to appear in SPAA'97

Temporal Notions of Synchronization and Consistency in Beehive*

Aman Singla
Umakishore Ramachandran †
Jessica Hodgins

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332-0280
e-mail: *aman,rama,jkh@cc.gatech.edu*

An important attribute in the specification of many compute-intensive applications is "time". Simulation of interactive virtual environments is one such domain. There is a mismatch between the synchronization and consistency guarantees needed by such applications (which are temporal in nature) and the guarantees offered by current shared memory systems. Consequently, programming such applications using standard shared memory style synchronization and communication is cumbersome. Furthermore, such applications offer opportunities for relaxing both the synchronization and consistency requirements along the temporal dimension. In this work, we develop a temporal programming model that is more intuitive for the development of applications that need temporal correctness guarantees. This model embodies two mechanisms: "delta consistency" – a novel time-based correctness criterion to govern the shared memory access guarantees, and a companion "temporal synchronization" – a mechanism for thread synchronization along the time axis. These mechanisms are particularly appropriate for expressing the requirements in interactive application domains. In addition to the temporal programming model, we develop efficient explicit communication mechanisms that aggressively push the data out to "future" consumers to hide the read miss latency at the receiving end. We implement these mechanisms on a cluster of workstations in a software distributed shared memory architecture called "Beehive." Using a virtual environment application as the driver, we show the efficacy of the proposed mechanisms in meeting the real time requirements of such applications.

1 Introduction

The shared memory programming paradigm is useful for a large class of parallel applications. Models of synchronization and consistency in this paradigm are quite adequate for expressing the requirements in many scientific and engineering applications. The programming model presented to the application is one in which the mechanisms for synchronization and consistency are based purely on the order of access to *shared data* and *synchronization variables*. Therefore, we will use the label *spatial* while referring to the standard shared memory systems and the corresponding programming model.

Several applications that are enablers for "everyday computing" have real time constraints inherent in their specification. Simulation of interactive virtual environments, interactive speech recognition, and interactive vision are examples of such applications. As the problem size is scaled up, these applications impose tremendous computational requirements that are beyond the realm of sequential processing. These applications are prime candidates for parallel computing. Being interactive in nature, *time* is an important attribute of these applications and this characteristic sets them apart from scientific applications. However, like applications from scientific and engineering domains, the state sharing needs of such applications are easier to express using the shared memory programming paradigm. However, there is no straightforward way for the programmer to present the temporal requirements in such applications to the spatial memory systems.

A spatial memory model like *release consistency (RC)* [7] corresponds to a programming model wherein synchronization operations are used to protect shared data. Consequently, it guarantees the consistency of shared memory at well defined synchronization points in the program. On the other hand, the interactive nature of time based applications calls for synchronization as a "rate control" mechanism to maintain "loose" synchrony among various parts of the application. Further, the consistency requirements for data are temporal in nature and the applications can tolerate a certain degree of "staleness" with respect to shared state information. Adapting spatial synchronization constructs (e.g. a barrier) necessarily result in a "tight" synchrony among the threads of a computation. Similarly, a spatial memory system enforces strict consistency guarantees commensurate with the memory model. Adhering to strict semantics for synchronization and consistency can fail to exploit opportunities for weaker interactions ("loose" synchrony, "staleness" of data) and can thus adversely affect both performance and scalability of the memory system. Further, the inability to express temporal requirements leads to cumbersome programming practices for interactive applications. Therefore, enhancing the shared memory paradigm to allow the programmer to naturally express the temporal require-

*This work has been funded in part by NSF grants MIPS-9058430 and MIP-9630145 and NSF II grant CDA-9501637.

†currently with Digital Equipment Corporation, Cambridge Research Lab, on sabbatical leave from Georgia Tech

ments and allowing relaxation of synchronization and consistency in an application specified manner has the potential to provide improvements in efficiency and programming "ease." We develop two novel concepts in this context: *delta consistency* and *temporal synchronization*.

We build these mechanisms into a flexible software architecture for a distributed shared memory (DSM) system called *Beehive*. Beehive implements a global address space across a cluster of uniprocessor or SMP nodes. It also supports a configurable access granularity, multithreading, and a flexible memory system. Because workstation clusters have to deal with message latencies that are an order of magnitude larger than those in tightly coupled multiprocessors, there is a need for efficient mechanisms for latency tolerance. Beehive addresses these issues by incorporating mechanisms for aggressive state sharing. These mechanisms provide intelligence to the system on "when" to send data, and to "whom" to send data. Along one dimension, the *temporal mechanisms* give a (time) window of opportunity for coalescing writes by the processor to the same location as well as aggregating consistency actions to the same coherence unit (cache line). Along the other dimension, Beehive provides a set of *explicit communication mechanisms* that allow the producer of data to push it to prospective consumers thus avoiding the read stall overhead at the receiving end. While there have been several other efforts in building DSM systems [3, 11, 12, 16, 21], to the best of our knowledge, all such efforts support the standard spatial notions of synchronization and consistency.

The primary contributions of this paper are:

- temporal mechanisms for consistency and synchronization to enhance the shared memory programming paradigm for time based applications,
- explicit communication mechanisms for latency tolerance,
- design and implementation of a flexible memory system embodying these mechanisms in a workstation cluster,
- performance results showing the utility of this flexible memory system for a virtual environment application.

The rest of the paper is organized as follows: Section 2 discusses the virtual environments application domain which is used as the driver in synthesizing the temporal mechanisms for consistency and synchronization. The mechanisms themselves are presented in Section 3. Section 4 gives an overview of the system architecture of Beehive focusing primarily on the memory subsystem design. Using a virtual environment application, Section 5 presents the experiments and the performance results. In particular, we show the utility of the proposed mechanisms to meet the real time requirements of the application. Section 6 presents concluding remarks.

2 Virtual Environments

We use interactive applications (with "soft real time" requirements) as the driver for synthesizing the temporal programming model. Therefore, we now describe one such domain in general, and the specific application which we use in our experimental study in particular. Applications in the domain of *virtual environments* include interactive virtual world scenarios ranging from video arcade games for entertainment to medical, security, and military training exercises. This domain is rich in applications with both practical uses and mass appeal and will sustain a prolonged interest in parallel computing.

