Exploiting Temporal & Spatial Constraints on Distributed Shared Objects*

Richard West, Karsten Schwan, Ivan Tacic & Mustaque Ahamad

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Gigabit network technologies have made it possible to combine workstations into a distributed, massively-parallel computer system. Middleware, such as distributed shared objects (DSO), attempts to improve programmability of such systems, by providing globally accessible 'object' abstractions. Researchers have developed consistency protocols for replicated 'memory' objects. These protocols are well suited to scientific applications but less suited to multimedia or groupware applications. This paper addresses the state sharing needs of complex distributed applications with (1) high-frequency symmetric accesses to shared objects, (2) unpredictable and limited locality of accesses, (3) dynamically changing sharing behavior, and (4) potential data races. We show that a DSO system exploiting application-level temporal and spatial constraints on shared objects can outperform shared object protocols which do not exploit application-level constraints. We compare our S(emantic) DSO against entry consistency using a sample application having the four properties mentioned above.

1 Introduction

Gigabit network technologies have made it possible to combine workstations into a distributed, massively-parallel computer system. These platforms are difficult to program, because implementors must explicitly code complex message passing protocols to exchange shared state information between processes. Distributed shared objects (DSO) have the potential of simplifying the implementation of such shared state, by providing globally accessible 'object' abstractions. A run-time system is responsible for maintaining consistency across distributed object components.

Early research on distributed shared objects concerned 'memory' objects accessible via 'read' and 'write' operations (DSM) [1] or 'fragmented' objects offering relatively simple operational interfaces[2, 3]. Since then, object-based research has moved toward more general representations of shared abstractions, including the support of arbitrary type hierarchies in object access[4]. Simultaneously, researchers have investigated the efficient implementation of DSM, including the development of efficient consistency maintenance protocols[5, 6, 7, 8, 9], experimentation with alternative representations of shared memory pages[10] and with alternative methods for dealing with specific implementation issues, like false sharing[11].

This paper focuses on the required consistency of logically shared state information in complex distributed applications that range from irregular parallel scientific codes to distributed groupware environments[12]. Specifically, applications considered in our research include (1) distributed multimedia games; (2) virtual environments; (3) distributed real-time command and control; and (4) high performance applications exhibiting dynamic data access patterns[13]. The characteristics of such applications relevant to our work are:

- *Poor and unpredictable locality*: Distributed processes may read and write arbitrary shared objects at any time, thereby making it difficult to cache shared state.
- *Symmetric data access*: All processes can read and write 'memory' objects, unlike in an asynchronous client-server system, where the clients might only be allowed to read and the server can read and write to shared objects.
- *Dynamic changes in sharing behavior*: Accesses to shared objects change with time.
- *Data races may occur*: Data races occur when two or more processes attempt to access the same memory location, where at least one process is performing a 'write' to that location[14, 6].

These characteristics lead to differences in assumptions

*This work is supported in part by the Engineering and Physical Sciences Research Council grant 92600699 and DARPA contract DABT63-95-C-0125.

compared to past research on DSM systems. Most notably, existing DSM systems assume that the programs using them are data race free, because this property is easily guaranteed if programs synchronize on access to shared state. However, it is not easy to avoid data-races in the more complex multimedia or collaborative applications investigated in our work[15].

The central tenet of our work is that high levels of concurrency and scalability for complex distributed applications may be attained if programmers can exploit the specific semantics of these applications when implementing and using shared state abstractions. In order to focus on the exploitation of application-level semantics, we utilize the relatively 'simple' model of shared 'memory' objects (e.g., bitmaps) accessed with 'read' and 'write' operations and not subject to run-time resolution of type hierarchies. For such shared abstractions, application-level semantics are used for deciding *when* and *who* should be informed of updates to them. In particular, we define the notions of 'temporal' and 'spatial' consistency, which jointly capture a wide range of consistency knowledge and constraints about shared state in complex distributed programs. A S(emantic)-DSO system supporting efficient encodings of temporal and spatial consistency knowledge is developed. Novel consistency protocols using this knowledge are implemented and evaluated in comparison to protocols not using such information. With the S-DSO system, we then show that complex distributed applications using it perform comparably well to programs using explicit message passing, with improved programmability compared to such programs.

