed (Figure 1.8). This is what is most familiar to programmers. Unfortunately, it is also the model that would lead to least efficiency, since the node performing the update must somehow send the new content of the object immediately to all other nodes which in fact may not need to access the object.
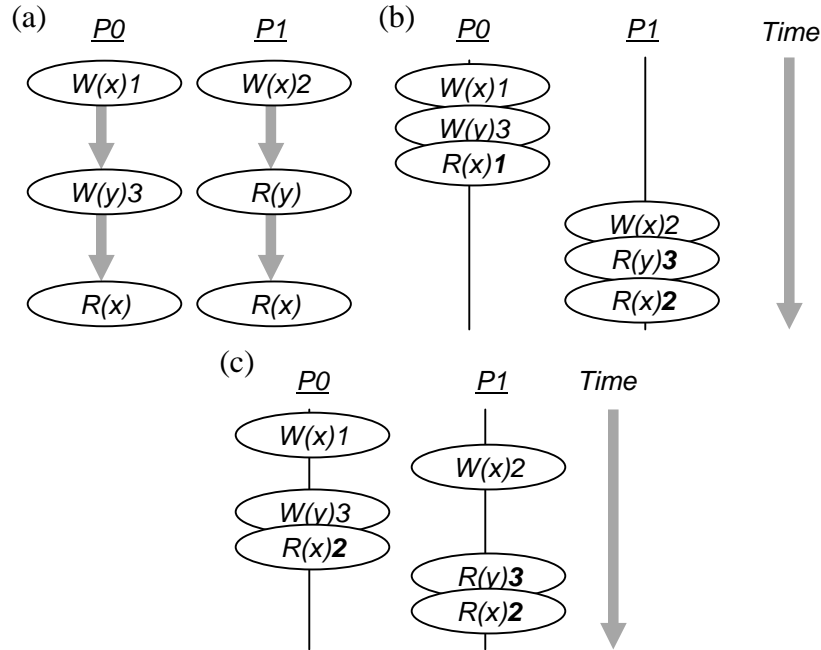
Figure 1.8. An example illustrating sequential consistency (SC): (a) Instruction execution sequence specified by SC. (b) A possible execution scenario for the sequence in (a) under SC. (c) Another possible scenario.

Therefore, certain *relaxed* memory models have been proposed. Instead of propagating the updates right away, the updates are propagated at specific intervals. Some models even introduce specific synchronization operations, so that data propagation will be performed only when statements involving those operations are executed.

Through the use of the relaxed memory models, unnecessary data communication can be eliminated, with some of the burden being shifted to programmers, as they need to add specific synchronization operations into the application program code. This way of programming goes against two of the objectives for achieving SSI in JESSICA. First, existing multi-threaded Java applications would need to be modified before they can be executed. Second, the relaxed memory model does not

match the JVM standard, as JVM observes sequential consistency.

The solution adopted by JESSICA is to use a middle layer of code (i.e., the DOM) to hide the relaxed memory model of the underlying DSM support, and to provide a GOS that follows sequential consistency. Here we shall introduce two of the relaxed memory models that can be used in the underlying DSM support, and then discuss how the DOM makes use of the underlying memory model used to conform to sequential consistency as specified by the JVM standard.

Lazy Release Consistency (LRC)

One of the most popular memory models used in DSM systems is the *lazy release consistency* (LRC) model [17], which is a variation of *release consistency*. In release consistency, the memory operations can be divided into two types: *synchronization operations* and *ordinary operations*. Synchronization operations are further divided into *acquire* and *release* operations. Though the concept is borrowed from the idea of locks, they can be implemented using semaphores or barriers. The acquire and release operations always appear in pairs, and together they guard the entrance and exit of a critical section in which only one processor at a time can access the resources guarded. Hence, a DSM system can assume that no other processor can access the objects within the critical section, and the system can delay the propagation of the updates of these objects at least until the processor in the critical section leaves the critical section.

According to the time when the processor propagates the update to the other processor(s), release consistency can be classified into *eager* or *lazy release consistency*. Eager release consistency (ERC) [18] is defined as follows:
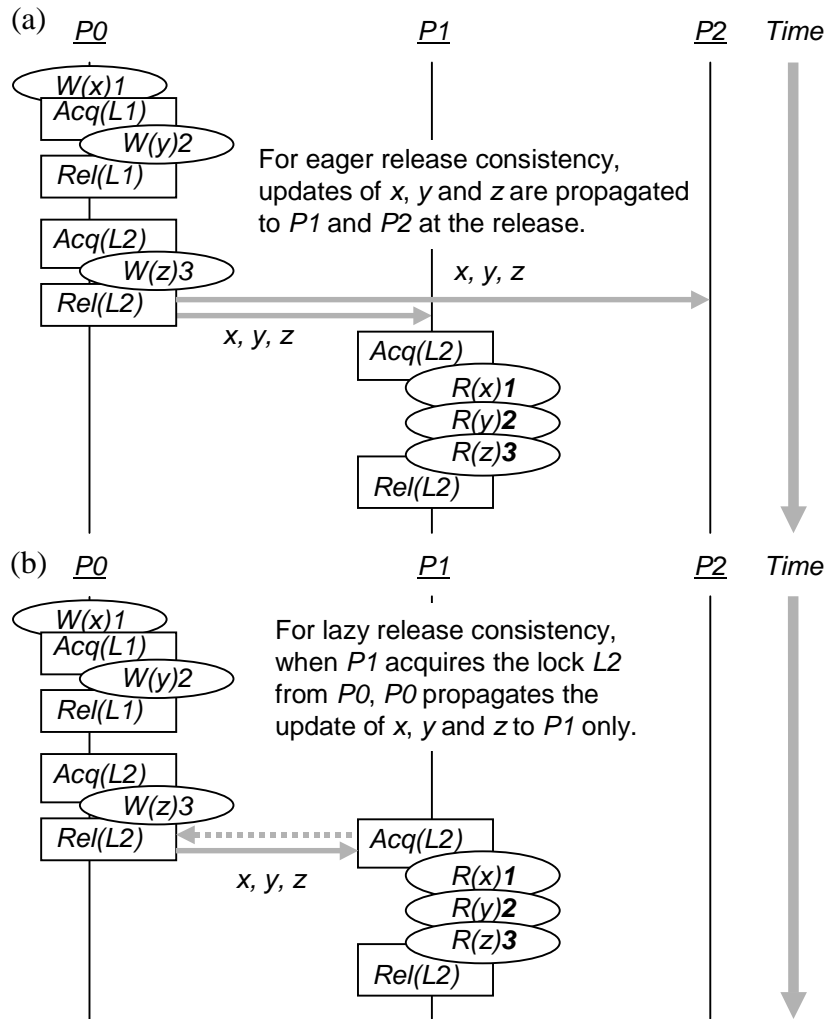
When a processor P issues a release, all the memory updates made by P before the release are made known to all the other processors.

