REPLICATION TECHNIQUES FOR SPEEDING UP PARALLEL APPLICATIONS ON DISTRIBUTED SYSTEMS

Henri E. Bal *
M. Frans Kaashoek
Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science,
Vrije Universiteit,
Amsterdam, The Netherlands

Jack Jansen

Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Email: kaashoek@cs.vu.nl

ABSTRACT

Traditional programming methods for loosely-coupled systems are based on message-passing. More recently, methods have emerged based for ‘‘virtually’’ sharing data. These methods simplify distributed programming, but are hard to implement efficiently, as loosely-coupled systems do not contain physical shared memory. We introduce a new model, *the shared data-object model*, that eases the implementation of parallel applications on loosely-coupled systems, but can still be implemented efficiently.

In the shared data-object model, shared data are encapsulated in data objects, which are variables of user-defined abstract data types. To speed up access to shared data, data objects are replicated. This paper discusses the design choices involved in replicating objects and their effect on performance. We have implemented several options to determine which strategy is most efficient.

Index terms: Amoeba, distributed systems, multicast, Orca, parallel programming, replication, shared data-object model.

1. INTRODUCTION

Distributed systems are becoming increasingly popular for running large-grain parallel applications. These systems are easy to build and extend, and offer a good price/performance ratio. The issue of how to program parallel applications that use many loosely-coupled machines is still open. Traditional programming methods are based on some form of message-passing. More recently, methods have emerged based on sharing data. Since distributed systems lack shared memory, this sharing of data is logical, not physical.

For many applications, support for shared data makes programming easier, since it allows

* This research was supported in part by the Netherlands organization for scientific research (N.W.O.) under grant 125-30-10.

processes on different machines to share state information. The main problem, however, is how to implement it efficiently on memory-disjunct architectures. In this paper we introduce a new model providing shared data and we discuss efficient implementation techniques for this model, based on *data replication*.

Several systems exist that use replication for implementing shared data. Probably the best known example is Kai Li's Shared Virtual Memory (SVM) [Li 1988]. This system gives the user the illusion of a shared memory. It stores multiple read-only copies of the same page on different processors. Each processor having a copy can read the page as if it were in normal local memory. Other systems providing replicated shared data are described in [Carriero and Gelernter 1986; Bisiani and Forin 1987; Cheriton 1985; Stenstrm et al. 1988; Fleisch and Popek 1989]

The model studied in this paper is called the *shared data-object model*. It is intended for implementing parallel applications on distributed systems. The unit of replication in our model is not dictated by the system (as in the SVM), but is determined by the programmer. Shared data are encapsulated in *data-objects**, which are variables of user-defined abstract data types. An abstract data type has two parts:

- A specification of the operations that can be applied to objects of this type.
- The implementation, consisting of declarations for the local variables of the object and code implementing the operations.

Instances (objects) of an abstract data type can be created dynamically, each encapsulating the variables defined in the implementation part. These objects can be shared among multiple processes, typically running on different machines. Each process can apply operations to the object, which are listed in the specification part of the abstract type. In this way, the object becomes a communication channel between the processes that share it.

The shared data-object model uses two important principles related to operations on objects:

1. All operations on a given object are executed *atomically* (i.e., *indivisibly*). To be more precise, the model guarantees *serializability* [Eswaran et al. 1976] of operation invocations: if two operations are executed simultaneously, then the result is as if one of them is executed before the other; the order of invocation, however, is nondeterministic.
2. All operations apply to *single* objects, so an operation invocation can modify at most one object. Making *sequences* of operations on different objects indivisible is the responsibility of the programmer.

These two principles make the model easy to understand and efficient. Moreover, in our experience, they provide sufficient support for many parallel applications. Distributed applications like banking and airline reservation systems can profit from more support (e.g., atomic multi-object operations), but such applications are not our major concern here. Also, parallel applications on *closely-coupled* (shared-memory) systems can use a finer grain of parallelism (e.g., parallelism within objects), but again these are not the type of applications we are interested in here. These issues are addressed by other models, such as atomic transactions and concurrent object-oriented programming and are not the topic of this paper.

We have designed a new programming language called *Orca*, based on this model. Orca is intended for implementing distributed *user* applications. In particular, the language is intended for parallel, high-performance applications. Orca is a simple, procedural, type-secure language. It

---

* We will sometimes use the term object as a shorthand notation for data-objects. Note, however, that unlike in most parallel object-based systems, objects in our model are purely passive.

supports abstract data types, processes, a variety of data structures, modules, and generics. The language, its implementation, and use are described elsewhere [Bal and Tanenbaum 1988; Bal et al. 1990, 1989a, 1989b; Bal].

In the rest of this paper we will study replication techniques for the shared data-object model. In Section 2, we will describe the space of possible design choices. The most important issues are (1) updating versus invalidation of copies, (2) the protocols used for updating or invalidating copies, and (3) the degree of replication. As we will see, the best choice depends on the communication primitives supported by the underlying distributed system. We will study two important cases. In Section 3, we will look at an implementation of the model using point-to-point message passing. In Section 4, we will discuss a second implementation, based on reliable multicast messages. The two implementations cover a broad spectrum of design choices. In Section 5, we will measure the performance of the two implementations. In Section 6, we will present our conclusions.

2. DESIGN SPACE

The technique of data replication in distributed systems is typically used to increase the availability and reliability of the data in the presence of processor failures and network partitions [Gifford 1979; Bernstein and Goodman 1981; Joseph and Birman 1987; Van Renesse and Tanenbaum 1988; Davidson et al. 1985]. For example, if multiple copies of the same logical data are stored on different processors, the data can still be accessed if some of the processors are down.

In contrast, we use replication primarily for speeding up access to shared data and for decreasing the communication overhead involved in sharing data. The general idea is to replicate an object on those processors that frequently access it. A copy may be accessed by all processes running on the same processor, without sending any messages, as shown in Figure 1.