Virtual environments consist of humans and simulated actors interacting in a virtual world. Virtual environments require the ability to incorporate input from human users, move synthetic actors in a compelling and interactive fashion, and produce a real time graphical rendering of the virtual world. In our virtual environment, the user rides a recumbent bicycle to move through the world. A view of the world that reflects the motion of the user's head is displayed on a head-mount display. The world is populated with dynamically simulated, legged robots that attempt to avoid the bicyclist, each other, and obstacles in the world. The user's goal is to herd, like a sheep dog, the legged robots towards a goal. Many interacting actors are required to create an interesting environment.

One of the open problems in virtual environments is generating the motion for virtual actors. The solution we use for this problem is to perform realistic physical simulation of multiple virtual actors in real time [9]. In the particular application we are exploring, each creature has between 4 and 22 degrees of freedom and requires between 0.25 and 0.9 seconds to compute 1 second of motion on an Ultra Sparc. The realistic simulation of multiple actors is the core of the application and the mechanisms we synthesize in this work are inspired by the requirements for supporting the computational requirements of this simulation.

Parallel processing is required because of the high computation cost of simulating multiple creatures. The application is inherently parallel as each actor can be simulated as an independent thread with some sharing of state information. As the number of synthetic actors increases from the current 25 to hundreds, a network of workstations will clearly emerge as the most cost effective and scalable parallel processing platform. Cluster systems are also more amenable to concepts of availability, a critical constraint in some virtual environments.

A shared memory programming paradigm is the simplest programming model for developing a virtual environment application and a shared memory abstraction is an attractive option for the type of state involved in these applications. But applications in this domain have novel synchronization and consistency requirements. A concept of time is inherent in the application: all threads need to have a concept of real time because they simulate actors that must interact with a human user; the graphics display rate of the system should be 30 frames/second and the graphics engine must receive updates of each actor's location and configuration at that update rate. Actors that are near each other but not in contact should be allowed to move together and avoid collisions. Thus, there is a need to maintain "loose" synchrony among synthetic actors while each still adhering to the concept of real time. The application can tolerate significant levels of "staleness" with respect to various actors' perception of each other's state because the time granularity of the simulation is relatively fine compared to the rate at which the state is visible to the outside world (simulation rate of 0.003 s vs. a frame rate of 0.03 s). A creature's behavior is normally expected to be fairly stable and thus requires low update rates for most planning operations (in the range of 0.03 to 0.3 s). However, if two actors were to collide, the simulations would need to share state at the simulation rate in order to transfer force information and maintain physical realism.

Applying standard synchronization constructs and adapting "spatial" memory models to this problem is an artificial solution that would significantly hurt performance and scalability on a cluster of workstations. All opportunities for latency hiding are specified temporally in such applications and spatial memory systems cannot take advantage of them. Spatial memory systems also lead to cumbersome programming practices for interactive applications. For example, *nop* loops are often required in such applications to make the passage of real time to correspond to the application's notion of time. For these reasons, we exploit the application characteristics and develop time dependent mechanisms to deliver performance and meet the real time demands of any interesting virtual world involving many creatures.

The work by Singhal [23] addresses the problem of scaling to large virtual worlds at the application level by reducing the amount of information that needs to be shared and the frequency of communication. Our effort in synthesizing appropriate system level mechanisms for communication and synchronization is complementary to Singhal's work.

3 A Temporal Programming Model

It is our belief that applications that require "temporal guarantees" for correctness need a radically different programming model than the one offered by spatial memory systems. In this section we will develop the mechanisms of temporal synchronization and consistency for supporting such a programming model. The two together form what we will refer to as a *temporal memory system*. In Section 4, we present the API to this temporal memory system.

In the previous section, we presented a virtual environment application to motivate the need for a temporal programming model. In fact, any application that has any notion of "time" inherent in its specification can benefit from the mechanisms we develop in this section. We expect such other domains as process control, interactive vision, web-servers and on-line monitoring to benefit from a temporal programming model.

The temporal programming model is based on any application specific notion of time called *virtual time*. The ingredients of a temporal programming model include: providing an interface for the application thread to advertise its virtual time; maintaining virtual time on behalf of each application thread; computing a *global virtual time* (GVT) as the *minimum* of all the per-thread virtual times; and providing temporal guarantees for synchronization and shared memory operations with respect to the thread virtual times and GVT.

Concepts of virtual time and GVT are not new, and have been used extensively in distributed discrete-event simulation systems [4, 6, 10]. Similarly, concepts of "deadline scheduling" and "dead-reckoning" in real time systems [22, 26] have similarity to the temporal synchronization we are proposing. The primary contribution of our work is in integrating these concepts into a shared memory programming paradigm that allows an intuitive approach to developing applications with temporal requirements.

3.1 Delta Consistency

In spatial shared memory systems, the correctness criterion for memory coherence is the following: "a read of a memory location returns the most recent write to that location". Memory consistency models such as sequential consistency (SC) [14], and release consistency (RC) [7] build on this criterion to provide a global ordering guarantee for shared memory reads and writes.

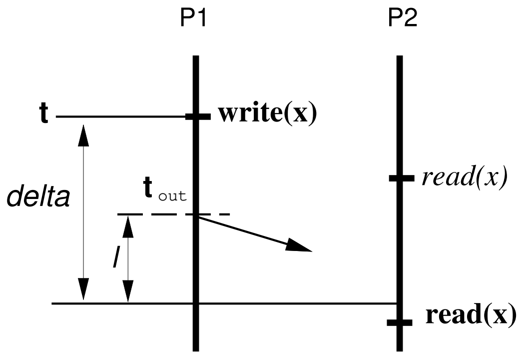

Figure 1: Delta Consistency

For applications needing temporal guarantees, it is more natural to express the correctness criterion for memory coherence, as well as the ordering guarantees temporally. The delta consistency model presented here is motivated by this fact. The consistency model builds on the following correctness criterion for memory coherence: "a read of a memory location returns the last value that was produced at most *delta* time units preceding that read operation". *Delta* is an application-specified "temporal tolerance" given in virtual time.