And lazy release consistency (LRC) is defined as:

When a processor Q acquires a lock, which is most recently released by another processor P, all the memory updates made by P are made known to processor Q.

¿From the definitions, it is easy to notice that LRC can reduce some unnecessary communication, since some of the processors may never need the updates anyway. This is shown in the example in Figure 1.9. In Figure 1.9(a), ERC is used, and the updates of values *x*, *y* and *z* by *P0* are propagated to both processors *P1* and *P2* at the time when *P0* releases the lock. Under LRC, on the other hand, the updates only need to be sent to *P1* when it acquires the lock, as shown in Figure 1.9(b). The updates are never propagated to *P2*, as long as it does not acquire lock *L*.

Regardless of which form of release consistency is being used, the programs are guaranteed to behave sequentially consistently if all the shared memory accesses are guarded properly by the synchronization operations. In a word, LRC is quite

(a)

P0   P1   P2   Time

W(x)1
Acq(L1)
          W(y)2   For eager release consistency,
Rel(L1)   updates of x, y and z are propagated
                        to P1 and P2 at the release.
Acq(L2)
          W(z)3   x, y, z
Rel(L2)
              x, y, z
                            Acq(L2)
                                      R(x)**1**
                                      R(y)**2**
                                      R(z)**3**
                            Rel(L2)

(b)

P0   P1   P2   Time

W(x)1
Acq(L1)
          W(y)2   For lazy release consistency,
Rel(L1)   when P1 acquires the lock L2
                        from P0, P0 propagates the
Acq(L2)   update of x, y and z to P1 only.
          W(z)3
Rel(L2) ◀┄┄┄┄┄┄┄▶   Acq(L2)
              x, y, z   R(x)**1**
                                      R(y)**2**
                                      R(z)**3**
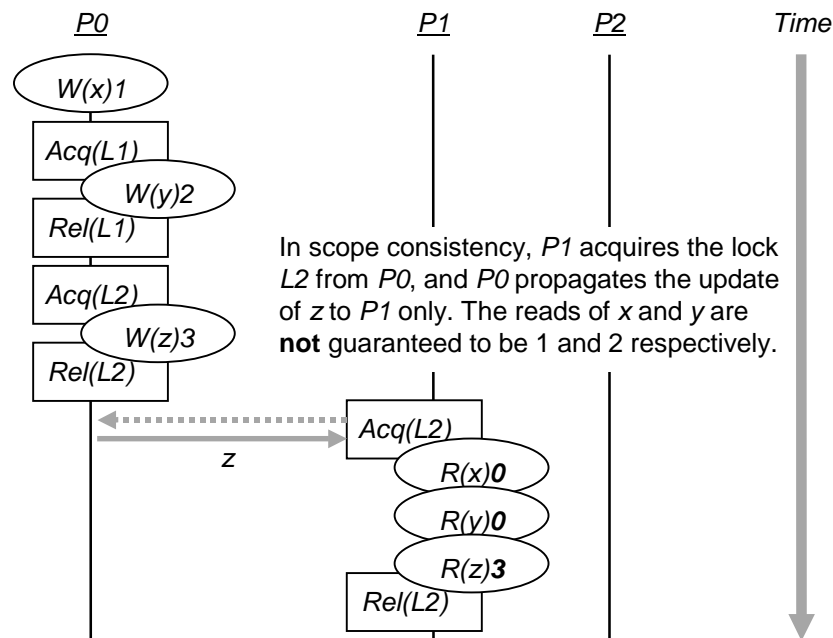   Rel(L2)

Figure 1.9. An example illustrating release consistency (RC). (a) Under eager release consistency (ERC). (b) Under lazy release consistency (LRC). Note the difference in data propagation under the two models.

efficient and provides a relatively simple programming interface, as in the case of *TreadMarks* [12].

Scope Consistency (ScC)

More recently, research has turned to the pursuit of a new memory model which can achieve high performance, while retaining good programmability like release consistency. This objective has been achieved by *scope consistency* (ScC) [19], which was proposed by researchers at Princeton in 1996. It features the brand new concept of *scopes*, which can reduce the amount of data updates among the processors, and fit naturally the synchronization provided by the lock mechanism.

A scope is a limited view of memory regarding which memory references are performed. Updates made within a scope are guaranteed to be visible only within the same scope. For example, in Figure 1.10, all critical sections guarded by the same lock comprise a scope. The locks in a program thus determine the scopes implicitly, making the scope concept easy to understand. In addition, barriers define a global scope covering the entire program. Thus at a barrier, all the updates on the shared objects made by every processor will be propagated to the others.

In scope consistency, *P1* acquires the lock *L2* from *P0*, and *P0* propagates the update of *z* to *P1* only. The reads of *x* and *y* are **not** guaranteed to be 1 and 2 respectively.

Figure 1.10. An example illustrating scope consistency (ScC).

A scope is said to be opened at an acquire operation (lock acquire or leaving a barrier), and is closed at a release (lock release or approaching a barrier). With these concepts, ScC is defined as follows:

When processor Q opens a scope, which is previously closed by another processor

P, the updates made within the same scope in P is propagated to Q.

This means that updates made outside the same scope will not be propagated at the time of the acquire. This is shown by the program example in Figure 1.10, which is the same example as that in Figure 1.9. The contents of x and y updated by *P0* will not be propagated to *P1* at the lock acquire. This allows ScC producing less data communication in the cluster than using LRC, hence improving the performance. ScC shares the same programming interface with LRC. In a nutshell, if a particular variable or object is guarded by the same lock every time it is accessed, LRC and ScC both guarantee that the memory is consistent.

A Sequentially Consistent GOS

In order to abide by sequential consistency as observed by the JVM standard, the JESSICA middleware is responsible to hide the relaxed memory models used in the underlying DSM support, so that the applications on top are able to see a sequentially consistent GOS. As described above, users can see a sequentially consistent memory space under a relaxed memory model if every object is properly guarded by suitable locks. Therefore, for every line of code which accesses the objects in the GOS, a lock acquire is performed before the object access. The lock is then released afterwards. In ScC, the lock used must be the same one for every access of the same object, starting from the moment the object is created. A mapping function from object address to lock ID is hence needed. Here a many-to-one mapping is used, since theoretically an unlimited number of objects can be created in a Java application, and therefore it is impossible to assign one lock to each object for resource consideration. The current version of JESSICA supports 1024 locks.

