HARDWARE ACCELERATION FOR CONSERVATIVE PARALLEL DISCRETE EVENT SIMULATION ON MULTI-CORE SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Elizabeth Whitaker Lynch

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2011

HARDWARE ACCELERATION FOR CONSERVATIVE PARALLEL DISCRETE EVENT SIMULATION ON MULTI-CORE SYSTEMS

Approved by:

Dr. George Riley, Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. George Riley, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Yorai Wardi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Bonnie Ferri
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Thomas Conte
School of Computer Science
Georgia Institute of Technology

Dr. Arun Rodrigues
Computer Science Research Institute
Sandia National Laboratories

Date Approved: February 4, 2011

To my husband, Eric

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. George Riley. He inspired me to pursue Computer Engineering when he taught my Introduction to Computer Engineering class, nine years ago. Since then, he has given me his patience, advice, and guidance.

I would like to thank my committee, Drs. Sudha Yalamanchili, Tom Conte, Yorai Wardi, Bonnie Ferri, and Arun Rodrigues. Their advice and suggestions have been invaluable during this process. I would especially like to thank Dr. Rodrigues for travelling across the country to attend my thesis defense and offer his interesting insights.

I would also like to thank my family. They have always been eager to provide advice, encouragement and editing. I would never have made it through this without them. My friends and colleagues have also provided me with support, advice, and laughter. Finally, I would like to thank my husband, Eric. I could not imagine going through this Ph.D. adventure with anyone else.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In the past decade, chip manufacturers have begun producing chips with more and more cores, rather than increasing clock frequency as a means to increase chip performance. However, the performance of applications is often not improved by the addition of multiple cores, especially when the applications require frequent communication. This occurs when the overhead generated by communication and contention for resources outweighs the benefit of dividing the computation between the cores. One common class of applications that has large communication overhead is *parallel discrete event simulation*.

Discrete event simulation is a technique commonly used to model physical systems as a series of events which occur at discrete points in time. These simulations are commonly parallelized when either the state of the model is too large to fit into the memory of a single processor, or the runtime of the simulation is too long. Parallel discrete event simulation has two main sources of overhead, time synchronization and message passing.

The goal of this thesis is to decrease these sources of overhead for parallel discrete event simulators on multi-core systems through the use of specialized hardware. By using the proposed specialized hardware units for both time synchronization and message passing, the communication required by the simulation will be greatly reduced, decreasing the runtime of the simulation.

The contributions of this work are as follows:

- We present the design for a Global Synchronization Unit, which performs the time synchronization for parallel discrete event simulators on multi-core systems. We have demonstrated a 40% reduction in runtimes for a parallel network

simulation using this specialized hardware on up to 32 cores.

- We have also introduced a software implementation of the Global Synchronization Unit, which runs as a separate thread or process. This software implementation can be used when the cost of specialized hardware is prohibitive or on existing multi-core systems.

- We present the design for two hardware units, the Atomic Shared Heap and Atomic Message Passing, which are used together to perform zero-copy message passing on multi-core systems. We have demonstrated a 16% decrease in runtime using these devices in a parallel simulation on up to 32 cores.

- Finally, we introduce software implementations of the Atomic Shared Heap and Atomic Message Passing, with each implemented as a separate thread or process. These software implementations can be used together to perform zero-copy message passing on existing multi-core systems or when the cost of specialized hardware is too great.

CHAPTER I

INTRODUCTION

In the past decade, due to the limits of transistor size and power usage, chip manufacturers have started to turn to multiple cores on a chip instead of increasing clock frequency as a means to increase chip performance. In the past 5 years, multi-core architectures have become common, and core counts continue to increase. There are six- and eight-core chips currently in production, such as Intel Gulftown, and many-core chips with dozens of cores, such as the Intel Teraflops 80-core chip, are projected in the next five years [26]. However, adding more cores often does not improve the performance of applications, especially when frequent communication between the cores is required. This is because the overhead added by communication can outweigh the benefit yielded by dividing the computation between the cores. One example of these communication-heavy applications is *discrete event simulation*.