The delta consistency model specifies the ordering properties for shared memory reads and writes with respect to the above correctness criterion. It is a contract that puts a bound on the "staleness" of values seen by a consumer (i.e. reads to shared memory) to at most *delta* units before the time of the read operation. The temporal synchronization mechanism ensures that this contract is met and is explained in detail in Section 3.2. There are three basic "production rules" for data values (i.e. writes to shared memory):

1. A value produced at time "*t*" by any producer in the system is guaranteed to be visible to all future consumers no later than time "*t+delta*".
2. Successive writes to the same location by the same processor are observed in the same order by all future readers.
3. There is no global ordering for writes to the same location from different processors. Therefore, different consumers can see writes to the same location by different processors within a *delta* window of time in any order.

The ordering properties of delta consistency are reminiscent of processor consistency [8], with the difference that the ordering guarantees are per-location and are with respect to the temporal correctness criterion for memory coherence.

In the proposed temporal programming model, shared memory access primitives use delta consistency as the ordering criterion. Figure 1 illustrates delta consistency. The second read(x) on P2 is guaranteed to return the value of write(x) on P1. The first read(x) can return an older value.

3.2 Temporal Synchronization

The second aspect of a temporal programming model is synchronization. The mechanism we are proposing for temporal synchronization achieves two objectives simultaneously. The first objective is to keep the threads of the computation in "loose synchrony" along the time axis. As we will see shortly, this objective also achieves the delta consistency guarantee for the consumers of data. The second objective is to advertise to the system where a thread is along the virtual time axis so that the system can compute GVT. As will become apparent (see Section 3.3), it is fairly straightforward to link the thread's notion of virtual time to real time (if necessary). This feature is crucial for interactive applications where "the correctness criterion" for the application is tied to the passage of real time and responsiveness to external real time events.

Essentially the idea of temporal synchronization is very simple, and is best explained via the primitives we have developed to support it. The primitive for synchronization is *tg_sync()* (stands for time-group synchronization), and can be viewed as the temporal counterpart to the spatial notion of a barrier synchronization. We also provide a book-keeping primitive *tg_register(time_step)* (time-group register). This primitive achieves two things: (a) it initializes the virtual time of the calling thread to zero; (b) it advertises the time quantum (*time_step*) by which the system should advance the calling thread's virtual time on every *tg_sync* call. Different threads can have different values for the *time_step* parameter.

The semantics of the *tg_sync()* call is the following: (a) advance the virtual time of the calling thread by the *time_step* quantum specified in the *tg_register* call; (b) suspend the thread if its new virtual time is greater than "GVT + *delta*", until GVT catches up to make this inequality false; (c) take the necessary consistency actions to satisfy the production rule (1) in Section 3.1.

The first point advertises the new virtual time of the thread to the system. The *time_step* parameter cannot be negative, which means that the system will not allow the virtual time of a thread to go backwards. The second point ensures that the thread remains in "loose synchrony" along the virtual time axis with the other threads in the system. Here, *delta* is the degree of "looseness" in the synchrony i.e. the maximum permissible skew in the virtual time of threads is bound by *delta* (refer Figure 2). It also enforces the guarantee of delta consistency from the consumer end. At a higher level both "staleness of *delta* in consistency" and the "maximum degree of looseness permissible" translate into the same application level guarantee desired (Section 2). A computation's temporal position is reflected via its state only. The third point allows *tg_sync* to serve as a launch point for the required consistency actions and a check point for others which may need to be completed before the next *tg_sync* call. As should be apparent, temporal synchronization and delta consistency are closely inter-twined. In particular, both give guarantees using the same unit for time specified by the application. The implicit condition for the synchronization and consistency to work correctly is that delta should be greater than or equal to the the largest *time_step* specified by any thread.

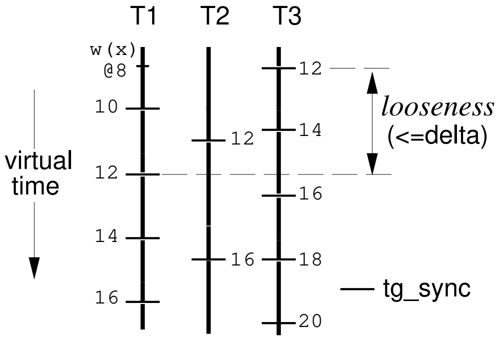

Figure 2: Temporally synchronized threads

Figure 2 shows three threads in loose synchrony via temporal synchronization. Thread T2 has a *time_step* twice that of threads T1 and T3 and the corresponding virtual time is shown with each thread at each *tg_sync* point. Figure 2 also shows the relationship between delta consistency and temporal synchronization. If *delta* is equal to 8, the system will not allow any thread's virtual time to advance to 16 until the write w(x) at 8 is visible to the future consumers.

3.3 Implementation Considerations

In this section we discuss implementation considerations for the temporal programming model on a cluster of workstations supporting software distributed shared memory. The key to the temporal programming model presented in the previous section is the notion of GVT. This term has a fairly well-understood connotation in distributed systems [17] and discrete-event simulation systems [25], namely, the minimum of all the per-thread virtual times. All the temporal guarantees for consistency and synchronization rely on GVT calculation by the system which we show is not very expensive to maintain.

Virtual time is an application-specific attribute. For example, an application thread could use an internal counter for generating successive values for its virtual time. The call *tg_sync* announces monotonically increasing values for the virtual time of a thread to the system. Thus a node in the cluster knows the virtual times of all the threads executing on it. Further, each node can also quite easily discern the virtual times of the threads executing on the other nodes of the cluster. In a "live" system, message exchanges are frequent (for example to access distributed shared memory) and therefore each node of the cluster can calculate the GVT locally based on the messages it has received from the other nodes. A node can send the virtual times of the threads executing on it piggy-backed on appropriate memory system messages or periodically as separate "I am alive" messages.

One of the primary purposes of the temporal programming model is to support interactive applications (such as interactive virtual environments). In this case, it is natural to use "real time" as the time base. That is, the passage of virtual time of a thread corresponds to the passage of real wall-clock time. Since this is a special case, we have an interface call *set_interactive* that allows an application to instruct the system to use real time as the time base. This operation causes the system to synchronize the virtual time in the application with real time at each *tg_sync* point and provide the needed rate-control mechanism.

