Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

SHARING EVENT DATA IN OPTIMISTICALLY SCHEDULED MULTICAST APPLICATIONS

Garrett Yaun
David Bauer
Christopher D. Carothers

Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, U.S.A.

ABSTRACT

A major consideration when designing high performance simulation models is state size. Keeping the model state sizes small enhances performance by using less memory, thereby increasing cache utilization and reducing model execution time. The only remaining area for reducing model size is within the events they create. The event population is typically the most memory intensive region within a simulation especially in the case of multi/broadcast like applications which tend to schedule many events within the atomic processing of a single event. This paper introduces the idea of *shared event data* within an optimistic simulation system. Here, the read-only data section is shared for a multicast event, which may then be delivered to several LPs. From our performance study, we report a 22% reduction in the data cache miss rate, a processor utilization in excess of 80% and a reduction in model memory consumption by a factor of 20.

1 INTRODUCTION

In parallel discrete-event simulations, a model is decomposed into two key data structures: *events* and *logical processes (LPs)*. LPs communicate by exchanging timestamped events messages. These events are then processed in timestamp order. Beyond the parallel synchronization problem, one of challenging issues for large-scale parallel simulation is keeping the model size small. A good way to address this problem is to reduce the amount of duplicate information. As mentioned in (Yaun et al. 2003) Logical Processes (LPs) that share common data can have a global information pointer to the shared data which reduces the total state of the model. From this, the question of *"why must this just be limited to the LPs?"* arises. Could a sharing approach be employed in event data as well? Our experimentation shows that the answer is yes and one good example is a multicast network model (Deering 1989) because of the duplicate nature of the events. However, this approach could be used in several, more generic model scenarios. In fact, at any point in a model where an event is to be broadcast to two or more LPs there can be a significant memory savings attributed to this approach.

In the multicast protocol, data transmission is minimized by sending messages through a multicast tree before being broadcast to each subscriber. The goal is to minimize individual transmissions sent separately to each subscriber. This protocol model has duplicate information being sent to each subscriber where there are branches in the multicast tree. In a simulation with shared memory we have the ability to have a global view of the system. Typically, a multicast model generates messages that result in multiple, identical messages being sent to each subscriber LP in the system. Each LP would then read those messages, update it's state, and possibly generate more events in the system.

Rather than identical events being sent to each subscriber with the same data attached, in our system we keep a pointer to the data in the event header and send each subscriber LP this pointer. Each LP is required not to overwrite the data, as it is understood in the model that this event data is being shared globally throughout the system. Our second requirement is that only once each subscriber has received the multicast event is the attached data reclaimed.

This optimization is important because it drastically reduces the most memory exhaustive component of simulation: the event population. This optimization can be applied to all types of simulation: sequential, parallel and even distributed. In this paper, we will discuss the sequential and parallel implementation of this approach. Additionally, there are two types of parallel synchronizations. With sequential and conservative synchronization the implementation of this idea is trivial because attached data can immediately be reclaimed.

Optimistic simulators pose a greater design and implementation challenge because processed events are maintained for possible future rollback scenarios. These scenarios are much more difficult to address because events must have been read by all receivers prior to reclamation. In fact, there