M: box wid 4*boxwid ht 4*boxht B1: box wid 1.25*boxwid "process-1" with .nw at M.nw+(0.3*boxwid, -0.3*boxwid) B2: box wid 1.25*boxwid "process-n" with .sw at M.sw+(0.3*boxwid, 0.3*boxwid) X: box ht 1.5*boxht "copy" "of" "X" with .e at M.e - (0.3*boxwid, 0) arrow from B1.e to X.w+(0, 0.2*boxwid) arrow from B2.e to X.w-(0, 0.2*boxwid) "." at 1/4 <B1.s, B2.n> "." at 2/4 <B1.s, B2.n> "." at 3/4 <B1.s, B2.n> "CPU 1" at M.n + (0, 0.3*boxwid) N: box wid 4*boxwid ht 4*boxht with .w at M.e+(boxwid, 0) C1: box wid 1.25*boxwid "process-1" with .nw at N.nw+(0.3*boxwid, -0.3*boxwid) C2: box wid 1.25*boxwid "process-n" with .sw at N.sw+(0.3*boxwid, 0.3*boxwid) Y: box ht 1.5*boxht "copy" "of" "X" with .e at N.e - (0.3*boxwid, 0) arrow from C1.e to Y.w+(0, 0.2*boxwid) arrow from C2.e to Y.w-(0, 0.2*boxwid) "." at 1/4 <C1.s, C2.n> "." at 2/4 <C1.s, C2.n> "." at 3/4 <C1.s, C2.n> "CPU 2" at N.n + (0, 0.3*boxwid) line "n e t w o r k" below from M.sw-(0,boxht) to N.se-(0,boxht) line from M.s to M.s-(0,boxht) line from N.s to N.s-(0,boxht)

Fig. 1. Replication of data-objects in a distributed system. Each processor contains multiple processes running in pseudo-parallel. These processes belong to a single job and run in a single address space, so they can share copies of objects.

It is useful to distinguish between *read* operations and *write* operations on replicated data: a read operation does not modify the data, while a write operation (potentially) does [Joseph and Birman 1987]. For our model, we define a read operation as an operation that does not change the internal data of the object it is applied to.

The primary goal of replicating shared data-objects is to apply read operations to a local copy of the object, if available, without doing any interprocess communication. On a write operation, all copies of the object except the one just modified must be invalidated or updated. To deal with this problem, communication will be needed, so write operations involve communication. This is a departure from most of the replication techniques cited above, which in general need interprocess communication for every read and write operation.

The second goal of replication is to increase parallelism. If an object is stored on only one processor, each operation must be executed by that processor. This processor may easily become a sequential bottleneck. With replicated objects, on the other hand, all processors can simultaneously read their own copies. Since a read operation does not change its object, it can be executed concurrently with other read operations without violating the serializability principle.

The effectiveness of replication depends on many factors. One important factor is the ratio of read and write operations on objects, which is determined by the user application. Another factor is the overhead in execution time for reading or writing objects. These costs are determined by the implementation of the model. They depend on:

- The action undertaken after each write. If each write operation *invalidates* all copies, a subsequent read operation will need to do communication. If, on the other hand, all copies are *updated*, this disadvantage disappears, but write operations will become more expensive
- The protocol used for invalidating or updating copies. Many protocols exist (e.g., owner protocols, two-phase update protocols), each with their own advantages and disadvantages.
- The replication strategy. If an object is replicated everywhere, each read operation can be applied to a local copy, which is much cheaper than doing the operation remotely. On the other hand, writing an object that has many copies will be more expensive than writing a non-replicated object.

In the following subsections we will study these design choices in more detail.

2.1. Invalidation versus Updating of Copies

If a write operation is applied to a replicated object, its copies will no longer be up-to-date. There are two different approaches for dealing with this problem. The first scheme is to *invalidate* all-but-one copies of the object. The second scheme is to *update* all copies in a consistent way.

With invalidation (or *write-once*), each object is initially stored on only one processor, say *P*. If another processor wants to do a read operation on the object, it fetches a copy of the object from *P*. In this way, the object automatically gets replicated. On a write operation, all the copies are thrown away, except the one being modified.

The alternative scheme is to update (or *write-through*) all copies of an object after each write operation. A problem here is how to update all copies *in a consistent way*. The shared data-object model guarantees that all operations on objects are executed indivisibly. Hence, updating of all copies should appear as one indivisible action. On systems supporting only point-to-point communication, this is hard to do. In essence, a *2-phase* protocol is needed, as we will see. If reliable indivisible multicast messages are available, updates become much simpler, as we will discuss in Section 4.

There are several important differences between invalidation and update schemes. For one thing, keeping copies up-to-date is more complicated than invalidating copies, so the update scheme may require more messages to implement a write operation. Also, update messages will be larger than invalidation messages. An update message will either contain:

1. The new value of the object, or
2. The write operation and its actual parameters.

In the latter case, each processor applies the operation to its local copy of the object. The first approach is most effective for small objects, which do not contain large amounts of data. If an object contains a large data structure of which the operation modifies only a small part, applying the operation to all copies will be more efficient. Whichever scheme is chosen, the update message will always be larger than an invalidation message, which merely needs to specify the object to be invalidated.

On the other hand, the update scheme also has several advantages. If an object is read after it has been written, the invalidation scheme will have to fetch the current value of the object from a remote processor. With the update scheme, this value will still be stored locally, so no messages need be sent at all.

In conclusion, which of the two schemes is most efficient depends on:

1. The costs of the update protocol.
2. The size of the object.
3. The size of the parameters of the write operation.
4. Whether the write operation is followed by a read operation or by another write operation.

Kai Li argues that, for the Shared Virtual Memory system, an update scheme is inappropriate. In addition to being almost impossible to implement, it will cause a page fault on *every* write instruction [Li and Hudak 1989]. In our model, however, this disadvantage is far less severe. Users can define write operations of any complexity on shared objects. As replicas are updated after each operation—rather than each machine instruction—updating will be less expensive than in the SVM. In addition, the SVM would require a whole page to be transmitted after every write. With our approach, shared objects frequently are much smaller than a page; furthermore, large objects can usually be updated efficiently by transmitting the operation and its parameter, instead of the new value of the object.