Discrete event simulation is a technique used to model physical systems by representing changes in the system as a series of events that occur at discrete points in time. In many cases, the state of the model for the system is too large to fit into memory, or the runtime for the simulation is too long for a single CPU. In these situations, it can be advantageous to parallelize the simulator. To achieve this, the model is partitioned into pieces that are each assigned to a different processor. This partitioning means that each processor only performs a fraction of the computation, and the computations on different processors can run in parallel, potentially decreasing the total runtime. However, there is some overhead added in the parallelization process. In cases where the new event generated is destined for the same partition as the event generating it, the event is simply inserted into the queue in the local

simulator kernel. However, when the event is intended for a portion of the model in a different partition, the event is serialized into a message, which is sent across processes, either over the network or through shared memory. In addition, the processes must periodically synchronize with each other to prevent one process from running too far ahead, which could result in incorrect results due to a process receiving events in its logical past. This *time synchronization* traditionally requires a global barrier or all-to-all reduction, such that no process can proceed until the slowest process has entered the barrier or reduction.

The frequency with which the time synchronization must be performed is determined by the *lookahead*. Lookahead is dictated by the physical properties of the system being generated, for example, the speed-of-light delay on a network link. When a packet is sent from one router to another, there is a minimum amount of time between when it leaves the sending router and is received at the destination. This minimum is determined by the properties of the network link, such as the speed-of-light delay on the wire. A parallel simulation can take advantage of these properties to determine the maximum allowable difference of simulation time between the simulation clocks of processes. When process A processes an event with a timestamp t, which generates a new event with timestamp $t + x$ for process B, lookahead translates to the minimum possible value for x. This means that a process can assume that it will receive no new messages with timestamp less than the smallest timestamp of the unprocessed events in the simulation, plus the lookahead. This assumption allows processes to process all pending events with timestamps less than that time, with the knowledge that the process will not receive any events that occur in its logical past. When the lookahead is low, synchronization must be performed more frequently, and fewer events are processed between synchronizations.

The goal of this work is to take advantage of the multi-core environment to speed

up parallel discrete event simulation.This is desirable because of both the ready availability of multi-core chips and because of the low-latency communication that is possible between the cores, which are in close proximity on the same chip. The current bottleneck for many parallel simulations is time synchronization. This is especially true for simulations of wireless networks and on-chip networks. These two classes of problems have physical properties (wireless transmission times and small link delays on-chip) which translate to low lookahead in the simulations. As a result, these simulations have high amounts of overhead from time synchronization and do not scale well. Message passing, as the other source of communication between the processes, is also a common simulation bottleneck. Not only do the simulations of wireless and on-chip networks scale poorly, they are also simulation domains that are highly desired today.

Wireless networks have become nearly ubiquitous, resulting in potential poor performance due to conflicts between overlapping networks. On-chip networks, such as those on multi- and many-core chips are under high loads and are frequently the bottleneck in application performance. As a result, network designers need access to high-fidelity, efficient simulators to help them predict the performance of current and future networks under these loads.

This work includes the design of hardware at a functional level that performs the time synchronization for parallel discrete event simulation asynchronously and in just a few clock cycles, eliminating the need for global communication with message passing or lock contention for shared memory. This hardware, which is called the *Global Synchronization Unit* (GSU), consists of three register files, each the size of the number of cores, and is accessed using five new atomic instructions. A performance study has been conducted using the Georgia Tech Network Simulator (GTNetS) running inside the Simics system simulator. We determined that for up to 32 cores the hardware unit reduces the runtime of a low-lookahead parallel network simulation by 40

percent over a shared-memory barrier implementation of synchronization. Although the GSU has not been designed at a gate-level to determine how long these atomic instructions will take to execute, we showed that even if they take hundreds of cycles each, there will still be a significant performance improvement over a shared-memory implementation. In addition, an estimate of the delays has been determined, falling well within the acceptable range, as determined by the sensitivity analysis.