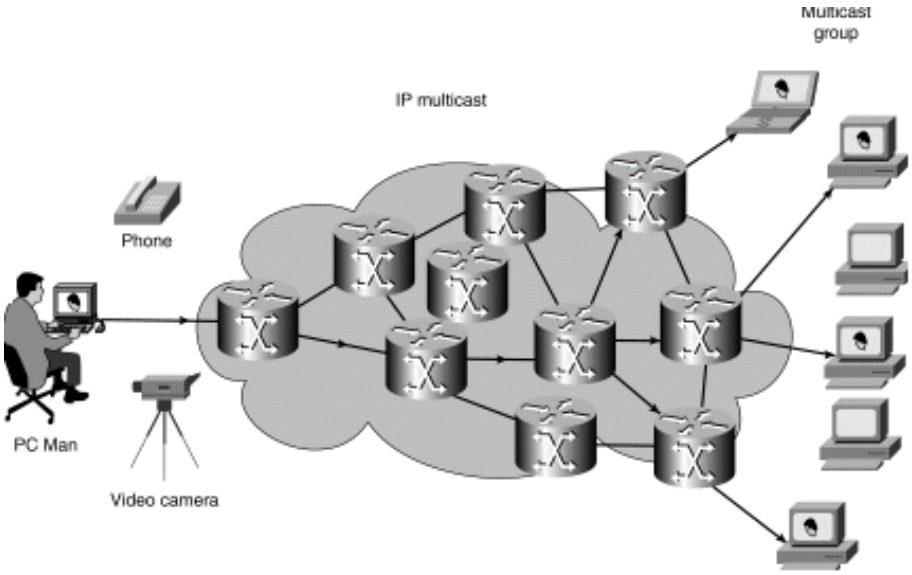

Figure 1: Multicast graphic from Cisco (Cisco 2002)

are several cases that we outline in this paper where this optimization must be managed properly by the optimistic simulation executive.

We chose the multicast protocol model as our primary example for this experimental investigation. We follow with possible implementations of the method assuming different synchronization mechanisms. There will be a detailed discussion of the implementation in a discrete-event simulation which employs optimistic synchronization. For the performance results a benchmark multicast-like model will be used in the evaluation.

2 MULTICAST BACKGROUND

The multicast protocol is a bandwidth-conserving technology that aims to reduce packets in a network by transmitting a single stream of data to thousands of receivers on the network. Many applications take advantage of this protocol, including: video conferencing, corporate communications, distance learning, and distribution of just about any type of information.

Multicast has been in use on large scale networks since the introduction of the Mbone (Macedonia and Brutzman 1994) in 1992. Today, Microsoft China has one of the largest multicast networks, and plans to begin providing multicast television to viewers in 2005 (IPMSI 2004). The multicast solution will become widely used on the Internet as a solution to higher bandwidth costs due to ever increasing numbers of Internet subscribers.

The multicast model works by delivering traffic from a single source to multiple receivers through the multicast tree. Multicast receivers subscribe to a given source, and the information is then disseminated through the multicast tree back to the multicast group of receivers. Internet routers replicate packets at branches in the multicast tree so that all subscribers will receive the same packet data. This low-cost solution not only reduces the amount of bandwidth required to transmit a large stream of data to multiple receivers, it also reduces the number of requests serviced by the source. The multicast protocol is efficient in it's design compared to other protocols which commonly require the source to send individual copies of the same information to multiple receivers. When the amount of data being transmitted is large, it quickly becomes difficult for the source to send multiple copies across a network, as in the case of MPEG video. A large amount of network resources are consumed providing an individual stream for each receiver. The multicast protocol can also provide a substantial savings when the data transmitted from the source is small because there may be thousands of receivers to be serviced. Figure 1 demonstrates how data from a single source is delivered to multiple recipients through a multicast tree.

Our model of the multicast protocol clearly illustrates the performance impact of our shared event data optimization. It is easy to see that if we are able to move X bytes into a shared message and there are 1000 multicast subscribers the savings will be roughly 1000-fold bytes. Conversely, if X is very large, and there are only few receivers, the memory savings will also yield a significant performance improvement as the large event memory segment would only need to be written into memory once. Furthermore,

as these large packets traverse the network the number of copy operations is zero.

3 IMPLEMENTATION

In order to successfully deploy this new idea in a current simulator executive, two restrictions must be met. Once the shared event data has been identified, it can be allocated and written only once, and then dereferenced for each subscriber event created. The first restriction is that each subscriber may not destroy or modify the shared event data. The second restriction is that the simulation executive may not prematurely reclaim the shared event data. *Only once each subscriber has received the event can the shared event data be freed.*

3.1 Sequential and Conservative Simulation

In the sequential simulation this optimization can easily be implemented entirely within the model. The modeler can keep a pointer to the shared event data. For each newly created event that will forward the shared data, the pointer to the shared data is set. Next, there needs to be a way to free the shared data once the event has been processed by all subscribers. If the final subscriber is known, the shared event data can be freed once that subscriber has processed the event. Another approach is to maintain a counter that indicates the number of sends and receives. When the last subscriber receives the event and does not forward it to any other subscriber, the counter will be zero and the shared data segment is safe to reclaim. Since the execution is sequential, only one LP will be accessing the counter at a time and there is no need for mutual exclusion. Finally, the last subscriber to receive the event will be the correct one to free it.