2.2. Invalidation and Update Protocols

The protocol used for invalidating or updating copies of objects must make sure that simultaneous operations on the same object are executed indivisibly. The simplest way to implement this is to serialize all write operations (i.e., to execute them one at a time, in a mutually exclusive way). This is the approach taken by all our implementations.

In an invalidation scheme, mutual exclusion is achieved by selecting one copy of each object as the *primary copy*. In the simplest scheme, all write operations are directed to the processor containing the primary copy. On receiving a write operation, the processor first invalidates all secondary copies and then applies the operation to the primary copy. When a processor executes a read operations, it locates the primary copy and asks for the value of the object. A more sophisticated scheme allows the primary copy to move from one processor to another. Kai Li compares several of these schemes and analyzes their performance.

In an update scheme, mutual exclusion can be achieved in two ways. One way is to appoint one copy of each object as *primary copy* and direct all write operations to the processor containing the primary copy. This node will execute the write operations one by one and propagate their effects to all other copies, called *secondary copies*. An alternative approach is to treat all copies as equals and use a *distributed protocol* that takes care of mutual exclusion.

The first approach is conceptually the simplest. Moreover, it allows one important optimization: the primary copy can be migrated to the processor that changes the object most frequently, making updates more efficient. In particular, if only one processor changes the object, the overhead of mutual exclusion can be eliminated by storing the primary copy on that processor. In general, however, the primary copy method requires one extra message, from the process invoking the write operation to the primary-copy site.

With the second approach—a distributed protocol—there is no distinction between primary and secondary copies. Instead, each processor can initiate a write operation on an object. Clearly, some cooperation among the processors is needed to prevent simultaneous write operations on the same object from interfering with each other.

2.3. Replication Strategies

Replicating a shared data object is only useful if it is read relatively often. Thus, simply replicating all objects on all processors is unlikely to be efficient. In general, we can distinguish between several *strategies* for replication:

No replication:	Each object is stored on one specific processor.
Full replication:	Each object is replicated on all processors.
Partial replication:	Each object is replicated on some of the processors, based on (a) compile-time information, (b) run-time information, or (c) a combination of both.

The first approach is used in most parallel object-based languages. In this case, all operations on a given object are executed by the same processor. For many applications, this may easily lead to sequential bottlenecks and high communication overhead.

The second approach indiscriminately replicates all shared objects on all processors. It will be most effective for architectures supporting fast reliable multicast messages, since these will allow efficient updating or invalidation of all copies.

The third strategy selectively replicates objects, based on information gathered by either the compiler, the run time system (RTS), or both. With this approach, several scenarios are possible. For example, the compiler may disable replication of objects that do not have any read operations at all. Also, if a processor does not contain any processes that share a given object, it is unnecessary to store a copy of the object on that processor.

The most advanced scheme based on partial replication is to let the RTS decide dynamically where to replicate each object. For example, the RTS may keep track of read and write operations on an object issued by each processor, to determine which processors frequently read the object. If the read/write ratio exceeds a certain threshold, a replica of the object is created dynamically on that processor. This strategy is most suitable if communication is slow, so the overhead of maintaining statistics is worthwhile.

Note that an invalidation scheme automatically leads to partial replication, since the copies are thrown away after each write operation. Still, there are several strategies for deciding when to re-install the copy. The RTS may, for example, obtain a new copy the next time the object is read. If the object is large, it may be better to replicate it only after a certain number of successive read operations.

2.4. Discussion

We have discussed several design choices related to replication of objects. In general, it is hard to determine which ones will give the best overall performance. Furthermore, different types of distributed systems may require different design decisions. In particular, the communication primitives provided by the system are very important.

In the next two sections we will examine two existing implementations of the shared data-object model. Each implementation basically is a run time system for Orca. Both RTSs use the same hardware: a collection of 10 MC68020 CPUs connected by a 10 Mbit/sec Ethernet®, but use different communication primitives and consistency protocols.

3. AN IMPLEMENTATION USING POINT-TO-POINT COMMUNICATION

The first run time system we have implemented for Orca runs on top of Amoeba [Mullender and Tanenbaum 1986]. It uses point-to-point messages (RPC) for interprocess communication. Below, we will look at each of the three design issues discussed in Section 2 and motivate our choices. In Section 5.1 we will describe the performance of this system.

3.1. Invalidation versus Updating

The first issue is the choice between an invalidation or an update scheme. With point-to-point messages it is expensive to update all copies of an object in a consistent way. Simultaneous write operations on the same object can be serialized using a primary copy protocol, as described in Section 2.2. A harder problem is how to achieve serializability if a sequence of operations on *different* objects is executed.

Suppose a program uses two objects, X and Y, that have their primary copies on different processors. If X and Y are written simultaneously, either all processors should observe the change to X first or all processors should observe the change to Y first. Under no circumstances should these two events be mixed, since that would violate serializability. Because of this restriction, it does *not* suffice to implement a write operation by sending it to the primary-copy site and having this site forward it to the secondary-copy sites [Bal and Tanenbaum 1988].

The problem can be solved using a more complicated and expensive update protocol [Bal and Tanenbaum 1988]. Therefore, updating copies will be expensive, so it is not clear whether updating will be more efficient than invalidation. We have decided to implement both options and to determine experimentally which of the two is best.

3.2. The Protocols

In this section we will discuss the protocols for invalidating or updating copies, using point-to-point messages.

The Invalidation Protocol

The invalidation protocol is quite simple. A process that wants to do a write operation on a shared object sends the operation and the parameters to the processor containing the primary copy of the object. This processor locks the object and sends point-to-point *invalidate* messages to all processors containing a secondary copy. If a secondary-copy site receives this message, it throws away its local copy of the object and sends back an acknowledgement. As soon as the primary-copy site has received all acknowledgements, it updates and unlocks the primary copy. This protocol requires two messages for each secondary copy; if the primary copy is on a different processor, two messages are needed for updating the primary copy.

If a process *P* wants to do a read operation on an object of which is does no longer have a local copy, it sends a *fetch-object* message to the primary-copy site. If this processor has not yet received all acknowledgements, the primary copy will be locked and *P* will temporarily be blocked. When the object is unlocked, the primary-copy site sends a new copy to *P*.