In order to reduce the simulation overhead from message passing, two independent pieces of hardware have also been designed at a functional level, the *Atomic Shared Heap* (ASH) and *Atomic Message Passing* (AMP), which can be used to perform zero-copy message passing on a multi-core system. The Atomic Shared Heap is composed of registers that, for a system with N cores, have a width of $lg(N)$. The ASH contains one register for each allocatable unit of the heap and is accessed using three atomic instructions: *Allocate*, *IncrementUsage*, and *Free*. The Atomic Message Passing unit contains N circular queues, each of size k . It also uses two atomic instructions, *Read* and *Write*. A performance study has been conducted of ASH and AMP used in conjunction, again using GTNetS and Simics. From this study, it has been determined that ASH and AMP reduce the runtime of a low-lookahead parallel network simulation running on 32 cores by 16 percent over a version using traditional shared memory message passing.

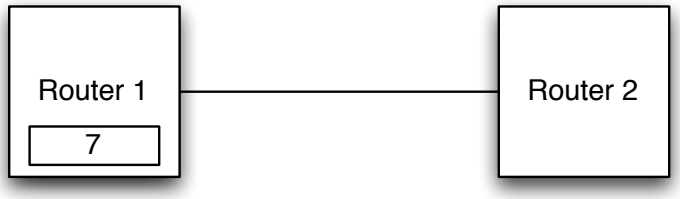
The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of the background and related work in the areas of discrete event simulation, time synchronization, lock-free message passing, and hardware acceleration. Chapter 3 is a description of the functional-level design for the Global Synchronization Unit, as well as a description of the experiments with it and their results. Chapter 4 contains descriptions of the Atomic Shared Heap and Atomic Message Passing, along with the description of their performance analysis and the results. Finally, Chapter 5 covers the conclusions from this work and future work to be done in this area.

CHAPTER II

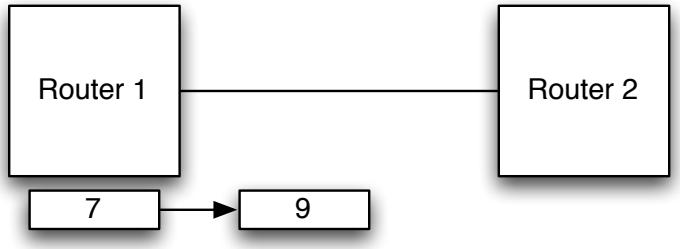
ORIGIN AND HISTORY OF THE PROBLEM

Discrete event simulation is a technique used to model physical systems. In a discrete event simulator, changes in the state of the system are represented by *events* with corresponding timestamps. The simulator kernel maintains the current simulation time, as well as a list of scheduled events, which are processed in chronological order. The requirement that events must be processed chronologically is known as the *causality* constraint, and is necessary to guarantee correctness of the simulation [11]. When an event is processed, it can generate zero or more new events, which are inserted into the simulator's event queue. Other than the initial events inserted in the queue when the simulation starts, this is the only way that new events are generated. As an example, in Figure 1 a router model receives an event notifying it of a packet arrival on one of its links at time $t = 7$. When the router model processes this event, it generates a new packet arrival event for the router to which the packet is being forwarded, with a timestamp of the current time plus the link delay and the transmission time, $t + d + tx = 7 + 1 + 1 = 9$.