With conservative simulation systems such as DaSSF (Nicol a d Liu 2002), this optimization can also be implemented by the model. Once again the modeler creates a pointer in the event message for the shared data segment. Each newly created event that is going to be forwarding the data can point to the same shared data segment. The freeing of the data can be done the same way as in the sequential case because in conservative simulation there are no roll backs and all events executed are processed in the correct causal order. However, the execution is parallel and so would require mutual exclusion for the counter if that is the method used to determine the second restriction.

3.2 Optimistic Simulation

Within optimistic simulation there are additional issues to address. In particular, speculative execution complicates the allocation/deallocation process when rollbacks occur. We chose to implement this novel idea within ROSS (Carothers, Bauer, and Pearce 2002) because it provides a reversible memory library similar to the structures described in GTW (Carothers, Perumalla, and Fujimoto 1999) and ROSS and it is these memory buffers which form the shared event data segments. ROSS handles causal errors through reverse computation. When a rollback occurs, an event's reverse computation event handler is called, which has the inverse effects on the LP's state compared to the forward execution of the event. ROSS includes a memory library which allows for the dynamic allocation of statically allocated memory. This library was designed to overcome the problem of reverse computing memory operations such as `malloc` and `free` during event execution. The memory library greatly reduces the complexity of many models by allowing them to create memory buffers and either maintain them in their LP state or to send them as part of the event data. When we discuss reclaiming the shared event data segments, it is these memory buffers to which we are referring.

Our implementation used a counter within the event header to track subscriber sends. The easiest implementation of this idea is not to try to reclaim the shared data when it reaches the end points. The reason is the supposed final end point might not be the final end point due to the fact that other end points might be rolled back. In an optimistic solution, the event data segments are reclaimed only once the possibility of a rollback is eliminated by the passing of the global virtual time (GVT) (Jefferson 1985). Only those events with a timestamp less than the current GVT value may be reclaimed by the system (Fujimoto 2000). Typically, for caching purposes, those events are made readily available for the next event creation. This improves the cache hit rates because we know that the newly reclaimed event is in our cache, and so it should be the next event to be allocated. On the reallocation of the event the shared event data can be reclaimed. This method requires additional memory because the shared event data is being reclaimed later in the simulation, but still dramatically less than the amount of memory needed for a non-shared data approach. Within the shared data there is a counter and a mutual exclusion lock which the simulation executive manages. This optimization is entirely transparent to the model.

A second issue is that the execution of an event might not create the desired new event. For example in ROSS, once all event-memory is allocated, a special event called the *abort* event is returned as opposed to returning a null pointer. This enables regular optimistic processing to continue until the scheduler reaches a point at which it can correctly and safely re-claim memory. This approach is similar to the approach taken in Georgia Tech Time Warp (GTW) (Fujimoto and Hybinette 1997) as well as ROSS. In the "event-send" routine, if it finds an abort event has been scheduled, it continues processing but does not send that event. Additionally, when the current event execution

completes, it is rolled back and any events which it created are cancelled.

The steps for the *memory set* routine are illustrated in Figure 2. Here, the newly allocated memory buffer denoted by `b`, has its access control counter increased by one provided the owning event, `e`, is not the abort buffer. If the abort buffer is encountered, and the counter is zero, then that shared memory segment is freed. Otherwise, this routine returns. Next, Figure 3 shows the algorithm for how a shared event segment is deallocated or freed. In this routine, the memory segment's access counter must be zero prior to the actual release of the memory segment. Please note, critical sections are denoted by the `lock` and `unlock` routines. Both increment and decrement operations of the access counter variables are "locked". Additionally, any tests for zero are placed within the lock since one and only one processor should free a shared memory buffer.

```
if(e has been ABORTED) {
    if( *b )
    {
        lock(&((*b)->mem_lck));
        if((*b)->counter != 0)
        {
            unlock(&((*b)->mem_lck));
            return;
        }

//free the memory pointed by event e
        free(e->memory);
        unlock(&((*b)->mem_lck));
        *b = NULL;
    }
    return;
}

lock(&((*b)->mem_lck));
(*b)->counter++; unlock(&((*b)->mem_lck));

e->memory = *b; return;
```

Figure 2: Memory Set Routine. This routine is preformed when setting a memory buffer to an event. B is the memory buffer. E is the event.