The Update Protocol

Updating all copies of an object in a consistent way is more difficult than invalidating them. The real problem is to guarantee serializability, as discussed above. We solve this problem using a *2-phase* primary copy update protocol. The protocol updates copies by sending the operation and its parameters to the secondary-copy sites. For most programs, this is more efficient than transmitting the new value of the object.

During the first phase, the primary copy of the object is locked and a *lock-and-update* message is sent to all secondary-copy sites. This message specifies an object, an operation to be applied to the object, and the parameters of the operation. When a site receives the *lock-and-update*, it locks the local copy of the object and applies the operation to it. Next, it sends an acknowledgement to the primary-copy site, while still keeping its local copy locked. In the mean time, the primary-copy site waits for all acknowledgements and then sends an *unlock* message to all sites. The *unlock* message causes all copies of the object to be unlocked.

The 2-phase update protocol guarantees that no process uses the new value of an object while other processes are still using the old value. The new value is not used until the second phase. When the second phase begins, all copies contain the new value. Simultaneous write-operations on the same object are serialized by locking the primary copy. The next write-operation may start before all secondary copies are unlocked. New requests to *lock-and-update* a secondary copy are not serviced until the *unlock* message generated by the previous write has been handled.

Assuming no communication failures, this protocol requires two RPCs for each secondary copy. In the first phase, one request to *lock-and-update* the object is sent plus an acknowledgement for this request. In the second phase, an *unlock* message and acknowledgement are sent. Both in phase one and phase two a request and an acknowledgement have to be sent. To update an object whose primary copy is located on a remote processor, one extra RPC is needed.

3.3. Replication Strategy

With the above protocols, the costs of invalidating or updating N copies of an object will grow linearly with N. As a result, it will be expensive to replicate all objects on all processors. Our implementation therefore uses a partial replication strategy, based on run time statistics. Although this incurs some overhead on operations, communication costs can be reduced significantly. As communication in distributed systems still is expensive (on the order of milliseconds), this approach is attractive.

Initially the system contains one copy for each object: the primary copy. If some processor frequently tries to read the primary, a secondary copy will be created, so that future read operations can be applied to the local copy without sending any messages. Write operations are always directed to the primary copy.

In the invalidation scheme the owner of the primary copy invalidates all secondary copies before performing the write operation. A subsequent read operation on the same object always has to go to the processor containing the primary copy. So, the number of secondary copies of a given object is determined by its read/write pattern.

In the update scheme, the processor containing the primary copy of an object keeps track of the number of remote read and write operations issued by each processor. The overhead of maintaining these statistics is negligible compared to the total costs of remote operations. As soon as the read/write ratio of a remote processor exceeds a certain threshold, the RTS creates a copy of the object on that processor.

Each processor having a secondary copy keeps track of the ratio of local read operations and (global) write operations. If the overhead in updating the copy exceeds the time saved in doing read operations locally, the RTS discards the local copy. From then on, all operations on the object will be done remotely.

With the primary-copy update protocol, all write operations are forwarded to the processor containing the primary copy of the object. If the RTS discovers that an object is written frequently by a machine different from the one containing the primary copy, the RTS may decide to *migrate* the primary copy to that machine. Again, statistics are used to determine to best location for an object. If an object is migrated, precautions are taken for dealing with machines that are unaware of the object's new location.

4. AN IMPLEMENTATION USING MULTICAST COMMUNICATION

The second RTS runs on top of the bare hardware and uses the multicast capability of the Ethernet. The RTS uses the indivisible reliable multicast protocol described in [Kaashoek et al. 1989b]. This protocol is highly efficient and usually only requires two packets (one point-to-point and one multicast) per reliable multicast. Sending a short message to 10 processors, for example, takes 1.5 msec on the hardware described above.

In a distributed system supporting only point-to-point messages, serializability is difficult to achieve, because messages sent to different destinations may arrive with arbitrary delays. Some distributed systems (e.g., Ethernet-based systems) provide hardware support for sending a single message to multiple destinations simultaneously. More precisely, we are interested in systems supporting *indivisible reliable multicasts*, which have the following properties:

- A message is sent reliably from one source to a set of destinations.
- If two processors simultaneously multicast two messages (say m_1 and m_2), then either all destinations first receive m_1, or they all receive m_2 first, but not a mixture with some receiving m_1 first and others receiving m_2 first.

With this multicast facility, it becomes much easier to implement a protocol for consistent updating of all copies of an object. Basically, if a process wants to invoke a write operation on a shared object, it multicasts the operation to all processors. Since all processors receive all messages in the same order, all operations on shared objects are executed in the same order everywhere.

We have implemented an indivisible reliable multicast protocol in software on top of Ethernet [Kaashoek et al. 1989b]. Programs need not to worry about lost messages. Recovery of communication failures is handled automatically and transparently by the protocol. Efficiency is obtained by optimizing the protocol for no communication failures, as these rarely happen with current state of microprocessor and network technology.

4.1. Invalidation versus Updating

Reliable multicasting is useful for invalidation as well as updating. In both cases, a single reliable multicast message is needed for a write operation. If an object is written very frequently and hardly ever read, the invalidation scheme will be more efficient, since fewer messages are needed and invalidation messages are shorter than update message.

In general, however, the update scheme will be more efficient. Suppose, for example, that every processor reads a given object exactly once after it has been written. With P processors, the invalidation scheme requires a single (short) reliable multicast message for invalidating the copies and $2P$ point-to-point messages for fetching the object (or doing the read operation remotely). As a reliable multicast usually costs two physical messages, in total there are $2P + 2$ messages. In contrast, the update scheme requires only one reliable multicast message. So, even in the case that each write is followed by a read operation, the updating performs better than invalidation.

With the reliable multicast protocol we use, a multicast message is hardly more expensive than an RPC [Kaashoek et al. 1989b]. So, unless the read/write ratio of operations is close to zero, the update scheme will have a better performance. We have therefore only implemented the update scheme.