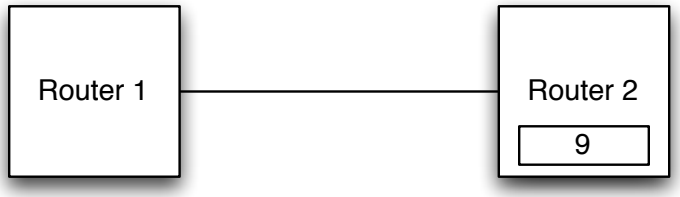
When the model of the system being simulated is too large to fit into memory or the runtime of the simulation is too long, one approach to mitigate this problem is to parallelize the simulation. This is achieved by partitioning the model into multiple *logical processes*(LPs), which are assigned to different processors, as seen in Figure 2. Each LP executes the events that affect the portion of the model assigned to it. If an event generates a new event that affects a portion of the model that resides on a different logical process, the event must be sent in a *message*. However, the destination LP may have processed events with timestamps greater than the timestamp of the



(a) Router 1 has a pending event with a timestamp of t=7.



(b) Router 1 processes the event, which generates a packet arrival event for Router 2 with a timestamp of t+d+tx=9.



(c) The event with timestamp of 9 is scheduled for Router 2.

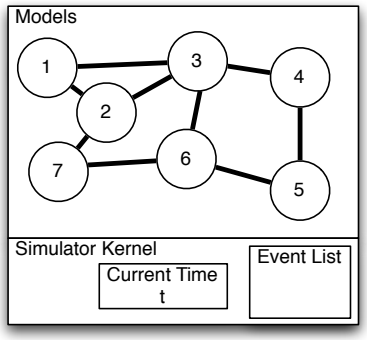
Figure 1: An example of how events are processed and generated.

message. This means the receiving process could receive a message that occurred in its logical past, violating the causality constraint. Violations of the causality constraint may result in the simulation returning incorrect results. For instance, in the example shown in Figure 3, say that in the previous example the second router is assigned to LP B and the first router is located on LP A. LP B has progressed to time $t + d + tx + x = 10$ by the time that it receives the message from LP A with timestamp $t + d + tx = 9$ and the router has processed another packet received event with a timestamp greater than $t + d + tx$. In this case, the second packet will arrive before the first, giving the user an incorrect answer. In order to preserve causality and the correctness of the simulation, the logical processes must be synchronized periodically.

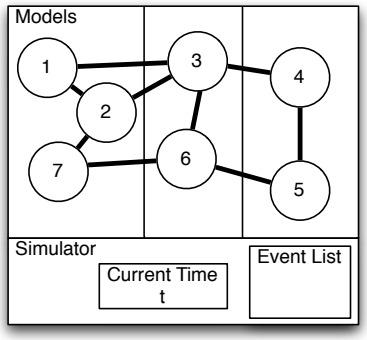
2.1 Time Synchronization

Time synchronization algorithms can be categorized into two main approaches, *conservative* and *optimistic*. Conservative synchronization works by preventing causality violations, while optimistic synchronization handles and corrects causality violations once they occur. In addition, synchronization algorithms can be classified as either *synchronous* or *asynchronous*. A synchronous algorithm requires simultaneous participation by each simulator instance, while asynchronous algorithms do not.