In the remainder of this paper, we first present a sample distributed program that is representative of the complex distributed applications addressed by our research. The notions of 'temporal' and 'spatial' consistency are then defined, followed by a description of the S-DSO system in Section 3. The S-DSO system and novel consistency protocols using these notions are evaluated in Section 4 in comparison to a well-known DSM protocol, entry-consistency (EC)[5]. Section 5 describes related work, while conclusions and future research appear in Section 6.

2 Memory Consistency for Interactive Distributed Applications

2.1 A Sample Application

Our sample application is a multi-player game with a shared environment (see Figure 1), derived from complex command and control applications and from interactive distributed simulations. The object of the game is for each player to maneuver her team of tanks to some known goal as quickly as possible, while picking up bonus items and avoiding obstacles and enemy tanks along the way. At any one time, each player need only know about the positions of enemy tanks in close proximity, since these tanks pose a threat to the local team. Novel characteristics of this application include its exploitation of user-specified attributes to improve the performance of consistency maintenance for its replicated objects, its high levels of concurrency and asynchrony while performing what appear to be sequentially consistent actions on shared data, and its scalability in terms of the amounts of data shared as well as the number of parties accessing shared data.

The game's efficient implementation utilizes a *lookahead* consistency protocol developed using S-DSO's attribute infrastructure. We use the term 'lookahead' to describe any protocol that has the ability to predict the future times at which groups of processes must exchange information regarding modifications to shared objects that each process may later need. Our lookahead protocol's realization exploits spatial and temporal application-level semantics to improve the game's concurrency and asynchrony in execution. Namely, with lookahead consistency, a process communicates only with the *subset* of other processes currently accessing the objects it must know about in the immediate future. Moreover, such processes do not synchronize at the access to a shared object unless they require knowledge about that object's current state. Thus, information about a shared object is only made known to a process if and when it is required. The 'spatial' constraints on consistency for this application are well defined. Namely, if one team's tanks move to within distance d of another team's tanks, each such team must know the exact position of the other team. When tanks

Figure 1: A sample distributed multimedia application: a video game

are separated by a distance larger than d, it is not necessary to update each team with the locations of enemy tanks. From this example, it is clear that such formulations of 'spatial' locality are possible only in reference to application-level program semantics. Clearly, the 2D shared environment used in the interactive game gives rise to fairly simple functions with which spatial consistency constraints may be computed, and similarly straightforward formulations may be found for 3D shared environments like those used in shared virtual environments. Even scientific applications exhibit such spatial consistency constraints, as is evident in n-body simulations, where the gravitational effects of bodies on each other are considered only when two bodies are within minimum distance d of each other. Likewise, molecular dynamics simulations tend to consider only those interactions of molecules within some known cut-off radius.

The distributed game exhibits the four features described in Section 1. Firstly, there is poor and unpredictable locality to shared objects, where each object is at the granularity of a team's tank. Secondly, each user can manipulate her own team of tanks independently of other users, which results in symmetric data accesses. Thirdly, at any point in time, one team is only concerned with a subset of all possible shared objects within range, but that set of objects may change with time. Consequently, the DSO system must deal with dynamic changes in sharing behavior. Finally, data races may occur, since two tanks may attempt to move to the same location in the shared environment.

2.2 Temporal and Spatial Consistency

All DSM systems deal with *temporal consistency*. Temporal consistency determines *when* (and in what order) changes to a shared object are made visible to all processes interested in that object. However, this says nothing about *which* processes should be informed of changes, and a system that does not utilize such knowledge can only assume that changes must be known by all processes that have a copy of the shared resource. Spatial consistency considers this latter problem.