4.2. The Update Protocol

As in the Amoeba RTS, indivisibility of write operations is obtained by executing them in a mutually exclusive way. With indivisible multicast, mutual exclusion comes for free. The communication primitive imposes a single system-wide global ordering on all write operations. Unlike the point-to-point scheme, there is no risk of different processors updating their copies in an inconsistent way. Also, there is no need to distinguish between primary and secondary copies of an object,

The distributed update protocol we use works as follows. Each processor maintains a queue of messages that have arrived on the processor but that have not yet been handled. As all processors receive all messages in the same order, the queues on all processors are basically the same, except that some processors may be ahead of others in handling the messages at the head of the queue.

If a process wants to execute a write operation on a shared object *X*, it multicasts an *update* message to all processors (including its own processor) and then blocks. The message contains the name of the object, the operation, and its parameters. The update message will be appended to the tail of each queue.

Each processor handles incoming messages in its queue in strict FIFO order. A message may be handled as soon as it appears at the head of the queue. To handle an *update* message, the message is removed from the queue, the local copy of *X* is locked, the operation is applied to the local copy, and finally the local copy is unlocked. If the message was sent by a process on the same processor, that process is made active again.

The protocol described above correctly implements the serializability requirement. The protocol guarantees that all processors observe changes to shared objects *in the same order*. Note that it does not provide a total (*temporal*) ordering [Lamport 1978] among operations. Suppose Processor P1 initiates a write operation on object *X* and, a few microseconds later, Processor P2 reads the value of *X*. The *update* message for *X* sent by P1 need not have even reached P2 yet, so P2 may still use the old value of *X*. This scenario is in accordance with the semantics of our model, however, which merely requires serializability of operations.

4.3. Replication Strategy

The multicast RTS replicates all objects on all processors. In other words, it uses the full replication strategy. This strategy was chosen, because it simplifies the implementation. The RTS does not have to keep track of which object is used by which processor.

In some cases, full replication may be less efficient than partial replication. Suppose, for example, process P1 wants to send information to another process P2 through an object shared between them. As the object will be replicated everywhere, all processors in the system will receive P1's update message, even though only P2 is really interested in it.

The overhead of sending the message everywhere usually is not dramatic, however. With our reliable multicast protocol, the elapsed time for a multicast message hardly depends on the number of destinations. The main disadvantage of full replication then is the fact that each processor will be interrupted once for each write operation. With partial replication, this CPU overhead would be less.

5. PERFORMANCE

There are several ways to measure the performances of the replication techniques. The approach taken in [Bal] is to implement several user applications in Orca, execute them on the different run time systems, and measure the speedups. Applications we have looked at are matrix multiplication, the all-pairs shortest paths problem, branch-and-bound, alpha-beta search, and successive overrelaxation.

In this paper we will use an alternative approach by looking at the basic times for reading, writing, updating, and invalidating shared objects. In this way we can determine under which circumstances a technique is most effective.

To determine the performance improvements due to replication, we have performed two experiments. In the first experiment, we have measured the costs of incrementing a replicated 4 byte integer object as a function of the number of replicas. In the second experiment we measured the cost for updating an entire 1K array object. These two types of objects occur frequently in application programs [Bal et al. 1990].

5.1. Performance of the Amoeba Run Time System

Figure 2 shows the basic execution times for the run time system that uses Amoeba RPC. The figure shows the costs for invalidating *N* copies of an object and for updating 4-byte and 1Kb objects. Invalidating a copy involves sending a short message containing an object identifier, so the invalidation costs do not depend on the size of the object.

We have also measured the costs for doing a read operation on a remote object. For a 4-byte object, these costs are 2.7 msec; for a 1Kb object, the costs are 5.5 msec.

As expected, the costs to update or invalidate copies after a write operation grow linearly with the number of copies. Therefore, selective replication is worth while.

For a small (4-byte) object, updating 10 copies costs 30.7 msec. Invalidating 10 copies takes 15.7 msec; in addition, re-installing copies costs 2.7 msec per copy (i.e, the costs of a remote read operation). If the object is read by 6 or more processors immediately after it has been written, updating will outperform invalidation, since 15.7 + 6*2.7 > 30.7. (We assume that re-installations are done sequentially; in reality, there may be some overlap, so the re-installation costs may be less. However, this effect is rather small, since the processor containing the primary copy will become a sequential bottleneck if a large number of processors re-install the secondary copy.)

In contrast, if a small object is written twice without being read, the invalidation scheme is more efficient. In this case, the update costs are 2*30.7 = 61.4 msec. Invalidating and re-installing all 10 copies takes 15.7 + 10*2.7 = 42.7 msec.

For large (1 Kb) objects, updating 10 copies takes 37.9 msec. If 5 processors re-install the object after a write operation, the invalidation scheme costs 15.7 + 5*5.5 = 43.2, which is slower than updating. If fewer than 5 processors read the object, invalidation is more efficient. If a large object is written two or more times successively, invalidation is also better than updating.

A case that occurs frequently in user programs is a large object that is written through an operation with only a few bytes of parameters (e.g., a 1Kb array of which only 1 element is changed). In this case, updating will often be more effective. For example, updating 10 such copies will take about 30.7 msec, while invalidating and re-installing the entire array will cost 15.7 + 10*5.5 = 70.7 msec. Even if the object is changed twice before being read, updating is more efficient.

We can also compare the partial replication scheme with a scheme that does not replicate objects. If, for example, a given 4-byte object is not replicated, each (remote) read operation will take 2.7 msec. For an object that is read relatively frequently, partial replication clearly pays off.