We observe that this interface only affects forward event processing. When a rollback occurs, the reverse event handling code is not effected and no new function calls or code modifications are required to support shared event segments.

4 PERFORMANCE STUDY

The Itanium-II processor (Intel 2002) is a 64 bit architecture based on *Explicitly Parallel Computing (EPIC)* which intelligently bundles instructions together that are free of data, branch or control hazards. This approach enables up to 48 instructions to be in flight at any point in time. Current implementations employ a 6-wide, 8-stage deep pipeline. A single system can physically address up to 2^{50} bytes and has a full 64-bit virtual address capability. The L-3 cache comes in a 3 MBs configuration and can be accessed at 48 GBs/second which is the core bus speed.

```
if(e->memory) {
    lock(&e->memory->mem_lck);
    e->memory->counter--;

    if(e->memory->counter == 0 )
    {
        unlock(&e->memory->mem_lck);
//free the memory pointed by event e
        free(e->memory);
    }
    else
        unlock(&e->memory->mem_lck);

    e->memory = NULL;
}
```

Figure 3: Memory free routine. This routine is preformed on all event allocations. B is the memory buffer. E is the event.

4.1 Benchmark Multicast Model

For the performance study we implemented a benchmark multicast model. We constructed binary trees to describe the network topology of sources, routers and subscribers. All of the trees were disjointed. The leaf nodes off the routers were the subscribers. The source root node was responsible for generating the packets. Once the packet was received by the left-most subscriber in the tree, the root will generate the next packet.

4.2 Model Parameters and Results

We experimented with many parameters in the multicast benchmark model. The most significant model parameters were the number of LPs and the size of the shared data segments. We varied the number of trees from 2 to 16 and the number of levels from 5 to 15. A power of two was not chosen because 15 was the largest number of levels that would still fit into memory. The shared data size ranged from 4 integers to 1024 integers and was modeled using individual memory buffers of the respective sizes. In addition we varied the number of start events from 2 to 8.

For the first set of experiments we ran ROSS sequentially with and without a shared data segment in the events.

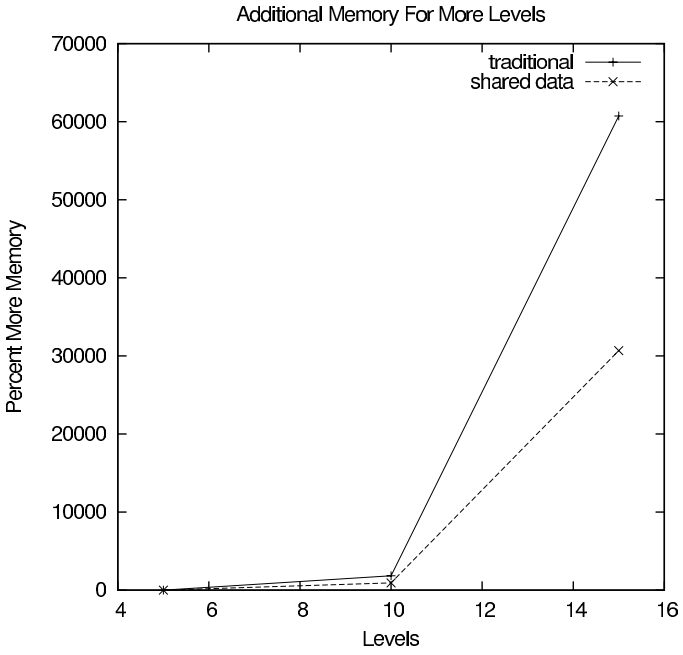

Figure 4: Memory required with respect to levels.