Spatial consistency determines *which* processes should be updated with changes to shared objects based on the locations of those objects in the shared space. For example, a contiguous block of memory, such as an array, may be shared between two processes P_i and P_j. A write to any element of this memory by P_i should be made visible to P_j, if the change is to an element in a range of elements that P_j needs to know about. Likewise, if two moving objects in a virtual environment are within a certain distance of each other, each object must know about the other. This 'spatial' consistency property is stated more precisely below in the context of the distributed game.

Consider a process P_i that writes to a location x at time t. At time $t+\tau$, P_j reads location x and, based on the value read, generates a write to one of two possible locations, y or z. Alternatively, P_j may decide to write to location y no matter what the value of x, but the actual value written to y does depend on the value P_j sees at x. Thus, we have a dependency between the write of P_i at time t and the write of P_j at time $t+\tau$, based on the read of x by P_j also at time $t+\tau$[1]. However, if P_i writes to a location x' at time t, out of the range of necessary locations that P_j must know about for its next operation, P_j can avoid being updated with the new value at location x'. Spatial constraints permit P_i and P_j to see inconsistent views of the value at x' at time $t+\tau$.

Defining the value of τ is critical for the performance of a system in which updates must be exchanged in this manner. Between t and $t+\tau$, P_i and P_j can be inconsistent, but at $t+\tau$ they must both see the same values in all locations that affect their next write operations.

We can define τ as the interval of time in which two or more processes may concurrently perform a write operation. These write operations will be based on the states of each process' local copy of the shared environment at time t. Only at time $t+\tau$ will each process P_i be required to synchronize with those processes that have generated write operations that may affect $P_i's$ operations in the interval $[t+\tau, t+2\tau]$.

In the worst case, each process must barrier synchronize with every other process after each interval τ, in which each process performed exactly one read of the shared environment, followed by one write. Synchronization at time t is required for two reasons: (1) to update the state of each process' copy of the shared environment, thereby ensuring any writes in the interval $[t, t+\tau]$ are based on correct previous states, and (2) to identify new locations in the shared environment where access races may occur as a result of the dynamically changing environment.

The synchronization actions described above do not eliminate data races, because in any period τ, two or more processes may be performing concurrent accesses to the same location, where at least one is a write. In such circumstances, only one process may access the common shared object. All other processes must block or perform access operations on other locations in shared space. We adopt a simple policy of assigning integer IDs to processes and block all processes except the one with the highest ID.

[1] This assumes the real-time taken to perform a read before a write is negligible.

2.3 Conventional Consistency Protocols

Previous consistency protocols for DSM systems have been designed for scientific rather than multimedia or groupware applications. Three prominent protocols are causal memory[6], entry(EC)[5], and lazy release consistency (LRC)[8].

In causal memory, data race avoidance using lock management schemes can severely curtail the potential levels of asynchrony, concurrency, and scalability of applications.

Entry consistency (EC) explicitly deals with data races by associating distributed locks with shared objects, thereby enforcing consistency among those objects while also permitting the concurrent execution of processes that lock disjoint sets of objects. However, in comparison to our 'lookahead' schemes, entry consistency does not deal well with multiple shared objects that have spatial relationships subject to dynamic change. Namely, any process requiring access to multiple and consistent objects must explicitly acquire locks on these objects prior to operating on them. The blocking overheads of lock acquisition can be severe if the number of required locks is large. In addition, potential deadlocks must be resolved.

Like EC, LRC also uses locks to synchronize accesses to shared objects and thereby, enforce consistency. With LRC, updates to shared data are propagated when locks are transferred between processes. However, unlike EC, LRC does not explicitly associate shared data items with synchronization primitives, so that it must include information concerning the changes to all shared data objects with lock operations. We therefore, do not compare LRC performance to the S-DSO approach.