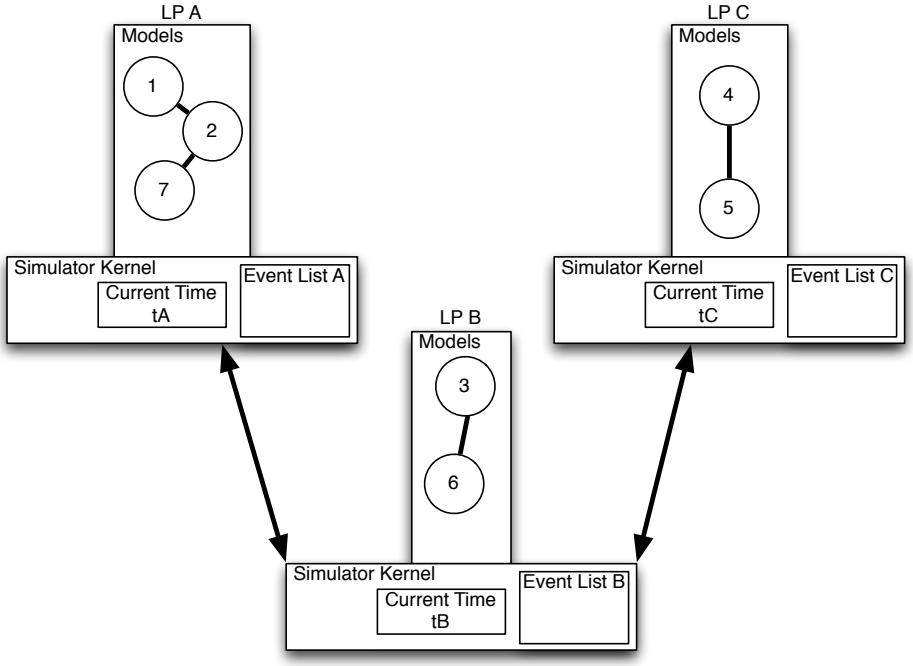
Another distinguishing characteristic is how the algorithm handles *transient messages*. Transient messages are messages that have been sent, but not yet received or processed by the recipient. A problem arises when a message in transit has the smallest timestamp of all unprocessed events in the system. In this situation, if the synchronization is performed while the message is transient, the result returned by the synchronization algorithm would be incorrect, as the timestamp of the transient message will not be included in the synchronization computation. This will allow the LPs to process events with timestamps greater than that of the transient message.



(a) A serial simulator with a system model and a simulator kernel, containing the current simulation time and a list of unprocessed events.

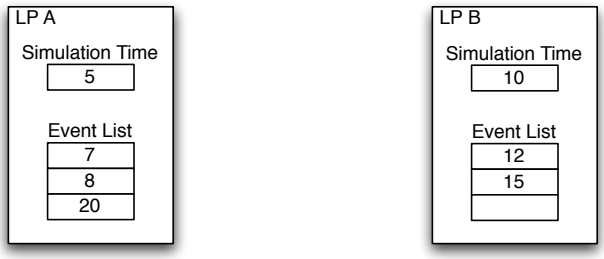


(b) The system model is partitioned into several pieces.

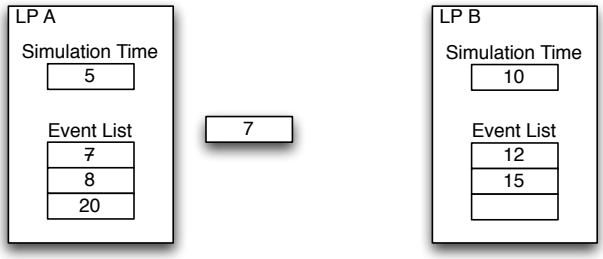


(c) The partitions are each assigned to an LP, which maintains its own simulation time and a list of unprocessed events destined for the portion of the model assigned to it.

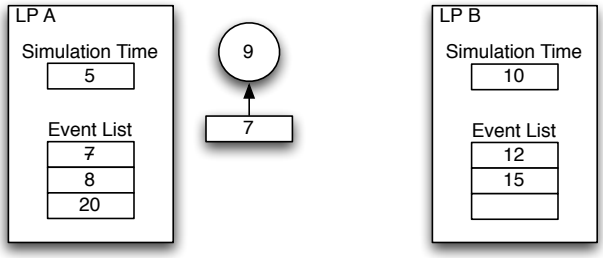
Figure 2: A serial simulation model is partitioned for parallel simulation.



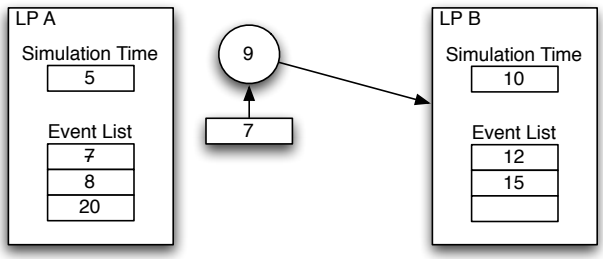
(a) LP A has a current simulation time of 5 and LP B has a current simulation time of 10.