Of course, there is a slight performance penalty for multiple objects to share a lock. First, when two threads try to access two different objects, which are mapped to the same lock, one of them must wait until the other exits the critical section. Second, since multiple objects share the same lock, at synchronization points, the data propagation will contain the updated contents of multiple objects, of which some may not be needed and are thus wasted.

Another issue raised by this mechanism trying to hide the underlying memory models is that the number of DSM lock acquire and release operations called by the JESSICA middleware may be excessive. A simple application program can indirectly invoke these DSM calls millions of times. Hence, a *light-weight* locking and unlocking mechanism is very much called for for an efficient GOS.

1.4.2   Coherence Protocols

Apart from memory consistency models, *coherence protocols* [20] can greatly affect the performance of the DSM. They are, however, often being overlooked, since unlike memory models, which affect the programming of DSM applications, users may not directly see the difference even when the coherence protocol is changed.

It is difficult to judge whether a coherence protocol is better than the others without considering the memory model. A protocol that adapts well to the memory model will result in an efficient DSM support. In addition, the efficiency of the protocol can be application dependent. The special features of memory allocation in the GOS of JESSICA introduce specific requirements, which make one protocol better than the other. But in other stand-alone DSM applications, the situation can be completely different.

Next, we compare three different types of coherence protocols. The features of each protocol are studied and their impact on JESSICA will be discussed.
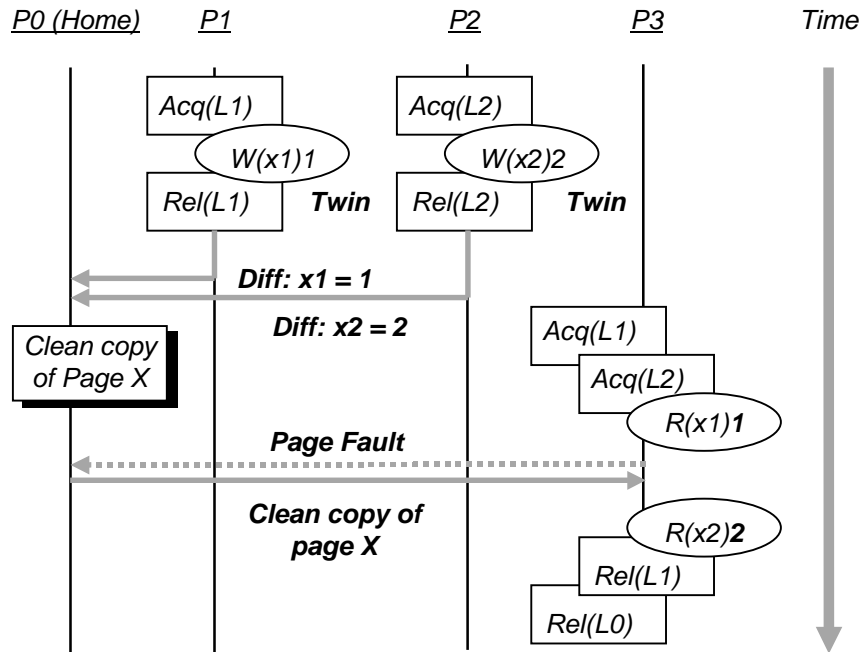
Home-Based Protocol

The *home-based protocol* [21] is used in the *JIAJIA* DSM system V1.1 [14]. In this protocol, each page in the shared memory is assigned a node known as the *home* of the page. The assignment is done in a round-robin fashion. Once assigned, the home is made known to all others and stays unchanged throughout the program execution.

The home node is responsible for storing the most up-to-date copy of the page, and responding to page requests from the other nodes. At the appropriate synchronization point (for JIAJIA, it is the lock release operation), the updates made on the specified page are propagated to the home node of the page in the form of *diffs* (i.e., the difference between the old and the new page contents). This means that the master copy of the page is guaranteed to be clean (i.e., containing the most updated contents) following synchronization. So, when a page fault later occurs, the whole page will be obtained from that particular processor only, without the participation of other processors. An illustration of this solution is shown in Figure 1.11.

For ordinary DSM applications, the home-based protocol performs satisfactorily, outperforming the homeless protocol (discussed later) in some of the applications [22]. The home-based protocol, however, does not adapt well to the memory access patterns of JESSICA due to two main reasons. First, with the home-based protocol, a remote node propagates the updates of a page to the home node at every synchronization point. Since the home node of a page is assigned arbitrarily, with no knowledge about which node is going to access the page, if the page is assigned to a node that never accesses it, the updates propagated to the home node will be redundant. In addition, for pages containing objects that are local to a thread, the home-based protocol tends to produce excessive network traffic when the accessing node is not the home node. The situation is worse when the protocol uses an eager approach to propagate the updates, as is the case for JIAJIA.

Second, using the home-based protocol, the lock acquire and release operations tend to be non-trivial. Acquiring a lock in JIAJIA involves the sending of the page invalidation information, while releasing a lock triggers the sending of the page updates to the home node. For both operations, a routine will be called in which each page in the shared memory space will be checked and all the updates will be saved (although the sending of these updates is only performed at the release).
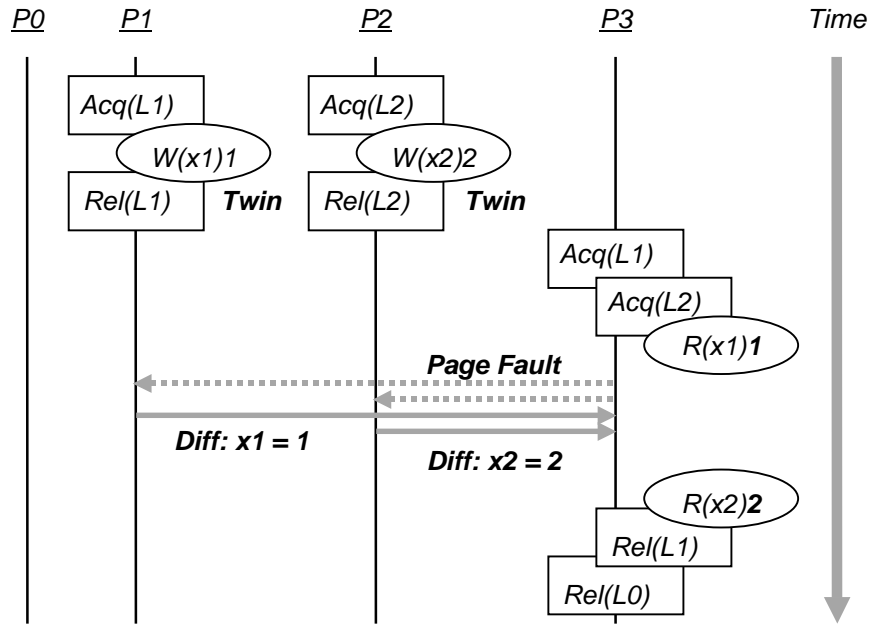
Figure 1.11. Illustrating the home-based protocol. Variables *x1* and *x2* are in the same page, *X*.