Although real time is used as the time base, it is certainly not necessary to have a global clock to implement the proposed mechanisms. It is sufficient if all the clocks on different CPUs *tick* at the same rate. Given that there is extremely low (absolute as well as mutual) clock drift with quartz crystals relative to the *time_step* granularity (which we expect to be of the order of several milliseconds for the kinds of interactive applications we are targeting with the temporal programming model), a constant rate of clock advance is a safe assumption. Therefore, the system can simply use the "local" real time clock to implement the temporal guarantees of synchronization and consistency.

Let us discuss the implementation of the three production rules in Section 3.1. To realize the first production rule the system ensures that all consistency information corresponding to a write at virtual time "t" is sent out from the producer by time "*t+delta-l*" (t_{out} in Figure 1), where l is the approximate messaging latency in the system. On the consumer side, a read should be blocked if it can violate the delta consistency contract. As we mentioned in Section 3.2, the temporal synchronization mechanism ensures that all threads remain within time *delta* of each other ensuring that consumers will never read any values older than *delta*.

The second production rule translates to a guarantee requirement from the underlying messaging layer that messages between any two nodes of the system be delivered in order. Such a guarantee is easily achievable. The third production rule basically says that there is no ordering requirement for messages originating from different processors to a set of receivers.

3.4 Comparison of Memory Systems

Consistency: We now contrast our temporal memory system against a memory system based on release consistency (RC) [7]. The first observation is in terms of opportunities for overlap of computation and communication. With reference to Figure 1, t_{out} is the "right" time to push out information about *write(x)*. This provides an optimal time window of opportunity for the memory system to coalesce writes by the processor to the same location as well as aggregate consistency actions to the same coherence unit (cache line). There is a similar opportunity to delay consistency actions in an RC-based memory system as well since the *release* operation acts as a fence to ensure that all such actions are complete before allowing the computation to proceed. However, the memory system cannot make much use of this opportunity since it has no way of knowing, at the point of a *write* for example, the distance in time to the next bounding *release* operation. Thus the memory system is either forced to take conservative decisions regarding launching of consistency actions (which is usually immediately), or wait until the *release* point, thereby forcing the processor to stall for the completion of the consistency actions. In the first case, the RC-based memory system has less opportunity to coalesce writes and results in more messages than the temporal memory system, and in the second case it fails to significantly overlap computation and communication.

The second observation relates to protocol optimization. Assuming a reliable messaging layer, all the required guarantees for delta consistency can be provided without any acknowledgments for consistency actions. This leads to a significant reduction in protocol overhead. For reasons similar to those discussed in the context of the first observation, such optimizations are just not viable for spatial memory systems. Therefore a temporal memory system can achieve a significant reduction in message traffic while still making optimal decisions regarding overlapping computation with communication. These overlaps improve the performance and increase the scalability of temporal memory systems compared to an RC-based memory system for such interactive applications.

There is another way to look at the temporal programming model. An RC model ensures that a read by a processor returns *at least* the value written into that location by some processor in the immediately preceding synchronization epoch in the execution history. If we use synchronization points to mark the passage of virtual time, delta consistency generalizes the RC notion to say that the read can return a value that can be *at most delta* epochs old.

Synchronization: Temporal synchronization allows an application to get the appropriate level of synchrony among the participating threads by a proper choice of the time base. With spatial synchronization, the user always gets "tight" synchrony, and there is no way for the user to specify loose synchrony. Thus temporal synchronization is expected to incur fewer messages than a system providing "tight" synchrony. Moreover, such a loose synchrony reduces the chances of a thread needing to block at synchronization operations, thereby achieving better overlap of computation and communication. Contrast this to a (spatial) barrier synchronization operation, wherein all threads must block until the slowest thread arrives at the barrier. Similar to spatial synchronization, temporal synchronization also serves as a launch point for consistency actions mandated by the memory model. Finally, for interactive applications, temporal synchronization can provide feedback for tuning the application to meet the real time deadlines. For example, suppose the *time_step* specified for *tg_sync* is less than the passage of "real time" since the previous *tg_sync* call. This indicates that the required computation exceeds the *time_step* specified by the thread (i.e. it missed a "deadline"). The temporal programming model allows for application-specified action to be taken when deadlines are missed.

Scheduling: Inherent to the temporal programming model is the requirement that the passage of virtual time for all the threads of the same application be approximately uniform. This requirement is crucial for interactive applications suggesting that "coscheduling" threads of the same application in a cluster will be very useful. Currently, in our Beehive testbed (described in Section 4) we dedicate the cluster to a single interactive application at a time. Global scheduling such as is done in Berkeley NOW [2], is a reasonable way of processor allocation for handling a mix of applications.

3.5 Implications for Parallel Programming

The temporal memory system provides the same set of operations as a spatial memory system, namely, reads and writes on a global address space. In addition it provides the temporal synchronization operation. As we mentioned earlier, the *tg_sync* operation would be used in exactly the same context as a spatial barrier synchronization. Therefore in the sense of programming a temporal memory system, the application programmer does not have to do anything differently from a spatial memory system. The significant programming advantage with the temporal memory system is that it eliminates the need for cumbersome "hacks" such as *nop* loops to mark passage of real time. The only additional requirement is for the programmer to specify the "delta" and "time step" parameters for the consistency and synchronization mechanisms, respectively. In the interactive applications for which this memory system is proposed these parameters are readily available.

Figure 3 shows pseudo code for the actors in the virtual environment application (Section 2) using the temporal memory system. If the simulation is not interactive (for example: display is an off-line process), the concept of time in the application still remains the same. The application will simply not use the *set_interactive* call.

RC has become an accepted model for shared memory systems because it corresponds exactly to a programming model where synchronization operations are used to protect shared data. The intent in proposing the temporal memory system is to bridge a gap that exists between interactive applications actor

```
#define delta ??

time_step = 5.4 ms
set_interactive
tg_register(time_step)

for each simulation step {
  simulate actor
  write(actor_state)
  ...
  tg_sync()
}
```

Figure 3: Pseudo code for simulated actors

and the standard parallel programming paradigm.