3 The S-DSO Framework

3.1 System Description

The *S-DSO* infrastructure enables the specification and use of application-level consistency semantics in conjunction with the execution of consistency protocols. Specifically, it permits end users to write application-specific functions, called *semantic functions* or *s-functions*, that can be invoked by consistency protocols. These functions allow users to state *when* each process must see the most recent updates to *which* of the objects being shared, thereby eliminating unnecessary message exchanges and increasing asynchrony, concurrency, and scalability in distributed applications. Namely, an *s-function* calculates the future times at which a process must send to and receive from other processes the updates to the various objects being shared. S-functions are stated using the S-DSO system's *exchange* call:

```
void exchange (obj_t *shared_obj, boolean resync_flag,
               send_t how, void (*s_func)(), any_t arg);
```

The *exchange* function takes, as its first argument, a pointer to the shared object. If 'resync_flag' is true, then exchange() will wait for all remote processes that receive updates at the current (logical) time to exchange any object updates they have made since the local process last exchanged with them. S-DSO uses *s_func* to calculate when to exchange (and, hence, resynchronize) in the future with the processes with which it has just performed exchange operations. Since *s_func* is specified by the user, it may convey to the consistency protocol the semantic attributes it should use for calculating future synchronization times with other processes. The *arg* argument to exchange() is the argument associated with *s_func*. The purpose of 'resync_flag' is two switch between two modes of operation: *push* and *push-pull*. If it is true, a synchronous push-pull is used. If it is false, exchange simply *pushes* changes out to appropriate processes, and the local process continues to execute asynchronously.

The exchange function is used as follows. Every time an application process modifies a shared object, it calls exchange(), and a logical system clock is advanced one time-tick. Each process maintains a local logical clock that essentially counts phases of the local process (i.e., the number of distinct periods of time, τ, in which one modification to a shared object requires updates to be sent to remote processes either now or some $n\tau$ periods in the future). This definition of τ is consistent with the definition in Section 2.2. The modified object is referenced by the first argument to exchange(). The S-DSO system can use the s-function specified in the exchange call to determine whether or not the updated object information should be sent to remote processes that have a local copy of that object.

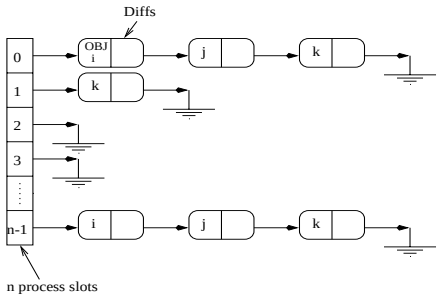

Figure 2: A sample slotted-buffer at local process 2. There is nothing to send to process 3.

S-DSO maintains a time-ordered list of (exchange-time, process) pairs for each process that must be up-

dated with object modifications in the future. The object modifications themselves must be buffered if they are not immediately sent to remote processes. Accordingly, S-DSO maintains a slotted buffer at each process for outstanding modifications to be exchanged with remote processes (see Figure 2). There is one slot in the buffer for each remote process. To reduce buffering needs, the buffered changes are *diffs* of the state of each object since their previous modification. In each slot is the list of modifications about which the corresponding process must be informed when it needs the latest information on those objects. S-DSO uses the s-function to calculate the times at which each list of buffered changes must be flushed (i.e., sent to the corresponding remote process). With this model, object changes are not sent to processes that will never need them. Furthermore, exchange() only exchanges with the subset of processes that need to know the changes immediately. Also note that each process may need knowledge of different objects at different times. This implies that the buffered changes maintained by S-DSO differ across processes at any one point in time. Furthermore, S-DSO can be tuned to merge multiple diffs to the same object into one diff since the last exchange with a given process. This kind of optimization is especially useful for real-time applications and games, since many such applications will not consider 'old' values when newer values of shared objects are available. Last, observe that an application process does not explicitly specify which processes receive changes to shared objects. Instead, the s-function is used to determine *who* receives the information.

To override the multicasting capabilities of exchange(), the *how* argument can be set to 'broadcast'. This forces the modifications to the object referenced by *shared_obj* as well as all buffered modifications to be immediately flushed to all remote processes. Under normal operation, the *how* argument is set to 'multicast'.

The exchange() function will update remote object copies for all objects that must be updated at the current time. If the *resync_flag* is true, exchange() will then block until all such remote processes have exchanged their buffered modifications with the local process. At this point, the local process is consistent with the remote processes involved in the exchange, for those objects just exchanged by this group of processes. The exchange() function will then use the s-function to recalculate (for each local process) the future time at which it must re-exchange information with this same set of processes. Note that every time an application process calls exchange(), it may do so with a different s-function.