The execution time for this routine is directly proportional to the number of shared memory pages initialized, which can be very long. The situation is particularly unpleasant when JIAJIA is used to support the GOS in JESSICA. This is because a large number of shared memory pages are initialized to form the GOS at the start of the application execution. These pages will be checked for updates by JIAJIA at each lock operation, regardless of whether they have been assigned any objects. Moreover, many lock acquire and release operations have to be called in JESSICA, generating an unbearable amount of overhead.

Homeless Protocol

Instead of assigning a fixed node to store the master copy of a page in the shared memory, an alternative is to allow nodes to store their own page updates. When a node needs to access a page, it tries to contact the previous writers of the page. Each of these writers sends the updates on the part of the page it has written (in the form of diffs) to the requesting node for generating a clean copy of the page. This protocol is known as the *homeless protocol* [23], which is used in DSM systems like TreadMarks.

At first glance, homeless protocols seem inefficient due to the fact that a page request is served by communicating with multiple nodes that have written on the page. In reality, the situation is not as bad, since the page requester can contact only the latest writers of the page to derive the clean page copy (Figure 1.12). This reduces the amount of network traffic, although the algorithm used is a bit complicated as in the case of TreadMarks.

Figure 1.12. Illustrating the homeless protocol. Variables *x1* and *x2* are in the same page, *X*.

The performance of the homeless protocol can be further enhanced by the *lazy-send* feature. The updates made on a page by a node are not extracted and sent until the node receives a request for the page. This feature is particularly efficient for pages containing solely objects local to a thread. As such pages will not be requested by other nodes, data will not be propagated under the homeless protocol. This "lazy diff creation" feature also helps the lock operations to become more light-weight. At a lock release, the releaser only needs to check for any existing pending acquirers, and sends the lock together with the set of page numbers that have been updated to the first pending acquirer (if it exists). For a lock acquire operation, the acquirer only needs to invalidate the pages that have been updated by other nodes when it receives the lock grant message. There is no need to check each page in the shared memory region for the update contents, as it is delayed until the specific

page request is received. This light-weight locking mechanism adapts particularly well to the locks and unlocks performed by JESSICA.

Migrating-Home Protocol (MHP)

A new protocol known as the *migrating-home protocol* (MHP) [24] has been proposed in the *JUMP* DSM system [13]. In this protocol, each page is assigned a home node in which the master copy is stored, just like the home-based protocol. The home location in the MHP, however, can be migrated from one node to another during execution, in order to adapt to the memory access pattern of the DSM applications.

With the MHP, a processor requesting a page from its current home processor can become the new home if the contents of the page is totally clean. This is illustrated in Figure 1.13. When $P0$ tries to access the page X and triggers a page fault, the page request will be directed to the original home of the page, $P2$. In return, $P2$ will send a copy of page X to $P0$, and also grant $P0$ to become the new home node. In this way, when $P0$ writes on page X and later releases the lock $L0$, it does not need to send out updates or diffs of page X, since its copy of page X is the master copy.

To notify the other nodes about the home change, $P0$ will send a message to each of the other nodes in the cluster at the release operation to declare that $P0$ is the new home node of page X. Notice that before this message is received, the request of page X from other nodes such as $P1$ will still be directed to $P2$. $P2$ will still send a copy of X to this late requester, but home will not be granted. The page obtained by $P1$ is still clean as long as the variables updated by $P0$ are not taken into account. Therefore, this mechanism guarantees that the contents of the variables accessed by later requesters are up-to-date, since by LRC or ScC, no processor is supposed to access the memory variables updated by other processors before any synchronization takes place. (If such a case really happens, the behavior is undefined according to the definition of the memory models.)

By the definition of the MHP, at the time the migration decision is made, the home node should have collected all the updates from all the nodes having previously requested the page. For example, using Figure 1.13 above, if there is a node $P3$, which tries to request for page X from $P0$ before $P1$ releases the lock $L1$, $P0$ will not grant $P3$ to be the new home when serving the page request from $P3$.

The movable feature of the MHP reduces the redundant data propagation at the time of a release that would otherwise occur if using the home-based protocol, because one of the nodes recently accessing the page must be granted the home under the MHP. This allows the MHP to adapt to applications with thread migration better than the fixed home approach. In particular, in JESSICA, if the whole page contains objects local to one thread, the MHP, like the homeless protocol, will introduce no data communication among nodes when the thread calls a lock synchronization operation.

The MHP and the homeless protocol each has its pros and cons in adapting to the GOS in JESSICA. First, the notion of home in the MHP allows only one node
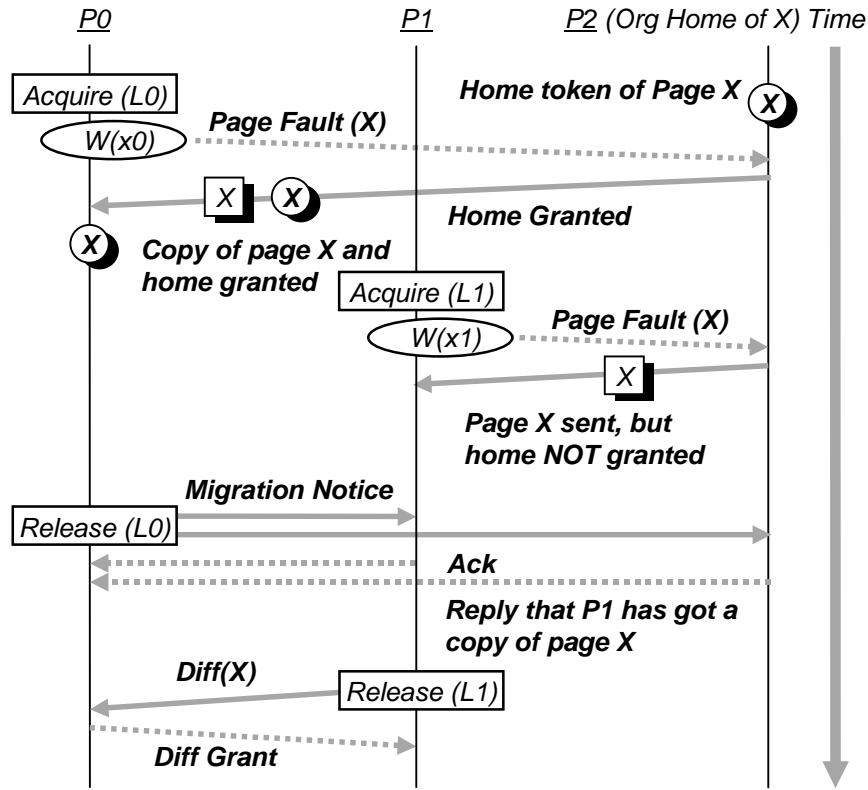
Figure 1.13. Illustrating the migrating-home protocol (MHP). Variables $x0$ and $x1$ are in the same page, X.