Temporal and spatial mechanisms can coexist in the same memory system. In fact, the testbed Beehive (to be described in Section 4) provides for both types of mechanisms. However, it is not clear if there will be significant benefits to using both in the same application. For the virtual environment application, we have strictly used only the temporal mechanisms. It should also be possible to change *delta* dynamically within the application, a characteristic which can be entirely determined by the scope of the implementation.

4 Beehive Cluster System

In this section, we summarize the system architecture of *Beehive*, a cluster system for supporting interactive applications and compute-intensive servers. More details of the system architecture and implementation can be found in a companion technical report [24]. The system provides a shared memory programming environment on a cluster of workstations interconnected by a high speed interconnect. The current implementation of Beehive uses Sun UltraSparc CPU boxes (Uniprocessors or SMPs) running Solaris operating system, interconnected by two different networks – 100 Mbit/Sec Fast Ethernet (using a UDP based message layer), or Myricom's Myrinet switch (using a Fast Messages Release 2.0 [18] based message layer). The message layer is only used by the Beehive system itself to implement the control, synchronization, memory allocation, and communication capabilities of the Beehive API.

The principal design features of Beehive include: a global address space with address equivalence across the cluster, a configurable access granularity, multithreading, and flexibility in supporting spatial and temporal notions of synchronization and consistency. The mechanisms for shared memory parallel programming are made available to the application programmer via library calls (see Table 1 for the Beehive API). Beehive uses a single consistent and powerful API to exploit both the multiple CPU resource of a SMP platform and resources on multiple nodes in the cluster. The underlying implementation is optimally efficient for each case.

Beehive provides flexibility in the memory system design along two orthogonal dimensions. Along one dimension we exploit knowledge of data sharing patterns in the applications to decide *whom* to send the data to (i.e the "future sharer set"). This aspect builds on our earlier work [19] wherein we proposed a set of *explicit communication* mechanisms within the confines of coherent shared memory. Similar ideas can be found in [1, 13, 15, 20]. Along the other dimension, we exploit the opportunities that may exist in the applications for deciding *when* to send the data to the sharers. This aspect is achieved via supporting different consistency levels

Control	
bh_fork	fork a beehive thread to local/remote node
bh_join	join with a beehive thread
Global Memory	
bh_sh_malloc	allocate a piece of global memory
Communication	
sh_read	shared memory read
sh_write	shared memory write (invalidation based)
pset_write	explicit shared memory communication
sync_write	"
selective_write	"
Synchronization	
bh_tg_sync	temporal time group synchronization
bh_barrier	barrier
bh_lock	mutex lock
bh_unlock	mutex unlock

Table 1: Beehive API

in the memory system – sequential consistency (SC), release consistency (RC), and delta consistency (Section 3.1) – and using the appropriate one based on application requirements.

The memory system supports access to shared memory via two calls: *sh_read* and *sh_write*, which are the simple reads and writes to shared memory. The base cache coherence protocol is a directory-based write-invalidate protocol. The simple read/write primitives are augmented with a set of three explicit communication primitives that push the data out upon a write to "future" readers so that the read-miss latency is avoided at the consumer end. The difference among the three primitives is simply the heuristic used to determine the "future" consumers. In *pset_write*, the set of consumers is statically determinable at compile time and is specified as a *processor mask* to the primitive. In *sync_write*, the next (if any) acquirer of a mutex lock is available from the lock structure. This information is used directly to push the data written during the critical section governed by this mutex lock to the prospective consumer node. The third primitive, *selective_write*, is the most general for use in situations where the communication pattern cannot be discerned *a priori*. The memory directory maintains a *current sharer set* for each data block. A *selective_write* freezes the sharer set and the use of a different write primitive destroys it. Now upon a *selective_write*, data is pushed to the current sharers by basically switching the coherence protocol to a write-update from a write-invalidate. However, if the sharer set changes again (i.e. new readers join the set), then the current sharer set is invalidated and the system reverts to the base write-invalidate protocol until the next *selective_write*. This set up is flexible enough to implement an alternate heuristic for identifying a sharer set.

Beehive implements the temporal synchronization mechanism as presented in Section 3.2. The memory system also provides the spatial synchronization primitives – mutex locks and barriers. The *sync_write* primitive uses this lock structure to obtain the identity of the next node (if any) waiting for the lock.

5 Performance Evaluation

The Beehive system architecture offers a rich environment for the experimental evaluation of different system options in cluster parallel computing. In this paper we focus on the experimental results pertaining to the temporal mechanisms and the explicit communication primitives. The main point we want to convey through the experiments is the power of these Beehive mechanisms in meeting the soft real time requirements of interactive applications. The experiments are conducted on a cluster of 8 uniprocessor UltraSparc nodes interconnected by Fast Ethernet and Myrinet. The latencies for a one-way null message for the two corresponding messaging layers (Section 4) are 400 microseconds and 30 microseconds, respectively. Thus this study is also useful in understanding the impact of messaging cost on various system design choices.

We use a virtual environment application that consists exclusively of simulated one-legged robots for our experiments. The one-legged robot has four controlled degrees of freedom: three degrees of freedom at the hip and one for the length of the leg. During each step of the computation, the equations of motion for the dynamic simulation are integrated forward in time to determine the new position and velocity for each robot. The control torques at the joints are computed to maintain balance, hopping height and forward speed. When the robot is part of a group, a new desired position in the group is calculated to avoid collisions.

Each step represents a passage of 5.4 milliseconds of real time in the life of a robot. Each robot updates its own state each time step. Robots are organized into groups called flocks. On an average, a robot reads the state information of all other robots in its own flock 30% of the time, and reads the state information of all the robots in the simulation about 2% of the time. The state information for a single robot is approximately 256 bytes which is also the cache line size specified to Beehive for this application.