3.2 Lookahead Consistency

We have implemented several semantic-based consistency protocols using the S-DSO system. These protocols are tailored to the sample video game described in Section 2.1. The first protocol, called BSYNC, broadcasts all object updates to every other process after each object modification. The s-function for this protocol is only used to establish when data-races can occur and thus avoids them without recourse to a locking protocol. Each time the local process broadcasts a synchronous update, it blocks until all other processes have responded with their updates. In this way, each process exchanges with every other process after each object modification. When two processes are in contention for the same object[2], we arbitrarily block the process with the lowest ID, while the other process generates an event that potentially modifies the common object. During update exchanges, blocked processes simply exchange control (SYNC) messages and wait for all other processes to respond with their data updates and/or SYNC messages. Unblocked processes send a SYNC control message paired with a data message. SYNC messages delimit one logical clock phase from the next.

In any one time quantum τ, the BSYNC protocol allows all processes to perform concurrent object writes, with each process performing at most one object modification before broadcasting its updates and waiting for responses from everyone else. Under this policy, a process can be executing in a time quantum at most one τ earlier or later than any other process. Thus, all processes' logical clocks are synchronized to within one time-tick, and their real-time clocks are synchronized to within τ seconds. This means that integer-valued logical timestamps must be sent with each update, to ensure that any early updates (by at most one logical time-tick) are not applied to object copies too soon. This also means that each process must buffer at most one early message from every other process. Unlike other consistency protocols, BSYNC does not require vector-timestamps and does not require unbounded buffer space for early update messages.

We also developed two lookahead protocols, MSYNC and MSYNC2 that use the exchange-list and slotted-buffer provided by S-DSO. In the video game application, the s-function for MSYNC computes the logical exchange times with each process (i.e., team of tanks) by halving the distance between the nearest tanks in any two teams. This approach is based

[2]For our application, this occurs when two enemy tanks are within one block of each other. In this case, a block is a single object in a two-dimensional array representing the shared environment for the tanks.

on the assumption that, in the worst-case, one team's closest tank to an enemy will always move towards the other team's closest tank, and vice versa. Every logical time-tick, each team's tank moves from one block to another in the shared virtual environment. In the context of the video game, MSYNC assumes that any enemy tank in the same row or column of the shared environment as a local tank can potentially affect a local tank's next operation. MSYNC2 refines this assumption by only exchanging tank locations and their image information with those processes whose tanks could have moved into the same row or column as a local tank, and the distance to those enemy tanks is less than d blocks.

4 S-DSO Experimental Evaluation

All measurements presented in this section are conducted on a cluster of 16 SGI Indy workstations (each with a single MIPS R4400 processor and 64MBytes of memory) connected via 10Mbps Ethernet, and using TCP. In each of the experiments, the video game is configured to run non-interactively, and the 2D shared environment consists of 32x24 blocks (shared objects). There is one team per process and one process per physical processor, so that every process runs on its own machine in the workstation cluster. In all cases, team size is fixed to one tank. Each tank's objective is to reach the goal as quickly as possible, while trying to acquire as many bonus points as possible and avoid being destroyed by enemy tanks. For correct operation of the application, it is mandated that each tank base its decision to move, rotate, or fire at an enemy by looking a certain number of blocks in each of four directions: north, south, east and west. This implies that, at the very least, all blocks within range in each direction have to be consistent when the corresponding tank looks at the contents of those blocks. Each tank performs a simple iteration each logical clock-tick: (1) look at all the blocks within range in each direction, north, south, east and west; (2) generate a task to modify a block object; and (3) goto (1), unless the goal is reached or tank is destroyed. The consistency protocol ensures that the necessary blocks, in the range of a tank, are all always consistent.