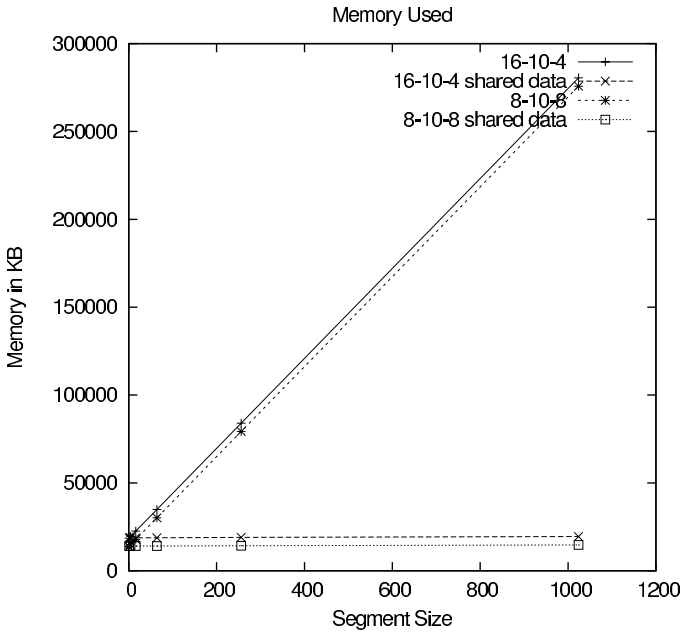

Figure 5: Memory required with respect to shared data segment size. One case is 8 multicast trees, 10 levels and 8 start events and the other case is 16 trees, 10 levels and 4 start events.

Obviously, as the number of trees and start events increase the memory increase according. When the number of levels in the trees grow, an exponential increase in memory usage was experienced. This can be seen in Figure 4 and is explained by the exponential nature of the data structure.

It can be observed that there was a smaller benefit for the small shared data segments. However, in the larger sized data segments the benefits are quite noticeable. This can be seen in Figures 5 and 6 and Table 1. In the larger cases the share data segments lead to significant decreases in memory and increases in performance. The decrease in memory is explained by the fact that the over head of the share data segment is surpassed by the duplicate information. In some cases the shared data models used 1/20th of the memory required by traditional sequential simulations (i.e., not sharing event segments). One observed result showed a speedup of 2.5. This performance is explained by the fact that the traditional model was in swap and thrashing. For the other data points, the speedup is attributed to a smaller memory footprint which enables more events to fit in the cache. Another part of the speedup was the model only had to assign values to pointers instead of copying data from events.

Table 2 is the profiling results of the models. It shows the data cache misses per memory reference for the shared event data and traditional models. The ratio was obtained by dividing DCU_LINES_IN by DATA_MEM_REFS which are counters that Oprofile monitored on a Pentium III. The table shows that as the event data size increases the shared event data model has fewer data cache misses than the

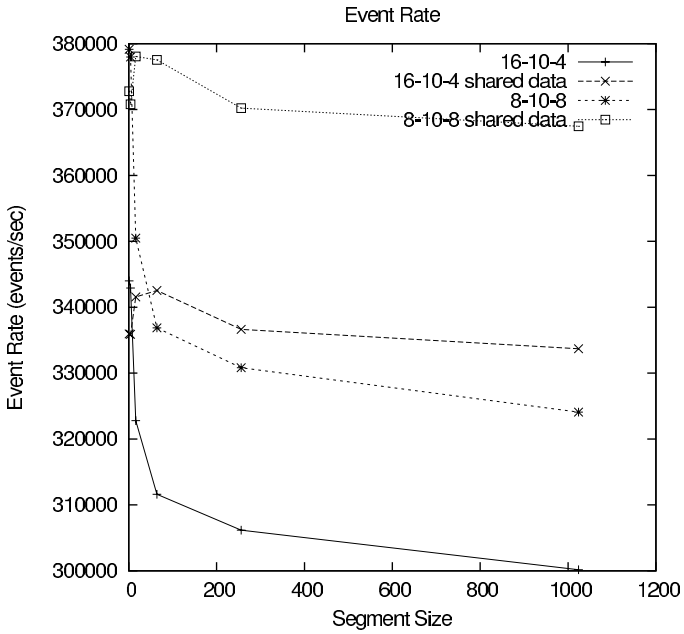

Figure 6: Event Rate with respect to shared data segment size. One case is 8 multicast trees, 10 levels and 8 start events and the other case is 16 trees, 10 levels and 4 start events

Table 1: Sequential Performance with and without shared data. T is the number of trees in the multicast graph. L denotes the number of level within each tree. E_{start} is the number of initial events each LP schedules at the start of the simulation. S is the size of the data size in the messages. $M_{traditional}$ and M_{shared} are the required memory for the traditional and shared event data models respectively. $ER_{traditional}$ and ER_{shared} are the event rate for the traditional and shared event data models respectively.