The standard notion of speedup with increasing number of processors for a constant problem size is not a meaningful metric for such applications. The application is interactive and the challenge is to be able to simulate more interesting virtual worlds as more computational capacity becomes available while still being able to meet the real time constraints. Therefore in all the experiments we linearly scale the problem size with increasing number of processors. The base experiment is a 10 second real time simulation of this virtual environment application with three robots per node. We have one "Beehive-Application" process per cluster node with three application threads – one per robot. We increase

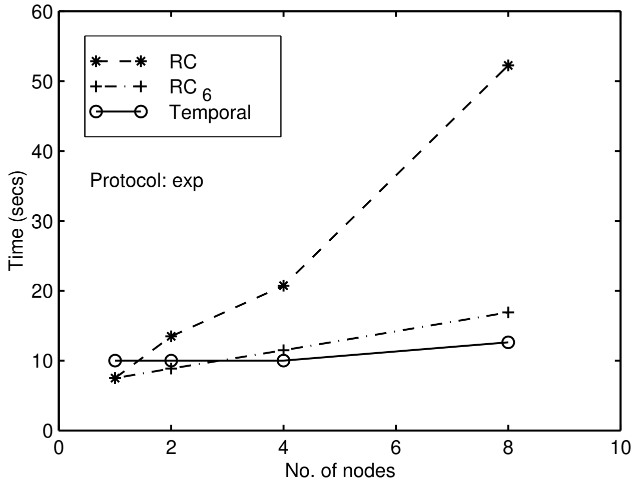

Figure 4: Memory Systems Comparison on UDP

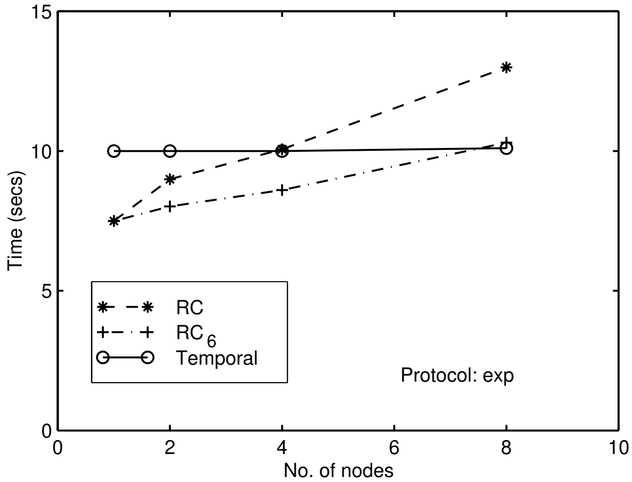

Figure 5: Memory Systems Comparison on FM

the number of robots participating in the simulation proportionate to the number of nodes. The robots are divided into two flocks. The aim is to be able to perform this 10 second simulation within 10 seconds even as the number of nodes (and robots) in the system is increased.

We performed two sets of experiments. The first set is to show the power of the temporal memory model to meet the real time requirements for this application. The next set of experiments is to show the benefit of the explicit communication mechanisms in context of the different memory models.

5.1 Memory Systems Comparison

We are comparing memory systems that use two different consistency models: *temporal* and *RC*. For RC we are considering two different implementations of the application. To keep the discussion simple we will refer to the two implementations on RC as two different memory systems.

Temporal is the memory system that uses delta consistency in concert with temporal synchronization developed in Section 3. In the coding of the application for this memory system, each robot does a *tg_sync* at each simulation step. The corresponding *time step* is 5.4 milliseconds as stated above and the mode is *interactive*. The *delta* for this application was chosen to be 35 milliseconds in consultation with the application developers.

RC is the memory system that uses the release consistency memory model in concert with spatial synchronization mechanisms. In the coding of the application for this memory system, each robot does a barrier synchronization with all the other robots for every step of the simulation.

RC_6 is the same memory system as *RC*. The only difference here is that instead of using barriers for synchronization on every step of the computation as in *RC*, the robots synchronize every 6 steps. This approach roughly corresponds to the 35 millisecond slackness (*delta* value for *Temporal*) that is available for updating the state information in the application.

For the memory systems being compared, we use the invalidation-based coherence protocol augmented with the explicit communication mechanisms described in Section 4. For this application only the *selective_write* primitive was used, and we essentially changed all shared writes to *selec-*

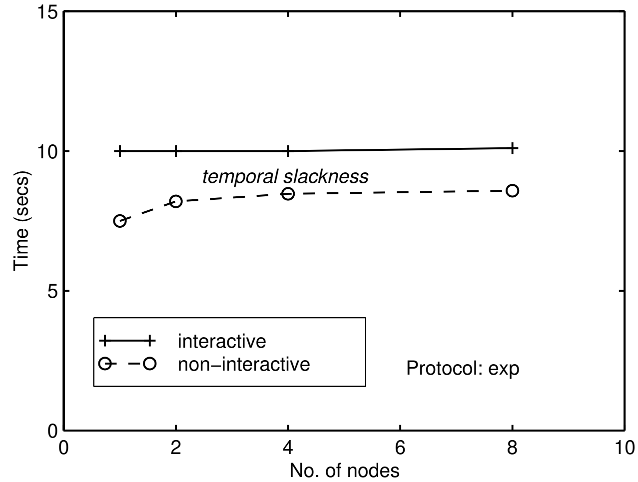

Figure 6: Temporal Slackness on FM

tive_writes. We refer to this protocol as *exp* in the rest of the section and the graphs, and to the base invalidation-based protocol as *inv*. All three memory systems use *merge buffers* [5] to coalesce outgoing write traffic.

Figures 4 and 5 compare the performance of the three memory systems using the *exp* protocol for UDP and FM respectively. The x-axis is the number of nodes in the cluster, and the y-axis is the execution time taken by the application. Since the application executes for 10 seconds of real time, we want a flat line at 10 seconds as the number of nodes (and robots) is increased to meet the correctness criterion for the interactive application. We observe that *Temporal* outperforms both *RC* and RC_6 by closely following a flat line at 10 seconds. *RC* fails to meet the real time constraint even for two nodes (6 robots) on UDP and beyond 4 nodes (12 robots) on FM despite the fact that messages are much cheaper on FM. The trend of the RC_6 curves shows that it scales much better than *RC*. The number of messages generated in RC_6 is quite close to that in *Temporal*, but it still fails to meet the 10 second constraint beyond 4 nodes (12 robots). This is because of stall times at synchronization points incurred in RC_6 to flush the write and merge buffers. On the other hand, *Temporal* uses the temporal information at its disposal to advantage, maximally overlapping computation with communication without incurring synchronization stalls. In all experiments each *tg_sync* deadline is met in real time.