Figure 3 shows the average execution times (in seconds) of processes, normalized by the average number of object modifications performed by each process. Normalization eliminates random effects, such as favorable locations of tanks (e.g., when their initial locations are close to the goal). For all cases, we use the same random seed value to place the teams of tanks in the shared environment. In all figures, each tank can

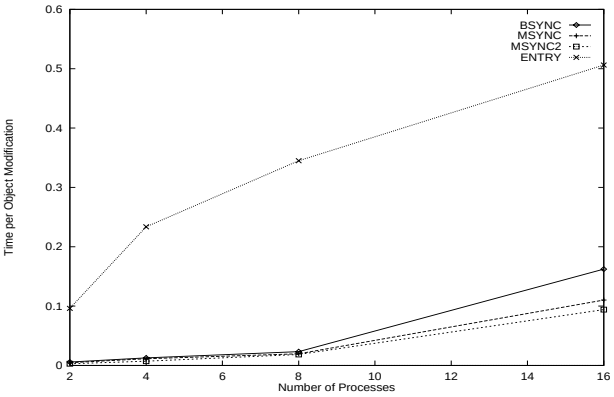

Figure 3: Average execution time per process normalized by average number of object modifications

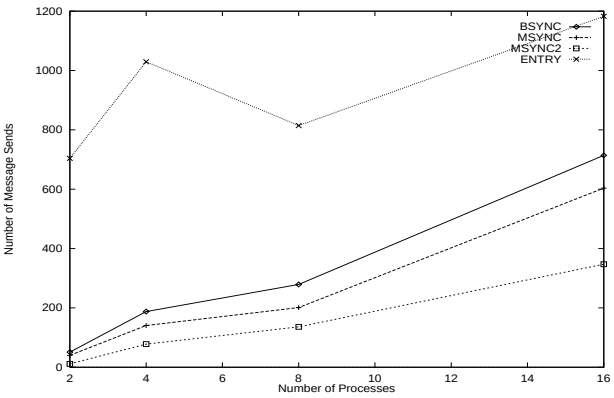

Figure 4: Total control and data message transfers by each protocol

'see' three objects away in each direction[3]. This means that for entry consistency, each process must lock 13 objects before its tank can move; one lock is for the location of the tank itself, four other write-locks are for all adjacent locations in which a tank might move, and the other locks are read locks. Locking is necessary to prevent two tanks from attempting to move to the same block.

From Figure 3, it is clear that entry consistency performs worse than all of the semantically richer synchronous 'lookahead' protocols, when the number of processes varies from 2 to 16. From the gradients of the graphs, it appears that entry consistency will remain worse than the other three protocols for greater than 16 processes. Note that MSYNC2 exhibits the highest performance, because its s-function captures application-level behavior more precisely than the functions used for MSYNC and BSYNC and

[3]For brevity, interpret 'range x' as corresponding to x visible objects in each direction.

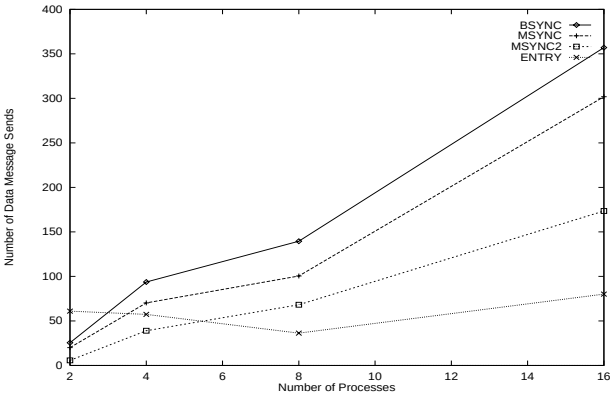

Figure 5: Total data message transfers by each protocol

thereby avoids sending unnecessary updates to remote processes.