to be contacted in order to serve a page fault, while in the homeless protocol, the page requester may have to ask multiple nodes in order to put together the page. The tradeoff is that in the MHP, all the nodes other than the home which have updated the page have to send the updates eagerly when release a lock, whereas the updates are sent on demand in the homeless protocol.

Secondly, in the MHP, each page request is served at the home node by sending the entire page. In the homeless protocol, diffs (parts of a page) are sent from recent writers to the faulting node, unless the faulting node does not have a copy of the page, which is not likely to happen at the later stages of application execution. The diff size depends greatly on how much of a page is modified, as well as the access pattern. It is possible for diffs to be larger or smaller than the size of a page (though they tend to be smaller). The comparison is even more complex as it takes time under the homeless protocol for a node to calculate the diffs on demand, and for the page requester to apply the diffs in order to serve the page fault.

Finally, the locking and unlocking overhead in the MHP is greater than that in the homeless approach. The eager diff propagation of the MHP can take a long time to complete in the release operation. As the number of lock acquire and release operations can be very large in JESSICA, the overhead due to the MHP has serious implication on the overall performance. Thus, some implementation optimizations have to be introduced, as described next.

1.4.3   Implementation Optimizations

As the DSM lock acquire and release overhead has a direct bearing on JESSICA's performance, the implementation of these lock and unlock routines, including the data structures used, needs to be very carefully done.

Locking Mechanisms

The performance of various locking mechanisms is often overlooked as they may differ only by milliseconds or even microseconds. This small difference, however, can be amplified into a serious performance impact on JESSICA due to the great number of lock operations performed. One of the simplest locking mechanisms is based on a *centralized* approach, as shown in Figure 1.14. In such an approach, each lock is assigned a fixed manager at the start of the distributed application. The manager holds the lock, and handles all incoming requests for the lock using a queue. All the acquires of that lock by any node in the cluster will be forwarded to the manager. The manager grants the lock to the requester at the head of the queue. When a node releases a lock, the lock's control is eagerly sent back to the manager, together with control information such as the pages updated within the lock's critical section.

The centralized locking mechanism works well for most of the DSM applications. Not so, however, for JESSICA. In JESSICA, all the Java objects are allocated in the GOS, and every access of this object has to be guarded by the same lock (recall that this lock's acquire and release are performed in the DOM in JESSICA, not explicitly by the Java application). Access to objects local to a thread is no exception (since the present JESSICA has no knowledge of whether objects are local or not). Hence for a local object, the associated lock is accessed always by the same thread. In such a situation, eagerly releasing the lock back to the manager is really not necessary.

An alternative is to delay the release of the lock until the next acquiring request is in. Instead of sending the lock back to the manager, the lock is directly sent to the next acquirer of the lock. With this approach, the manager is still fixed at the start of the distributed application. The manager, however, does not serve as the middleman for collecting the lock from the remote node that issues a release. Neither is it responsible for granting the lock to other nodes (except at the beginning, when the lock has not been acquired by anyone). Instead, when the manager receives a lock request from a remote node, it will forward this request to the previous acquirer of the lock, which may or may not have the lock. When a node holding the lock issues a release, it will check if there is any pending acquirer. If there is, it will send
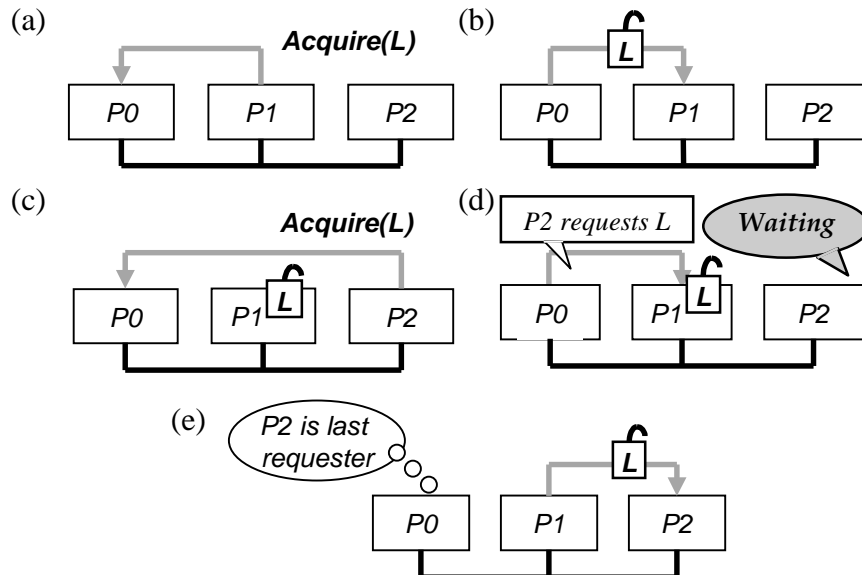
(a)

(b)

(c)

(d)

(e)