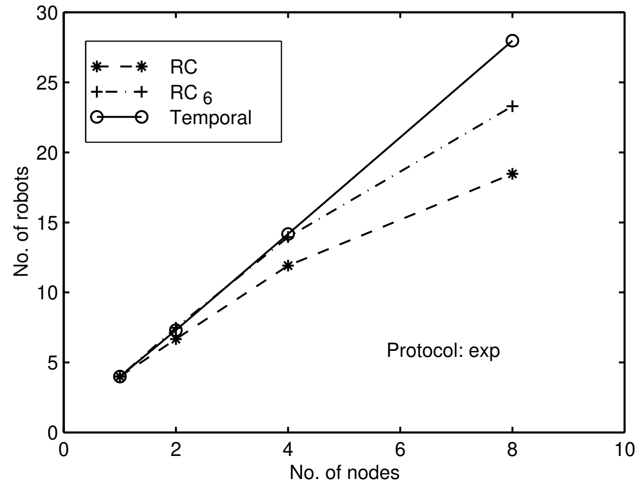

Figure 7: Possible "problem size" Scaling

Though the virtual environment application we use in this study is an interactive application, using an *interactive* mode (through the *set_interactive* call to the memory system) for it does not fully bring out the potential of the temporal memory system as visible in Figures 4 and 5. In certain instances, the system could be causing the application threads to remain idle in order to remain in synchrony with real time. These wasted cycles constitute a *temporal slackness* which can be gainfully assigned for other useful work on the node. Figure 6 shows the *temporal slackness* in the system on running this application on FM using the *exp* protocol obtained by comparing the performance for the *interactive* and *non-interactive* modes. It also highlights the superior scalability of the temporal memory system as shown by the performance for the *non-interactive* mode. There is no corresponding concept for *RC* and RC_6. Therefore it is interesting to compare the number of robots that can be simulated in real time with increasing number of nodes in the cluster using *RC*, RC_6 and *Temporal* with zero *temporal slackness*. Figure 7 shows the result of this experiment. It can be seen that the behavior of *Temporal* is almost linear in this range, while the other two memory systems show sub-linear scalability.

Apart from the performance advantage of the temporal memory system, it is easier to program for interactive applications. To program this application using the non-temporal memory model (either *RC* or RC_6), would require the programmer to explicitly add busy-wait in the threads to ensure that a *time step* had elapsed before proceeding to the next step of the computation. For example, this happens with 1, 2, and 4 nodes for RC_6 on FM (see Figure 5).

5.2 Effectiveness of Explicit Communication

The second set of graphs in Figures 8 and 9 illustrate the effectiveness of the explicit communication mechanism for the different memory systems. The first figure is for UDP and the second is for FM. Both figures show that the invalidation-based protocol (*inv*) incurs a significant read miss penalty. Increasing the cluster size implies increased data sharing since the problem size increases with the size of the cluster. Thus *inv* scales poorly with increasing cluster size. On the other hand, *exp* aggressively pushes out data to future consumers and thereby hides the read miss latency at the cost of increased number of messages. On UDP, where the message latencies are higher, it can be seen that *inv* performs better than *exp* for *RC* (Figure 8). But RC_6 and *Temporal* generally reduce the communication traffic at the memory system level and the messaging layer is not saturated despite the increased number of messages for *exp* relative to *inv*. Thus *exp* can still outperform *inv* for these memory systems. On FM where message latencies are much lower, *exp* is significantly better than *inv* even on *RC*. The performance of *inv* is always better for the small cluster size of two nodes; extremely limited degree of data sharing renders the use of any "aggressive" protocol worthless. These experiments show that explicit communication mechanisms are good for hiding read miss latencies in contemporary cluster interconnects.

The good performance of the *exp* protocol suggests that the *selective_write* primitive does a reasonable job of tracking the sharing pattern for shared data. In this application, which is a good representative of similar applications from this domain, active sharing is primarily restricted to a flock, and

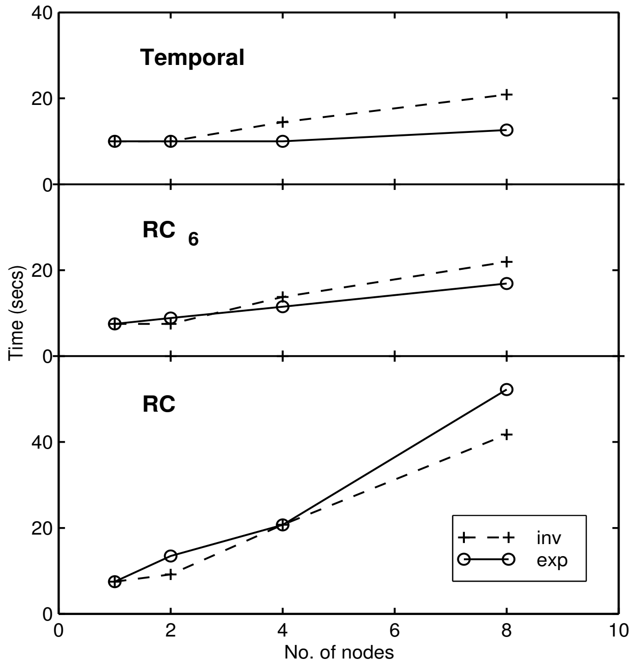

Figure 8: Protocol Comparison on UDP

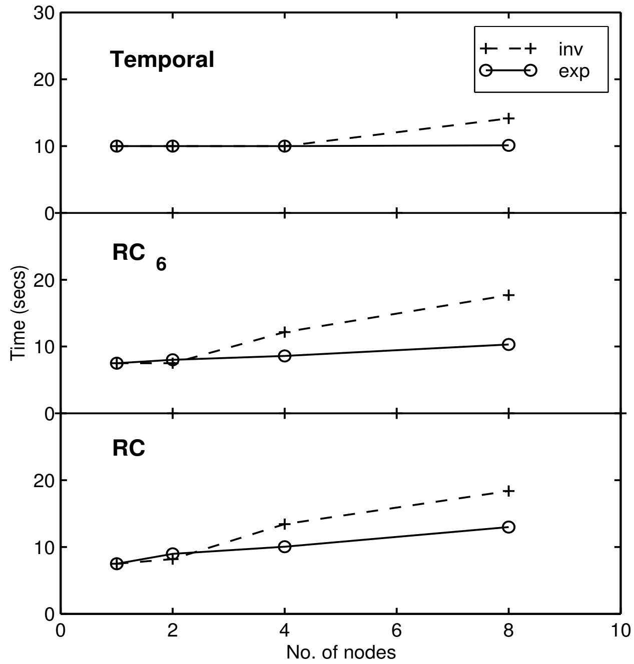

Figure 9: Protocol Comparison on FM

thus *exp* scales quite well with creatures dynamically entering or leaving a flock. *Exp* has the potential to scale well even with increasing problem size if increasing the problem size entails adding additional flocks while maintaining a fixed flock size. A simple update protocol would not perform well for this kind of application. Also it is not clear if prefetch can be of any help in this application. Due to the dynamic nature of this application it is unclear *what* to prefetch and *when* to prefetch and still hope that the prefetched items will be useful when the computation needs them.