Figure 4 depicts the total number of message transfers for each protocol, as a function of the number of processes in each experiment. In these graphs, the number of messages is the total number of control and data messages used by each consistency protocol. In all cases, the average data size is the same as the average control message size; both are 2048 bytes. Once again, entry consistency performs worse than the other protocols, since more message transfers take place. The major overhead to entry consistency is the number of lock-acquire messages it must send around the network to acquire all of the objects each process needs for each of its iterations. With our entry consistency implementation, the lock-managers for each lock-able object are evenly and statically spread across all host machines. With n processes there is a $1/n$ chance of the lock manager residing on the same machine as the process requesting the lock. Thus, as the number of dynamically shared objects increases, the majority of the lock-acquire messages are sent to remote machines. As a consequence, for 16 processes and when the number of shared objects is increased, entry consistency sends far more control messages than even BSYNC. Once again, MSYNC2 performs better than all of the other algorithms, at the cost of using a slightly more complicated s-function algorithm.

Figure 5, shows only the number of data messages transferred by each protocol, as a function of the number of processes. It is interesting to note that entry consistency transfers the fewest number of data messages overall, in both graphs. The reason lies in the fact that entry consistency is a 'pull-based' protocol that only pulls updates to objects when it is certain it needs them. The three lookahead protocols are sending updates to objects unnecessarily, even in the case of MSYNC2. The problem with MSYNC2 is that it assumes worst-case prediction of the minimum time that one object modification will affect another. Clearly, with the tank application, it isn't certain that two tanks will ever come within range of each other, but MSYNC2 assumes that this will eventually happen. Entry consistency, on the other hand, avoids updating objects that are never modified.

From experiments, we conclude (without showing) that the major overheads for entry consistency are acquiring locks, and retrieving object updates upon acquiring those locks to modified objects. The locking overhead for entry consistency rises when the number of dynamically shared objects increases, since more locks have to be acquired each time. For the other three protocols, the cost of exchanging updates dominates the runtime cost. MSYNC2 has lower overheads compared to MSYNC and BSYNC because, on average, each process needs to rendezvous at an exchange time with a smaller number of other processes. In all cases communication overheads dominate execution time, since local processing is minimal.

The effectiveness of any lookahead protocol is dependent on how accurately we can predict the worst-case time of pairwise accesses by two or more processes to the same shared object, or at least spatially-related objects. If the data size is small but the number of dynamically shared objects is large, it would appear a lookahead protocol with a suitable s-function allows far greater concurrency and scalability that a pull-based protocol like entry consistency. For large numbers of dynamically shared objects, we believe that entry consistent processes are spending far greater amounts of time in blocked modes, while waiting for locks that are potentially held by other processes. Although a lookahead scheme might send more data messages under the same conditions, it is able to do so with far less blocking overhead and therefore, exhibit better performance than the entry consistent scheme.

5 Related Work

Yavatkar[16] explores the notion of Δ-causality[17] in unreliable networks, supporting multimedia real-time collaborative applications. Δ-causality differs from our work by dealing with the delivery of messages that respect causal ordering only for messages received within their deadline-time, Δ. Messages arriving greater than Δ time units after their original transmission are never delivered in such a system. Yavatkar has built a Multi-Flow Conversation Protocol (MCP) to support the temporal synchronization of multimedia collaborative applications, with the explicit ability to support Δ-causally-ordered message transfers.

Griffoen et al[18] describes the Unify system for exploring scalable approaches to designing distributed multicomputer systems. They mention spatial and temporal consistency, but their notion of spatial consistency differs from ours in that it determines the relative order of data contained in various replicated objects, such as log files, associative and sequentially-ordered memories. In comparison, this paper's formulation of spatial consistency determines when processes should be updated with changes to shared objects, based on the locations currently accessed by these processes in shared space.

6 Conclusions and Future Work

We have implemented a framework, called S-DSO, which permits applications to specify the semantics of when and which processes should see updates to shared memory objects of varying sizes, using additional parameters associated with object accesses, called 'attributes'. We have shown that for applications with dynamic sharing behavior, poor spatial locality, dataraces, and symmetric object accesses, conventional DSM protocols like entry-consistency may be inadequate, particularly when there is a large number of shared objects.