(f)
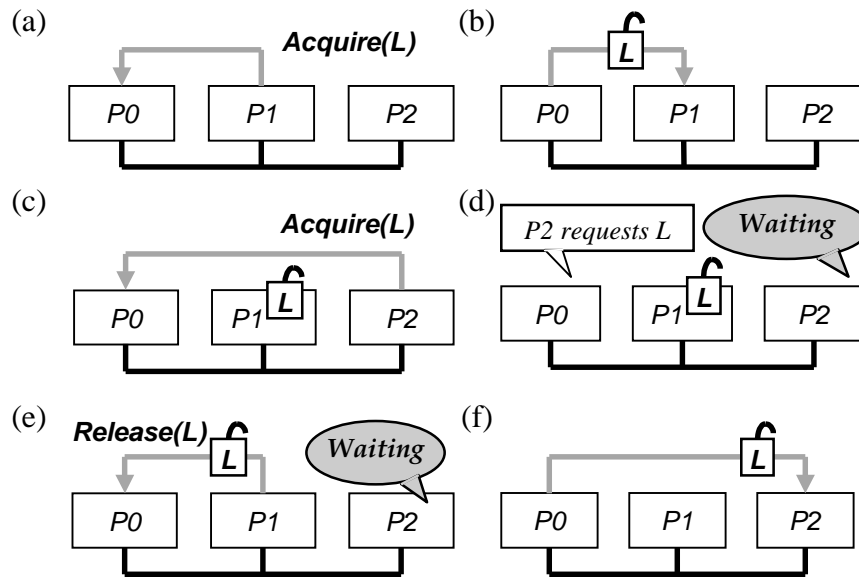
Figure 1.14. The mechanism of a lock using the centralized manager algorithm. *P0* is the manager. (a) *P1* acquires the lock *L*. (b) *P1* obtains the lock. (c) *P2* acquires the lock *L*. (d) *P2* has to wait since the lock is acquired by *P1*. (e) *P1* releases *L*. (f) Manager *P0* gives the lock to *P2*.

the lock and the control information to the pending acquirer; otherwise, it will just store the lock and wait until a request is forwarded from the lock manager. The mechanism is illustrated in Figure 1.15. One important feature of this mechanism is that, if there is no request that has come in for a lock *L* from the moment *L* is released by a node to the time it is acquired by the same node again, no messages need to be sent. In effect, the lock release and the next acquire operation cancel out each other.

The implementation of the GOS and the underlying DSM support should take advantage of the alternative locking mechanism described above, which is a kind of a lazy strategy.

Data Structures Used

As mentioned earlier, the data structures used in the implementation can affect the performance of the GOS and hence the entire system. It takes about 1 us for a 300-MHz Pentium II machine to loop 8 times within an empty `for()` statement in C. Thus, if we have a large array of elements, in which only a few of them pass a certain condition, it is better to record these elements in a separate array or linked

Figure 1.15. The mechanism of a lock using the alternative approach. *P0* is the manager. (a) *P1* acquires the lock *L*. (b) *P1* obtains the lock. (c) *P2* acquires the lock *L*. (d) *P0* forwards the lock request to *P1*, and *P2* waits for the lock. (e) *P1* releases *L* and directly sends to *P2*, while *P0* remembers that the last requester of lock *L* is *P2*.

list, rather than to use a `for()` loop to check each of the elements in the large array. Such a case is found in the MHP, as the original implementation of the lock acquire and release operations uses a `for()` loop to check for "non-home" pages which have been updated recently, so as to compute and save the updates for those pages. Because of the per-object granularity of JESSICA, however, only a few pages (one in most situations) will be updated in each critical section. Using a `for()` loop is clearly inefficient, and the execution time becomes linearly proportional to the size of the GOS. A better implementation is to use an extra linked list to hold the logical page numbers of all the pages that have their contents updated. When a node tries to write on a "non-home" page, a signal is triggered and an entry recording the page number is added to the linked list. Such an implementation has helped improve the performance of the GOS in JESSICA.

1.5   Performance Evaluation

To understand the effects of using different memory consistency models and coherence protocols have on the efficiency of the GOS, which in turn affect the perfor-

mance of the applications, a number of experiments are carried out on a cluster of four 300-MHz Pentium II PCs connected by a Fast Ethernet. We considered four machines to be enough to illustrate the essential performance features, and we expect seeing similar behavior for larger-size clusters. The JESSICA system is tested with two flavors of DSM support: the TreadMarks DSM, which employs a homeless protocol for implementing LRC, and the JUMP DSM system, which uses the MHP to implement ScC. We choose TreadMarks because it is by far the most popular DSM system to date, and it has reasonably good performance. The JUMP DSM system is our own development which is an improved version of the JIAJIA DSM, beating the original implementation in most applications tested [24]. The source code for both systems is available, and so that we can analyze and fine tune the code to achieve optimal results.

A set of seven multi-threaded Java applications are executed in each of these two environments. For each application, four threads are started at the console node. Each of them is then migrated to a different node immediately after system initialization. The performance results are summarized in Table 1.1. For comparison, the sequential execution timings, from all the threads running on a single node, are included. JESSICA appears clearly to be able to achieve a good speed-up with multiple nodes running. Note that two of the applications failed to run well on JESSICA under TreadMarks, the timings for which we omit.

Application	TreadMarks	JUMP	Sequential
Producers-Consumers	10.03	11.41	57.82
Dining Philosophers	10.46	10.79	58.83
Readers-Writers	10.83	12.11	58.17
Pi Calculation	33.55	34.92	97.93
Parallel Sieve	0.55	5.40	49.15
Radix Sort	N/A	5.85	48.05
Matrix Multiply	N/A	8.24	49.91

Table 1.1. Execution times of various applications on JESSICA with TreadMarks or JUMP DSM support (all the numbers are in seconds).

¿From the table, we find that for most of the applications, JESSICA under TreadMarks runs slightly better than under JUMP, despite the fact that Tread-Marks employs a less relaxed memory model. There are two reasons. First, the lazy diff creation and the lazy-send techniques of the homeless protocol are very effective for JESSICA applications. No redundant data are sent, whereas the home-based protocol would sometimes send data redundantly. Second, page updates are not performed at lock acquire and release operations, making the locks very light-weight.

In comparison, although the MHP also avoids the sending of unnecessary updates to the home node (as one of the nodes accessing the page must be the home), page updates are still triggered by locking operations. This turns out to be a perfor-

mance bottleneck as JESSICA tends to issue many lock and unlock calls. Table 1.2 summarizes the average cost in calling a lock acquire and release operation with each of the DSM systems, including JIAJIA. It shows that delaying the data propagation of a page request makes the locking and unlocking overhead much smaller. JIAJIA uses the centralized locking mechanism, with little or no implementation optimizations. The locking overhead is extremely large and proportional to the size of shared memory initialized. Thus it is unsuitable for supporting the GOS in JESSICA.

DSM System	Producers-Consumers		Pi Calculation	
	Acquire	Release	Acquire	Release
TreadMarks	4.02 μs	3.73 μs	3.93 μs	3.74 μs
JUMP	52.4 μs	52.8 μs	42.6 μs	45.7 μs
JIAJIA V1.1	$\sim$750 μs	$\sim$750 μs	$\sim$750 μs	$\sim$750 μs

Table 1.2. The cost of performing a lock operation with each of the DSM systems. (For each system, 32MB of shared memory is initialized.)