(b) LP A removes the next event from its queue, which has a timestamp of 7.



(c) LP A processes the event, which generates a new event with a timestamp of 9, which is destined for LP B.



(d) LP A sends the new event in a message to LP B. However, the timestamp of this event, 9, is less than the current simulation time of LP B, which is 10.

Figure 3: An example of a causality error.

This means that when the message is received and processed, its timestamp will be in the logical past, resulting in a causality error. Synchronization algorithms can either account for transient messages or require that each message be acknowledged by the recipient.

2.1.1 Conservative Synchronization

The first time synchronization algorithm developed for Parallel Discrete Event Simulation(PDES) is the Chandy-Misra-Bryant algorithm [4, 3, 2]. Chandy-Misra-Bryant is an asynchronous conservative algorithm that uses *null messages* to synchronize LPs. An LP sends a null message to its logical neighbors after it processes an event. These null messages contain the timestamp of the smallest unprocessed event on the sending LP, plus the *lookahead* between the sending and receiving LPs. Lookahead is the minimum simulation time between the event that generates a message on one LP and the timestamp of that message, which is destined for another LP. The lookahead is determined by the physical properties of the systems being modeled, for example, the speed of light delay on a network link. One downside to this approach is that it greatly increases the number of messages sent. In a variation on the algorithm, as proposed by Misra [15], null messages are not sent until requested. An LP only requests a null message when it is out of "safe" messages to process.

The simplest synchronous time synchronization algorithm uses *lockstep synchronization*, also called *time-stepped simulation*. This algorithm is implemented using global barriers. When using lockstep synchronization, the minimum lookahead in the entire system, l, is assumed as the lookahead. Looking at the example in Figure 4, the lookahead between LP A and LP B is the minimum of the delays on the links crossing the LP boundary between them, $min(6, 3, 5) = 3$. The lookahead between LP B and LP C is the minimum of the delays on the links crossing that boundary, $min(4, 5) = 4$. This means that the lookahead for the simulation is the minimum of

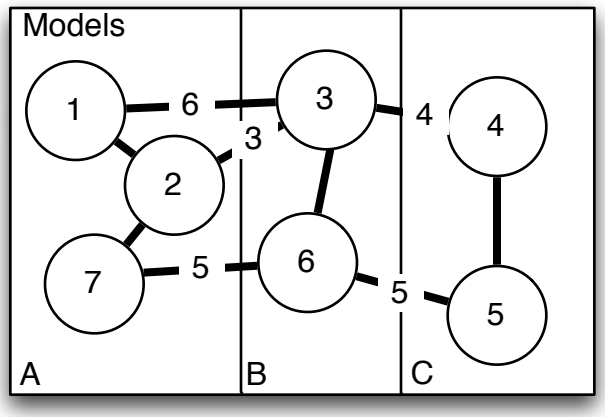


Figure 4: An example of how lookahead for the simulation is computed.