The S-DSO system presented in this paper will be one of several configurable substrates of the CORBA-compliant, distributed object system for high performance applications now being developed at Georgia Tech, called COBS. Once integrated within the COBS framework, S-DSO applications may execute on heterogeneous distributed platforms, across a variety of network links. As a result of this integration, we expect to be able to investigate the effects of wide area and high performance communication media on consistency protocols for the types of applications described in this paper.

We are also investigating the use of arbitrary graph structures to capture the spatial relationships between objects. Each shared object is represented by a graph node and an edge joins two nodes n_i and n_j, if the object represented by n_j can be accessed immediately after the object represented by n_i. We are looking at ways to decide when to update and exchange these replicated graph structures, along with the changes to the objects themselves.

References

[1] K. Li, "Ivy: A shared virtual memory system for parallel computing," in *ICPP*, pp. II 94–101, Aug 1988.

[2] C. Clemencon, B. Mukherjee, and K. Schwan, "Distributed shared abstractions (DSA) on large-scale multiprocessors," in *Proc. of the Fourth USENIX Symposium on Experiences with Dist. and Multiprocessor Systems*, pp. 227–246, USENIX, Sept 1993.

[3] M. Shapiro, "Structure and encapsulation in distributed systems: The proxy principle," in *Proc. 6th IEEE ICDCS, Boston, Mass.*, pp. 198–204, IEEE, May 1986.

[4] J. Siegel, *CORBA - Fundamentals and Programming*. John Wiley and Sons, 605 Third Ave, New York, NY 10158. ISBN 0471-12148-7, 1996.

[5] B. N. Bershad and M. J. Zekauskas, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," Tech. Rep. CMU-CS-91-170, CMU, Sept 1991.

[6] M. Ahamad, G. Neiger, P. Kohli, J. E. Burns, and P. W. Hutto, "Causal memory: Definitions, implementation and programming," *Distributed Computing*, vol. 9, pp. 37–49, Aug 1995.

[7] K. Birman, A. Shiper, and P. Stephenson, "Lightweight causal and atomic group multicast," Tech. Rep. 91-1192, Department of Computer Science, Cornell University, Feb. 1991.

[8] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proc. 19th ISCA*, 1992.

[9] J. K. Bennett, J. B. Carter, and W. Zwaenpoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pp. 168–176, March 1990.

[10] P. Kohli, M. Ahamad, and K. Schwan, "Indigo: User-level support for building distributed shared abstractions," in *Fourth IEEE International Symposium on High-Performance Distributed Computing*, Aug 1995.

[11] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in *4th Symposium on Experimental Distributed and Multiprocessor Systems*, pp. 57–71, Sept 1993.

[12] S. Gronemberg and D. Marwood, "Real-time groupware as a distributed system: Concurrency control and its effect on the interface," in *Proceedings of the ACM Conference on Cooperative Support for Cooperative Work, ACM press*, pp. 207–217, ACM, 1994.

[13] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandam, "Distributed memory compiler design for sparse problems," *IEEE Transactions on Computers*, vol. 44, pp. 737–753, June 1995.

[14] S. V. Adve and M. D. Hill, "Weak ordering - a new definition," in *Proc. 17th ISCA*, pp. 2–14, May 1990.

[15] B. Schroeder, G. Eisenhauer, J. Heiner, V. Martin, K. Schwan, and J. Vetter, "From interactive applications to distributed laboratories." Submitted to the Visual Supercomputing special issue of IEEE Computational Science and Engineering - June 1996, 1996.

[16] R. Yavatkar, "MCP: A protocol for coordination and temporal synchronization in multimedia collaborative applications," in *Proc. 12th IEEE ICDCS*, pp. 606–613, IEEE, 1992.

[17] R. Baldoni, A. Mostefaoui, and M. Raynal, "Causal delivery of messages with real-time data in unreliable networks," *Real-Time Systems, The International Journal of Time-Critical Computing Systems*, vol. 10, pp. 245–262, May 1996.

[18] J. Griffoen, R. Yavatkar, and R. Finkel, "Extending the dimensions of consistency: Spatial consistency and sequential segments," Tech. Rep. cs248-94, University of Kentucky, April 1994.