6 Conclusions

The shared memory paradigm is convenient and intuitive for developing parallel applications. The synchronization and communication abstractions available for shared memory parallel programming have been quite adequate for scientific and engineering kinds of applications. However, we have seen that these spatial notions of synchronization and consistency are inadequate to express an important attribute of interactive applications – time. Apart from making programming cumbersome for interactive applications, these spatial notions miss out on the opportunities available in such applications for relaxing the consistency requirements. Moreover, spatial synchronization results in a tight synchrony among participating threads that is neither necessary nor desirable in such applications. We have developed temporal notions of consistency and synchronization that provide the right system abstraction for interactive applications to express "time". In addition, by choosing the right frame of reference for the temporal primitives it is possible for the applications to achieve the desired level of synchronization coupling among the threads.

With the emergence of faster networks, cluster parallel computing is becoming a viable option for certain classes of compute-intensive applications. Since the latencies for internode communication in a cluster are several orders of magnitude more than what is typical in tightly coupled multiprocessors, it is necessary to find appropriate mechanisms for latency hiding. An *applications driven systems support for cluster computing* approach, such as ours, can be useful in coming up with the right mechanisms within the scope of the shared memory paradigm. The temporal mechanisms help in latency hiding along one dimension. Along another dimension, we have developed a set of explicit communication mechanisms for aggressively pushing the data out to the future consumers, that helps hide the read miss latencies at the receiving end. We have implemented these mechanisms in a software distributed shared memory architecture called Beehive for a cluster of workstations. The Beehive software architecture is flexible and allows experimentation of various system options for cluster parallel programming.

Using a virtual environment application as the driver we show the efficacy of the temporal and explicit communication mechanisms, working in concert, in meeting the real time requirements of such applications. We also show the inadequacy of the spatial notions for such applications despite using explicit communication mechanisms. While the specific applications of the temporal mechanisms in this paper used real time as the frame of reference, the mechanisms are general and can use any application specified frame of reference. Given this, an open question is whether these mechanisms can be adapted to other situations to relax the synchronization and consistency requirements in an application specific manner. It would also be interesting to study the impact of messaging systems with real time guarantees on the scope and implementation of the temporal model in shared memory programming.

Acknowledgments

Our special thanks to David Brogan who supplied the virtual environment application code for our experiments. Ron Metoyer and Mark Huang participated in discussions which laid the foundations for the work presented in this paper. Discussions with Bert Halstead and Rishiyur Nikhil helped clarify the concepts presented in this paper. Leonidas Kontothanassis's careful reading of an earlier version of this paper, and his thoughtful comments immensely improved the presentation in this paper.

References

[1] H. Abdel-Shafi, J. C. Hall, S. V. Adve, and V. S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Third International Symposium on High Performance Computer Architecture*, February 1997.

[2] R. H. Arpaci, A. M. Vahdat, T. Anderson, and D. Patterson. Combining Parallel and Sequential Workloads on a Network of Workstations.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *The 13th ACM Symposium on Operating Systems Principles*, October 1991.

[4] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[5] F. Dahlgren and P. Stenstrom. Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared Memory Multiprocessors. Technical report, Department of Computer Engineering, Lund University, April 1993.

[6] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] J. R. Goodman. Cache consistency and sequential consistency. Technical Report Technical Report #1006, University of Wisconsin, Madison, February 1991.

[9] J. Hodgins, W. Wooten, D. Brogan, and J. O'Brien. Animating Human Athletics. In *ACM SIGGRAPH 95 Computer Graphics Proceedings*, pages 71–78, August 1995.

[10] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[11] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[12] L. Kontothanassis, G. Hunt, et al. VM-based Shared Memory on Low Latency Remote Memory Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[13] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared memory multiprocessors. In *Ninth ACM International Conference on Supercomputing*, pages 255–264, July 1995.

[14] L. Lamport. How to make a Multiprocessor Computer that Correctly executes Multiprocess Programs. *IEEE Transactions on Computer Systems*, C-28(9), 1979.

[15] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *Transactions on parallel and distributed systems*, 4(1):41–61, January 1993.

[16] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM TOCS*, 7(4):321–359, November 1989.

[17] F. Mattern, M. Cosnard, Y. Robert, P. Quinton, and M. Raynal. Virtual Time and Global States of Distributed Systems. In *International Workshop on Parallel and Distributed Algorithms*, October 1988.

[18] S. Pakin, M. Lauria, et al. Fast Messages (FM) 2.0 User Documentation.

[19] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors. In *Proceedings of Supercomputing '95*, December 1995.

[20] E. Rosti et al. The KSR1: Experimentation and modeling of poststore. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, 1993.

[21] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[22] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, August 1992.

[23] S. K. Singhal. *Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments*. PhD thesis, Department of Computer Science, Stanford University, August 1996.

[24] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of Synchronization and Consistency in Beehive. Technical Report GIT-CC-96-27, College of Computing, Georgia Institute of Technology, October 1996.

[25] Z. Xiao, F. Gomes, B. Unger, and J. Cleary. A fast asynchronous GVT algorithm for shared memory multiprocessor architectures. In *Ninth Workshop on Parallel and Distributed Simulation*, 1995.

[26] R. Yavatkar. MCP: a protocol for coordination and temporal synchronization in multimedia collaborative applications. In *12th International Conference on Distributed Computing Systems*, June 1992.