T	L	E_{start}	S	$M_{traditional}$	M_{shared}	$ER_{traditional}$	ER_{shared}
8	10	8	4	14.4 MB	14.8 MB	377986.673	370801.924
8	10	8	16	17.4 MB	14.8 MB	350491.922	378053.970
8	10	8	64	29.4 MB	14.8 MB	336866.703	377541.440
8	10	8	256	77.4 MB	14.9 MB	330823.230	370219.797
8	10	8	1024	269.3 MB	15.4 MB	324083.951	367463.726
16	10	4	4	19.1 MB	19.4 MB	342921.408	335885.514
16	10	4	16	22.1 MB	19.4 MB	322796.458	341562.543
16	10	4	64	34.1 MB	19.4 MB	311617.293	342541.110
16	10	4	256	82.0 MB	19.6 MB	306166.442	336635.595
16	10	4	1024	273.9 MB	20.0 MB	300169.391	333708.054
16	15	8	256	4936.3 MB	842.0 MB	137519.194	146734.510
16	15	8	1024	17224.1 MB	842.9 MB	57867.255	146681.164

Table 2: Data cache misses per memory reference. T is the number of trees in the multicast graph. L denotes the number of level within each tree. E_{start} is the number of initial events each LP schedules at the start of the simulation. MR_{shared} and $MR_{traditional}$ are the data cache misses rates for the shared event data and traditional models respectively. Finally, *% Reduction* is the amount the miss rate is reduced by the event sharing scheme.

T	L	E_{start}	S	MR_{shared}	$MR_{traditional}$	*% Reduction*
8	10	8	4	0.0221	0.0221	0.00 %
8	10	8	16	0.0221	0.0211	-4.73 %
8	10	8	64	0.0221	0.0231	4.33 %
8	10	8	256	0.0224	0.0259	13.51 %
8	10	8	1024	0.0224	0.0290	22.75 %
16	10	4	4	0.0224	0.0224	0.00 %
16	10	4	16	0.0225	0.0213	-5.63 %
16	10	4	64	0.0228	0.0234	2.56 %
16	10	4	256	0.0228	0.0261	12.64 %
16	10	4	1024	0.0229	0.0293	21.84 %

traditional model. These fewer misses can also explain the speedup which is shown in table 1.

One out-lier result was in the 16 integer case, the traditional model had a better ratio. Certain models are more sensitive than others to how the model fits into the L2 cache, yielding better performance in some cases. It appears that the 16 integer case was one of these situations. More investigation is needed to determine the precise effects of caching on performance.

Table 3 shows the result of the tests on the 1.5 GHz quad processors Itanium-IIs. The maximum speedup attain was 3.22 on four processors. The low values of speedups can be explained by the fact that the systems does not have enough work and can be remedied by increases the number

Table 3: Parallel results for shared event data. T is the number of trees in the multicast graph. L denotes the number of level within each tree. E_{start} is the number of initial events each LP schedules at the start of the simulation. *2-4 PEs* is performance measured in speedup (i.e., sequential execution divided by parallel execution time) for 2 to 4 processors.

T	L	E_{start}	S	*2 PEs*	*3 PEs*	*4 PEs*
8	10	4	256	1.36	1.95	2.34
8	10	4	1024	1.35	1.94	2.32
8	10	8	256	1.49	2.22	2.73
8	10	8	1024	1.48	2.21	2.74
8	15	4	256	1.56	2.43	3.21
8	15	4	1024	1.55	2.43	3.22
8	15	8	256	1.54	2.41	3.19
8	15	8	1024	1.54	2.41	3.19
16	10	4	256	1.51	2.25	2.80
16	10	4	1024	1.51	2.23	2.79
16	10	8	256	1.59	2.40	2.94
16	10	8	1024	1.59	2.43	2.64
16	15	4	256	1.54	2.42	3.20
16	15	4	1024	1.53	2.42	3.20
16	15	8	256	1.53	2.38	3.06
16	15	8	1024	1.54	2.37	3.04

of start events in the system. This can also be observe in the table.