Graph: [# gg 1 .. 10, 0 .. 55 define xy_gg @ (($1)-(.37))*.292398, (($2)-(-3.85))*0.0478469 @ define x_gg @ (($1)-(.37))*.292398 @ define y_gg @ (($1)-(-3.85))*0.0478469 @ frameht = 3 framewid = 3 Frame: box ht frameht wid framewid with .sw at 0,0 invis line from Frame.nw to Frame.ne invis line from Frame.sw to Frame.se line from Frame.sw to Frame.nw line from Frame.se to Frame.ne invis textht = .166667 textwid = .733333 Label: box invis wid 0 ht 2*textht "Time " "(in msec) " wid textwid with .e at Frame.w - (0.2,0) textht = .166667 Label: box invis wid 0 ht 1*textht "Number of copies" with .n at Frame.s - (0,2 * textht) ticklen = .1 Ticks_gg:line left ticklen from (0,y_gg(0)) "0 " rjust at last line.end line left ticklen from (0,y_gg(10)) "10 " rjust at last line.end line left ticklen from (0,y_gg(20)) "20 " rjust at last line.end line left ticklen from (0,y_gg(30)) "30 " rjust at last line.end line left ticklen from (0,y_gg(40)) "40 " rjust at last line.end line left ticklen from (0,y_gg(50)) "50 " rjust at last line.end ticklen = .1 Ticks_gg:line down ticklen from (x_gg(1),0) box invis "1" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(2),0) box invis "2" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(3),0) box invis "3" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(4),0) box invis "4" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(5),0) box invis "5" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(6),0) box invis "6" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(7),0) box invis "7" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(8),0) box invis "8" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(9),0) box invis "9" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(10),0) box invis "10" ht .25 wid 0 with .n at last line.end Lgg: xy_gg(1,.2) "□" at xy_gg(1,.2) line from Lgg to xy_gg(2,5.5); Lgg: Here "□" at xy_gg(2,5.5) line from Lgg to xy_gg(3,8.1); Lgg: Here "□" at xy_gg(3,8.1) line from Lgg to xy_gg(4,11.3); Lgg: Here "□" at xy_gg(4,11.3) line from Lgg to xy_gg(5,14.3); Lgg: Here "□" at xy_gg(5,14.3) line from Lgg to xy_gg(6,17.5); Lgg: Here "□" at xy_gg(6,17.5) line from Lgg to xy_gg(7,20.6); Lgg: Here "□" at xy_gg(7,20.6) line from Lgg to xy_gg(8,23.5); Lgg: Here "□" at xy_gg(8,23.5) line from Lgg to xy_gg(9,26.8); Lgg: Here "□" at xy_gg(9,26.8) line from Lgg to xy_gg(10,30.7); Lgg: Here "□" at xy_gg(10,30.7) Lgg: xy_gg(1,.17) "∆" at xy_gg(1,.17) line from Lgg to xy_gg(2,11.2); Lgg: Here "∆" at xy_gg(2,11.2) line from Lgg to xy_gg(3,14.2); Lgg: Here "∆" at xy_gg(3,14.2) line from Lgg to xy_gg(4,17.2); Lgg: Here "∆" at xy_gg(4,17.2) line from Lgg to xy_gg(5,20.1); Lgg: Here "∆" at xy_gg(5,20.1) line from Lgg to xy_gg(6,23.4); Lgg: Here "∆" at xy_gg(6,23.4) line from Lgg to xy_gg(7,26.8); Lgg: Here "∆" at xy_gg(7,26.8) line from Lgg to xy_gg(8,30.6); Lgg: Here "∆" at xy_gg(8,30.6) line from Lgg to xy_gg(9,34.6); Lgg: Here "∆" at xy_gg(9,34.6) line from Lgg to xy_gg(10,37.9); Lgg: Here "∆" at xy_gg(10,37.9) Lgg: xy_gg(1,.3) "×" at xy_gg(1,.3) line from Lgg to xy_gg(2,2.8); Lgg: Here "×" at xy_gg(2,2.8) line from Lgg to xy_gg(3,4); Lgg: Here "×" at xy_gg(3,4) line from Lgg to xy_gg(4,5.7); Lgg: Here "×" at xy_gg(4,5.7) line from Lgg to xy_gg(5,7.2); Lgg: Here "×" at xy_gg(5,7.2) line from Lgg to xy_gg(6,8.9); Lgg: Here "×" at xy_gg(6,8.9) line from Lgg to xy_gg(7,10.5); Lgg: Here "×" at xy_gg(7,10.5) line from Lgg to xy_gg(8,12.1); Lgg: Here "×" at xy_gg(8,12.1) line from Lgg to xy_gg(9,13.7); Lgg: Here "×" at xy_gg(9,13.7) line from Lgg to xy_gg(10,15.7); Lgg: Here "×" at xy_gg(10,15.7) Lgg: xy_gg(1,45) "□" at xy_gg(1,45) line from Lgg to xy_gg(2,45); Lgg: Here "□" at xy_gg(2,45) line from Lgg to xy_gg(3,45); Lgg: Here "□" at xy_gg(3,45) box invis wid 0 ht 1*textht "Time for 4-byte update" ljust at xy_gg(3.5,45) Lgg: xy_gg(1,50) "∆" at xy_gg(1,50) line from Lgg to xy_gg(2,50); Lgg: Here "∆" at xy_gg(2,50) line from Lgg to xy_gg(3,50); Lgg: Here "∆" at xy_gg(3,50) box invis wid 0 ht 1*textht "Time for 1Kb update" ljust at xy_gg(3.5,50) Lgg: xy_gg(1,55) "×" at xy_gg(1,55) line from Lgg to xy_gg(2,55); Lgg: Here "×" at xy_gg(2,55) line from Lgg to xy_gg(3,55); Lgg: Here "×" at xy_gg(3,55) box invis wid 0 ht 1*textht "Time for invalidation" ljust at xy_gg(3.5,55)

]

Fig. 2. Time for updating replicated objects using invalidation and 2-phase update protocol.

5.2. Performance of the Multicast Run Time System

The cost for updating replicated objects using the distributed update protocol described in Section 4 are depicted in Figure 3. As can be seen, the costs are almost independent of the number of replicas. This is what we should expect, since in our multicast protocol sending a reliable multicast message costs only two physical messages, independent of the number of receivers. The only overhead is sending one *state message* after a processor has received a certain number of messages. (This state message is only required if a processor does not multicast messages itself; if it does multicast messages, the state message is piggybacked.)