The large difference in the locking overhead between JUMP and TreadMarks, as well as the large number of lock acquires and releases called contrast sharply with the small difference in the actual execution time (Table 1.1). This suggests that the more relaxed ScC model and the MHP are rather efficient; they actually help to narrow the large performance gap due to locking overheads. The lesson learned is that the locking mechanism has a pivotal effect on the overall efficiency of the GOS.

Next, we look at the effect of dynamic thread migration facility of JESSICA under different DSM supports. We use the Pi Calculation application since it has the longest execution time, allowing more migrations to be performed. Four threads are initialized at the console node at the start of the program. Then we perform arbitrary thread migrations and retreats for m times, following a fixed pseudo-random pattern which can be regenerated for fair comparison. The destination worker node chosen each time is by round-robin. The timing results are shown in Table 1.3. From the data, we observe that with either DSM support, the execution time of the application increases with the number of migrations, m, which is slower than the load-balanced cases in Table 1.1. This is because thread migration and retreat take time, and also the random thread migrations may not result in a better balance in workload. In a production system, a policy module will dictate how migrations should take place in order to achieve satisfactory balancing of workload among the nodes.

From the table, we can see that with dynamic thread migration, JESSICA under JUMP performs better than under TreadMarks. This suggests that the MHP adapts better to the memory access pattern where there is dynamic thread migration than the homeless protocol. In particular, the feature that the home migration of a page would follow the migration of the thread closely has a positive effect. The

No. of Migrations (m)	TreadMarks	JUMP	Percentage Improvement
5	50.00s	49.38s	1.24%
10	51.65s	49.87s	3.45%
15	52.59s	50.66s	3.67%
20	53.62s	51.57s	3.82%

Table 1.3. Timings of the Pi Calculation application with arbitrary thread migration.

seemingly minor improvements of JUMP over TreadMarks should not be taken lightly because TreadMarks is a very mature system with optimized performance. We believe if JESSICA's locking mechanism can be simplified in the future, the improvement figures will definitely be much more substantial.

1.6   Related Work

Java/DSM [25] is one of the earliest research studies on the execution of Java applications in a distributed environment. It uses the TreadMarks DSM system for the underlying DSM support. Like JESSICA, all the Java objects are allocated in the shared memory region (i.e., a GOS). The primary goal of Java/DSM, however, is to support distributed Java computing in a *heterogeneous* cluster environment, rather than to provide dynamic thread migration and load balancing. Therefore much of the emphasis is on *data type conversion*. Threads cannot be migrated dynamically, and location transparency of threads is not achieved. Moreover, the JVM specification has been changed slightly, and programmers need to add synchronization operations explicitly in the application code, impacting code reusability.

Solaris MC [26] is a prototype of a distributed operating system for running in a cluster of computers. It extends the existing Solaris operating system to provide a single system view. It supports a global process space that spans the whole cluster, analogous to the GTS in JESSICA. Location transparency is achieved for processes in Solaris MC, but dynamic migration of processes is not supported. To achieve a global view of objects, instead of using DSM support, Solaris MC uses a *proxy file system* (PXFS) to cache remote object accesses. The PXFS is also responsible for maintaining memory consistency.

JavaParty [27] is a compiler support tool for executing Java programs in a distributed environment. Rather than using a DSM, it modifies the existing RMI support in JDK, and apart from the RMI semantics of classifying objects as local or remote, JavaParty also allows explicit declaration of remote classes. In JavaParty, objects created with the `new()` statement are accessible from the entire JavaParty environment at once, without explicitly exporting or binding them to a name in a registry as with traditional RMI. Local objects in JavaParty behave like those in distributed RMI applications, while methods of remote objects will invoke the RMI. JavaParty aims to improve the programmability of traditional RMI, but the

introduction of the remote class hinders code reusability because the JVM interface is altered.

Jalapeño [28] is a virtual machine for Java servers written from scratch in Java. Runtime services, conventionally supported by native methods, are implemented in Java code. It makes use of its own compiler to convert Java bytecode to machine instructions during runtime. Jalapeño supports a global object space in which all the Java objects are allocated. One main feature of this GOS is that it is divided into two subspaces, one for allocating large objects and one for small objects. This allows different object allocation mechanisms and garbage collection strategies to be implemented in each of the subspaces to achieve a more efficient memory subsystem.

cJVM [29] is a cluster-aware JVM implementation, which creates an SSI illusion over a cluster of computers. Instead of using a DSM, cJVM maintains a distributed heap by using the *master-proxy model* for object creation. It also uses the *method shipping* technique for transparent remote object accesses. For a Java object that is passed as reference to a remote node, a proxy of that object will be responsible for forwarding the execution flow back to the original node in which the master object resides, where operations on the object would be performed. Multiple ways to handle a remote object access request are implemented, and it is the responsibility of the object proxy to choose the most efficient handling method.

The proxy approach used in cJVM saves the use of explicit synchronization primitives for object access. The cJVM implementation, however, still needs to send out remote access requests when a bytecode instruction tries to access heap data that is located in a remote node. Moreover, although the proxy approach in cJVM eliminates false sharing, which is a problem in systems employing a page-based DSM, it does not allow multiple threads to write to different parts of a single object simultaneously. Hence, the effectiveness of cJVM depends on the pattern and frequency of remote object accesses, whereas JESSICA relies on the memory consistency model and coherence protocol employed.

Jackal [30] is a software DSM system for running Java applications. Like JESSICA, it hides the underlying memory consistency model, implicitly performing synchronization on every object access, so that application users see a shared memory space implementing sequential consistency as required by the JVM standard. Jackal relies on the compiler to analyze and optimize the Java application code to reduce unnecessary synchronizations. The shared memory space supported by Jackal is neither page-based nor object-based, but uses a granularity of 256-byte segments in the shared regions to reduce false sharing. Dynamic migration of threads is not supported.

1.7   Future Work

We have learned from our JESSICA implementation experience that the provision of an efficient GOS is vital to the performance of the entire system. To achieve an efficient GOS, many factors need to be considered. Some of these factors are closely tied to the specific characteristics of the system, such as memory allocation

policies, access patterns, as well as the frqeuency in which each routine in the system executes. We plan to consider the following in the future in our pursuit for a more efficient GOS.