5 RELATED WORK

Much of the research in parallel simulation for shared data was based on modifying and reading multiple LP's states. For example sharks world breaks a model down into sectors and each sector needed to be able to read or modify entities on its neighbor state (Conklin, Cleary, and Unger 1990). One method would be to use the push method, in which messages are passed to the correct neighbors with the entities information. The other way is posed in the space-time memory paper (Ghosh and Fujimoto 1991). This concept has shared objects with a time log attached to them. It allows for a easier model development over the push method. A distributed method for sharing variables is discussed in (Mehl and Hammes 1993). The main difference between this paper and these other papers is that our shared memory is not allow to be modify. This eliminates the issue of whether the memory is safe to read.

In the context of shared memory performance optimization, Panesar and Fujimoto have two key results. In (Panesar and Fujimoto 1995), they present a event buffer management scheme that reduces memory overheads on a cache-coherent shared memory multiprocessor (KSR systems). To efficiently avoid over-optimistic execution, as well as ensure that event memory is equally distributed among all processors, they devise a control flow technique which treats event memory like a window of network packets and apply a congestion control approach to throttling Time Warp event processing rates (Panesar and Fujimoto 1997).

Multicast is also used in the High-Level Architecture (fujimoto 2000, HLA 2005) also known as IEEE 1516. This is a general purpose architecture for simulation reuse and interoperability. Here, simulators communicate through a publish and subscribe interface. One of the key challenges is how to correctly disseminate update information. To address this problem, multicast groups are employed as a means to allow simulators to subscribe to regions of interest. Each "region" is assigned a multicast group identifier. This approach enables the efficient dissemination of update information about simulation entities of interest. The key difference here is that our shared-memory approach reduces memory consumption whereas the HLA's implementation reduces network bandwidth, but overall memory consumptions remains the same.

Finally, we note that sharing event data has some linkages to multi-resolution modeling (Natrajan, Reynolds, and

Srinivasan 1997). Here, MRM is primarily concerned with the correct temporal and spatial aggregation and disaggregation of simulation objects. The key difference between our approach is that we are only concerned with a spatial aggregation of event data that would be scheduled to a number of simulation objects at or about the same point in virtual time. Additionally, we are unaware of any MRM approach for an optimistic synchronization environment. In particular, how one would rollback either an aggregation or disaggregation operation is still an open question.

6 CONCLUSIONS

From the idea of shared data in the LP we transform it into the idea of shared data in the event. This paper shows that the idea of shared event data is possible and shows that there are benefits of 2 to 20 in memory savings. There are also speedup gains from this idea due to eliminating the copying for hop to hop. We show that it can be implemented in all major types of simulation engines. In addition we show parallel speedups of 3.22 on a quad processor system.

AUTHOR BIOGRAPHIES

GARRETT YAUN is a member of technical staff at Google.com. He recieved his Ph.D., M.S., and B.S. in Computer Science from Rensselaer Polytechnic Institute (RPI) in 2005, 2003 and 2000 respectively. His research interests include parallel and distributed systems, networking, modeling, and simulation. His e-mail address is <yaung@cs.rpi.edu>.

DAVID BAUER is a Ph.D. candidate in Computer Science Department of Rensselaer Polytechnic Institute (RPI). His research interests include parallel and distributed systems and network simulation with a focus on performance optimizations. His e-mail address is <bauerd@cs.rpi.edu>.

CHRISTOPHER D. CAROTHERS is an Associate Professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received the Ph.D., M.S., and B.S. from Georgia Institute of Technology in 1997, 1996, and 1991, respectively. Prior to joining RPI, he was a research scientist at the Georgia Institute of Technology. His research interests include parallel and distributed systems, simulation, networking, and computer architecture. His e-mail address is <chrisc@cs.rpi.edu>.