Graph: [# gg 1 .. 10, 0 .. 16 define xy_gg @ (($1)-(.37))*.292398, (($2)-(-1.12))*.164474 @ define x_gg @ (($1)-(.37))*.292398 @ define y_gg @ (($1)-(-1.12))*.164474 @ frameht = 3 framewid = 3 Frame: box ht frameht wid framewid with .sw at 0,0 invis line from Frame.nw to Frame.ne invis line from Frame.sw to Frame.se line from Frame.sw to Frame.nw line from Frame.se to Frame.ne invis textht = .166667 textwid = .6 Label: box invis wid 0 ht 2*textht "Time" "(in msec)" wid textwid with Frame.s - (0,2 * textht) ticklen = .1 Ticks_gg:line left ticklen from (0,y_gg(0)) "0 " rjust at last line.end line left ticklen from (0,y_gg(2)) "2 " rjust at last line.end line left ticklen from (0,y_gg(4)) "4 " rjust at last line.end line left ticklen from (0,y_gg(6)) "6 " rjust at last line.end line left ticklen from (0,y_gg(8)) "8 " rjust at last line.end line left ticklen from (0,y_gg(10)) "10 " rjust at last line.end line left ticklen from (0,y_gg(12)) "12 " rjust at last line.end line left ticklen from (0,y_gg(14)) "14 " rjust at last line.end line left ticklen from (0,y_gg(16)) "16 " rjust at last line.end ticklen = .1 Ticks_gg:line down ticklen from (x_gg(1),0) box invis "1" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(2),0) box invis "2" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(3),0) box invis "3" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(4),0) box invis "4" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(5),0) box invis "5" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(6),0) box invis "6" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(7),0) box invis "7" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(8),0) box invis "8" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(9),0) box invis "9" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(10),0) box invis "10" ht .25 wid 0 with .n at last line.end Lgg: xy_gg(1,.994) "□" at xy_gg(1,.994) line from Lgg to xy_gg(2,1.023); Lgg: Here "□" at xy_gg(2,1.023) line from Lgg to xy_gg(3,1.059); Lgg: Here "□" at xy_gg(3,1.059) line from Lgg to xy_gg(4,1.094); Lgg: Here "□" at xy_gg(4,1.094) line from Lgg to xy_gg(5,1.129); Lgg: Here "□" at xy_gg(5,1.129) line from Lgg to xy_gg(6,1.162); Lgg: Here "□" at xy_gg(6,1.162) line from Lgg to xy_gg(7,1.199); Lgg: Here "□" at xy_gg(7,1.199) line from Lgg to xy_gg(8,1.221); Lgg: Here "□" at xy_gg(8,1.221) line from Lgg to xy_gg(9,1.269); Lgg: Here "□" at xy_gg(9,1.269) line from Lgg to xy_gg(10,1.304); Lgg: Here "□" at xy_gg(10,1.304) Lgg: xy_gg(1,12.68) "∆" at xy_gg(1,12.68) line from Lgg to xy_gg(2,12.71); Lgg: Here "∆" at xy_gg(2,12.71) line from Lgg to xy_gg(3,12.75); Lgg: Here "∆" at xy_gg(3,12.75) line from Lgg to xy_gg(4,12.79); Lgg: Here "∆" at xy_gg(4,12.79) line from Lgg to xy_gg(5,12.81); Lgg: Here "∆" at xy_gg(5,12.81) line from Lgg to xy_gg(6,12.86); Lgg: Here "∆" at xy_gg(6,12.86) line from Lgg to xy_gg(7,12.9); Lgg: Here "∆" at xy_gg(7,12.9) line from Lgg to xy_gg(8,12.93); Lgg: Here "∆" at xy_gg(8,12.93) line from Lgg to xy_gg(9,12.96); Lgg: Here "∆" at xy_gg(9,12.96) line from Lgg to xy_gg(10,13); Lgg: Here "∆" at xy_gg(10,13) Lgg: xy_gg(1,14.5) "□" at xy_gg(1,14.5) line from Lgg to xy_gg(2,14.5); Lgg: Here "□" at xy_gg(2,14.5) line from Lgg to xy_gg(3,14.5); Lgg: Here "□" at xy_gg(3,14.5) box invis wid 0 ht 1*textht "Time for 4-byte operation" ljust at xy_gg(3.5,14.5) Lgg: xy_gg(1,15.5) "∆" at xy_gg(1,15.5) line from Lgg to xy_gg(2,15.5); Lgg: Here "∆" at xy_gg(2,15.5) line from Lgg to xy_gg(3,15.5); Lgg: Here "∆" at xy_gg(3,15.5) box invis wid 0 ht 1*textht "Time for 1Kb operation" ljust at xy_gg(3.5,15.5)

]

Fig. 3. Time for updating replicated objects using the distributed update protocol.

For small objects, the update costs are lower than the costs for remote read and write operations (see Section 5.1), because there is no operating system overhead. Consequently, for small objects, replication usually reduces the communication costs. One exception is an object with a low read/write ratio. In this case, the overhead of updating the replicas after each write will invalidate the gains of replication. The second exception is an object that is hardly ever accessed by remote processors. In both cases, it would be better not to replicate the object at all. In conclusion, a good strategy for small objects would be to either replicate a given object everywhere or not at all.

For larger objects, our multicast protocol is slightly less efficient, since it uses two large network packets for updating copies. So, the update costs are higher than the costs for reading or writing remote objects. With 10 CPUs, for example, updating a 1Kb object costs 13 msec; reading a remote 1 Kb object using Amoeba RPC costs approximately 5.5 msec (including marshalling overhead). Again, it depends on the read/write ratio whether replication is advantageous. If, for example, 3 of the 10 CPUs read the object immediately after it has been changed, replication is advantageous, since 3*5.5 > 13. Replication has the additional advantage of allowing more parallelism, since read operations on multiple local copies can be done simultaneously.

6. CONCLUSIONS

The model discussed in this paper allows programmers to define operations of arbitrary complexity on shared data-objects. In a loosely-coupled system, the model is implemented by replicating objects in the local memories of the processors. We have studied several protocols for keeping all these copies consistent and we have looked at replication strategies.

We have described two implementations of the model. One implementation replicates objects everywhere and updates copies through a fast multicast protocol. The other implementation uses only point-to-point messages. In this case, partial replication and migration may be useful.