- *Locking Mechanism*: It has been shown in the performance evaluation section that the locking mechanism in JUMP somewhat lags behind that used in TreadMarks. Although the modified JUMP's locking mechanism has deviated from the centralized approach (Figure 1.14) and now resembles the approach used in TreadMarks (Figure 1.15), page update information is still propagated among processors during lock operations. This makes lock operations more heavy-weight than in TreadMarks where page updates are propagated lazily upon page requests. An implementation of a light-weight locking mechanism in JUMP will require delicate changes in the memory coherence protocol. We expect a success along this line will lead to more marked performance improvements over TreadMarks.

- *Memory Coherence Protocol*: Although the MHP in JUMP has the potential to beat the homeless protocol in TreadMarks, there is room for improvement in the current implementation. For example, performing lazy page updates can definitely benefit applications with high access locality. In addition, the broadcasting of the migration notices in JUMP has the tendency to turn into a performance bottleneck, despite the fact that they are short and can be piggybacked on diffing messages. We will find methods to reduce the overhead of migration notices, such as by limiting the scope of the broadcast (i.e., multicast). Other implementation optimizations that can also be tried. For instance, supplying partially updated pages, rather than whole pages, when serving a page request can reduce data communication, particularlly for applications with sparse data updates.

- *An Object-based DSM*: In this chapter, we discuss the use of two page-based DSM systems to support the GOS. Page-based DSM systems, intuitively, are not necessarily a good match for the GOS which is object-based. Analyses have shown that most objects in Java programs are small, with sizes far smaller than that of a page. Using a page-based system to support these small objects, therefore, can suffer from serious problems such as false-sharing, which can lead to much redundant data communication. The advantage of page-based systems, on the other hand, is the benefit derivable from the pre-fetching of pages. We will carefully study the pros and cons, in order to determine whether an object-based DSM will serve the GOS better.

All these future work items now come into the scope of the project "JESSICA 2", of which the top priority is to improve the GOS design and implementation that will contribute to much higher performance for the execution of multi-threaded Java code in a cluster environment featuring SSI.

1.8   Conclusion

The use of distributed shared memory is one way to support a global object space to achieve a single system image in a cluster of PCs or workstations. Although performance is not necessarily the only most important consideration in a design for SSI, a reasonable level of performance needs to be achieved for the resulting system to be practical. We have discussed in this article the various design and implementation issues and problems related to the GOS, and solutions that have a non-trivial effect on the overall performance. We identified two key issues concerning respectively the memory consistency model and the coherence protocol to use. Because the current JESSICA version relies extensively on the use of lock acquire and release operations, the locking mechanism built into JESSICA needs to be sufficiently light-weight. This requirement is often overlooked in the design of a GOS or DSM. Our experiments show that in static thread migration mode, light-weight locking becomes a key contributor to high performance in JESSICA, while in dynamic thread migration mode, a migrating-home protocol, which adapts well to the object access pattern of the migrating threads, can lead to more efficient operation of the GOS.

1.9   Acknowledgements

1.10   Bibliography

[1] R. Buyya (ed.). High Performance Cluster Computing: Architectures and Systems, Vol. 1. Prentice Hall, 1999.

[2] K. Hwang, E. Chow, C. L. Wang, H. Jin, and Z. Xu. Designing SSI Cluster with Hierarchical Checkpointing and Single I/O Space. *IEEE Concurrency*, 1999.

[3] M. J. M. Ma, C. L. Wang, and F. C. M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, Vol, 60, No. 10, October 2000, pp. 1194-1222. (JESSICA source code is available at: *http://www.srg.csis.hku.hk/jessica-src/*)

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

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

[6] Javasoft. Java Remote Method Invocation: Distributed Computing for Java. White Paper. *http://java.sun.com/marketing/collateral/javarmi.html*

[7] Javasoft. ORBD – the Object Request Broker Daemon. *http://java.sun.com/j2se/1.4/docs/guide/idl/orbd.html*

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

[9] M. J. M. Ma, C. L. Wang, and F. C. M. Lau. Delta Execution: A Pre-emptive Java Thread Migration Mechanism. *Proc. of International Conference on Parallel and Distibuted Processing Techniques and Applications (PDPTA'99)*, pages 518-524, June 1999.

[10] P. Keleher. Distributed Shared Memory Home Pages. *http://www.cs.umd.edu/~keleher/dsm.html*

[11] D. Plainfosse and M. Shapiro. A survey of Distributed Garbage Collection Techniques. *Proc. of the 1995 International Workshop on Memory Management*, September 1995.

[12] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proc. of the Winter 1994 USENIX Conference*, pages 115-131, January 1994.

[13] B. Cheung, C. L. Wang, and K. Hwang. JUMP-DP: A Software DSM System with Low-Latency Communication Support. *2000 International Workshop on Cluster Computing - Technologies, Environments, and Applciations (CC-TEA'2000)* organized at the *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, June 26-29, 2000.

[14] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, pages 463-472, April 1999.

[15] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12): 66-76, December 1996.

[16] L. Lamport. How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transaction on Computers*, C-28(9): 690-691, September 1979.

[17] P. Keleher, A. L. Cox and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Proc. of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13-21, May 1992.

[18] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *Proc. of the 13th ACM Symposium on Operating Systems Principles (SOSP-13)*, pages 152-164, October 1991.

[19] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Proc. of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 277-287, June 1996.

[20] M. R. Eskicioglu. A Bibliography of memory Coherence Protocols. *http://www.cs.umd.edu/ keleher/bib/dsmbiblio/node3.html*

[21] W. Hu, W. Shi, and Z. Tang. A Lock-based Cache Coherence Protocol for Scope Consistency. *Journal of Computer Science and Technology*, 13(2): 97-109, March 1998.

[22] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual memory Systems. *Proc. of the 2nd Symposium of Operating Systems Design and Implementation (OSDI'96)*, pages 75-88, October 1996.

[23] P. Keleher. The Impact of Symmetry on Software Distributed Shared Memory. *Journal of Parallel and Distributed Computing (JPDC)*, 60(11): 1388-1419, 2000.

[24] B. Cheung, C. L. Wang, and K. Hwang. A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations. *Proc. of 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.

[25] 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*, June 1997.

[26] 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*, pages 191-204.

[27] JavaParty: A Distributed Companion to Java. *http://wwwipd.ira.uka.de/JavaParty*

[28] The Jalapeño Virtual Machine. *IBM System Journal*, Vol. 39, No. 1, February 2000.

[29] 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*, September 1999.

[30] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C. J. H. Jacobs, and H. E. Bal. Source-level Global Optimizations for Fine-Grain Distributed Shared Memory Systems. *Proc. of PPOPP'01*, June 2001.