Which protocol or strategy for replication is most efficient depends on many factors, such as the costs of the update protocols, the size of the object and the parameters of the operations, and the read/write pattern of the application. In the future we intend to do a more detailed analysis of our protocols and strategies, using a large set of user applications. Also, we will look at the differences and resemblances between protocols for replication and coherence protocols for CPU caches [Eggers and Katz 1989; Owicki and Agarwal 1989], file caches [Noe et al. 1985; Morris et al. 1986; Ousterhout et al. 1988], and distributed database systems [Bernstein and Goodman 1981]. Based on this analysis, we will try to improve our implementations.

Our model has several advantages over other models based on logically shared data, such as Shared Virtual Memory and Linda's Tuple Space [Ahuja et al. 1986]. Our model provides a higher level of abstraction and, in many cases, is more efficient.

The SVM, for example, can only invalidate but not update copies of data. Also, the SVM will perform very poorly if processes on many different processors repeatedly write on the same page. This situation arises if multiple processors write the same variable, or if they write different variables placed on the same page.

Linda's Tuple Space provides only a fixed number of low-level operations on shared data (tuples) [Kaashoek et al. 1989a]. Logical operations on shared data structures frequently consist of several low-level operations, each of which can require physical communication. In our model, the programmer can define a single high-level operation that does the job at less communication costs.

7. REFERENCES

Ahuja, S., Carriero, N., and Gelernter, D., Linda and Friends, *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, Aug. 1986.

Bal, H. E., Programming Distributed Systems, Silicon Press, Summit, NJ, 1990 (a minor revision of the author's Ph.D. thesis ‘‘The Shared Data-Object Model as a Paradigm for Programming Distributed Systems’’).

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., A Distributed Implementation of the Shared Data-object Model, *First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, Ft. Lauderdale, FL., Oct. 1989a.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., Experience with Distributed Programming in Orca, *Proc. IEEE CS 1990 Int. Conf. on Computer Languages*, New Orleans, LA, Mar. 1990.

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261-322, Sep. 1989b.

Bal, H. E. and Tanenbaum, A. S., Distributed Programming with Shared Data, *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, pp. 82-91, Miami, Fl., Oct. 1988.

Bernstein, P. A. and Goodman, N., Concurrency Control in Distributed Database systems, *ACM Comping Surveys*, Vol. 13, No. 2, pp. 185-221, June 1981.

Bisiani, R. and Forin, A., Architectural Support for Multilanguage Parallel Programming on Heterogenous Systems, *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 21-30, Palo Alto, Calif., Oct. 1987.

Carriero, N. and Gelernter, D., The S/Net's Linda Kernel, *ACM Trans. Comp. Syst.*, Vol. 4, No. 2, pp. 110-129, May 1986.

Cheriton, D. R., Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems, *ACM Operating Systems Review*, Vol. 19, No. 4, pp. 26-33, Oct. 1985.

Davidson, S. B., Garcia-Molina, H., and Skeen, D., Consistency in Partitioned Networks, *ACM Comping Surveys*, Vol. 17, No. 3, pp. 341-370, Sep. 1985.

Eggers, S. J. and Katz, R. H., A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation, *15th Int. Symp. on Computer Architecture*, pp. 373-382, Israel, June 1989.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., The Notions of Consistency and Predicate Locks in a Database System, *Commun. ACM*, Vol. 19, No. 11, pp. 624-633, Nov. 1976.

Fleisch, B. and Popek, G., Mirage: A Coherent Distributed Shared Memory Design, *Proc. 12th Symp. Operating System Principles*, pp. 211-223, ACM SIGOPS, Litchfield, AZ, Dec. 1989.

Gifford, D. K., Weighted Voting for Replicated Data, *Proc. 7th Symp. Operating Systems Principles*, pp. 150-162, ACM SIGOPS, Pacific Grove, CA, Dec. 1979.

Joseph, T. A. and Birman, K. P., Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems, *ACM Trans. Comp. Syst.*, Vol. 4, No. 1, Feb. 1987.

Kaashoek, M. F., Bal, H. E., and Tanenbaum, A. S., Experience with the Distributed Data Structure Paradigm in Linda, *First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 175-191, Ft. Lauderdale, FL., Oct. 1989a.

Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S., and Bal, H. E., An Efficient Reliable Broadcast Protocol, *ACM Operating Systems Review*, Vol. 23, No. 4, pp. 5-20, Oct. 1989b.

Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, *Commun. ACM*, Vol. 21, No. 7, pp. 558-565, July 1978.

Li, K., IVY: A Shared Virtual Memory System for Parallel Computing, *Proc. 1988 Int. Conf. Parallel Processing (Vol. II)*, pp. 94-101, St. Charles, Ill., Aug. 1988.

Li, K. and Hudak, P., Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Comp. Syst.*, Vol. 7, No. 4, Nov. 1989.

Morris, J. H., Satyanarayan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., and Smith, F. D., Andrew a Distributed Personal Computing Environment, *Commun. ACM*, Vol. 29, No. 3, pp. 184-201, Mar. 1986.

Mullender, S. J. and Tanenbaum, A. S., The Design of a Capability-Based Distributed Operating System, *The Computer Journal*, Vol. 29, No. 4, pp. 289-300, Mar. 1986.

Noe, J. D., Proudfoot, A. B., and Pu, C., Replication in Distributed Systems: The Eden Experience, TR-85-08-06, Dept. of Computer Science, University of Washington, Seattle, Sep. 1985.

Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B., The Sprite Network Operating System, *IEEE Computer*, Vol. 21, No. 2, pp. 23-37, Feb. 1988.

Owicki, S. and Agarwal, A., Evaluating the Performance of Software Cache Coherence, *Proc. 3nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 230-242, Boston, MA, Apr. 1989.

Stenstrm, P., Vrsalovic, D., and Segall, Z., Shared Data Structures in a Distributed System - Performance Evaluation and Practical Considerations, *Proc. Int. Sem. on Performance of Distributed and Parallel Systems*, pp. 15-29, Kyoto, Japan, Dec. 1988.

Van Renesse, R. and Tanenbaum, A. S., Voting with Ghosts, *Proc. of the 8th Int. Conf. on Distributed Computing Systems*, pp. 456-462, San Jose, CA, June 1988.