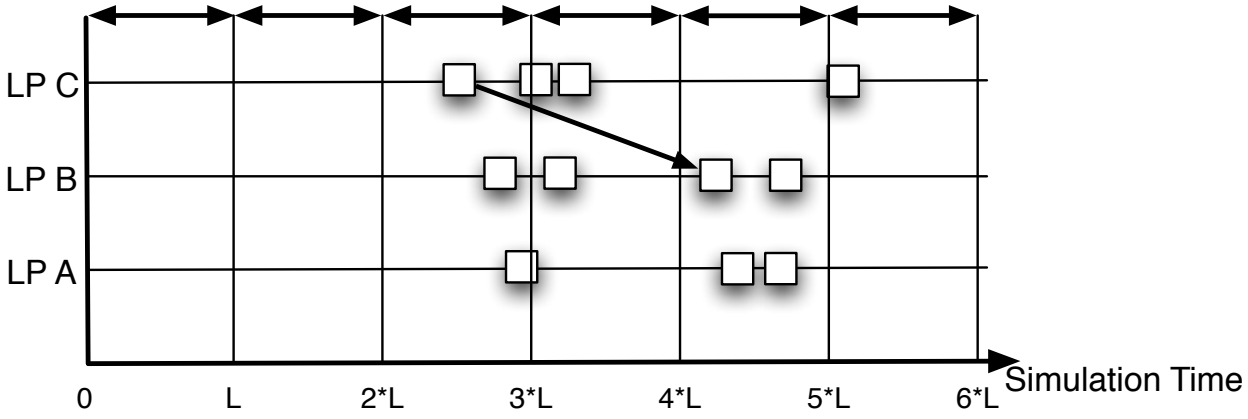


Figure 5: A timeline of events in a simulation using lockstep synchronization.

all the lookaheads in the system, $l = min(3, 4) = 3$. As seen in Figure 5, starting with simulation time 0, each LP then processes any events it has with timestamp less than l. Each LP then enters into a global barrier, and once all LPs have entered, each exits the barrier and begins processing events again, until the next barrier at simulation time $2l$. This algorithm has the disadvantage of only being able to advance time by l with each synchronization. In cases with bursty activity, where there are long stretches with no events, this could introduce significant overhead. For instance, in the example in Figure 5 we can see that six barrier synchronizations are required to complete the simulation.

Another conservative algorithm uses barriers to calculate the *Lower Bound on*

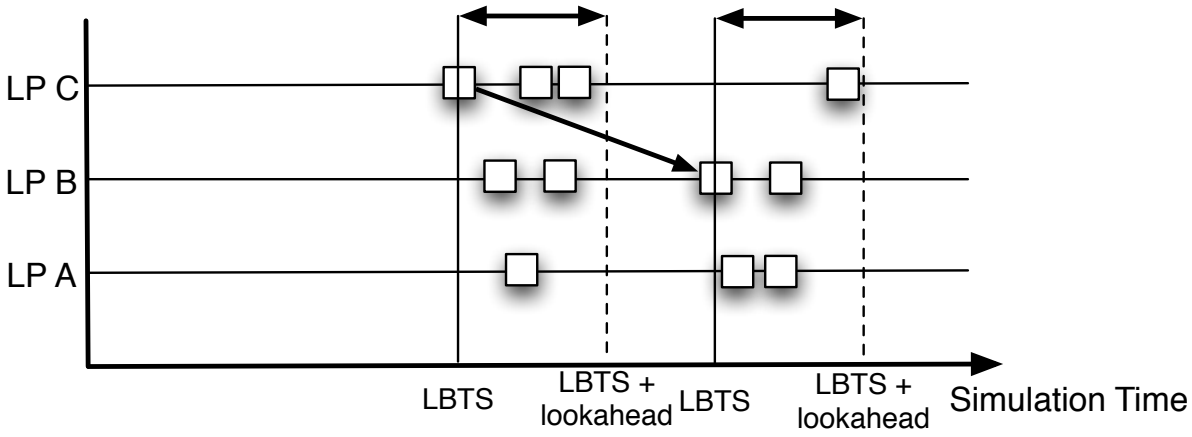


Figure 6: A timeline of events in a simulation using LBTS synchronization.

Timestamps(LBTS) [16]. In this approach, the minimum lookahead in the entire system, l, is assumed as the lookahead. Each LP processes any events it has with timestamps less than the last computed LBTS, t, plus l, as seen in Figure 6. The LPs then enter the barrier and send every other LP the timestamp of their next unprocessed event, the number of messages sent, and the number of messages received. Each LP then checks to see if the sum of all messages received is equal to the sum of all messages sent. If it is not, there are transient messages, so each LP checks for received messages and inserts any new events into the event queue before repeating the synchronization. If the sums are equal, each LP takes the minimum of all the event times as the new LBTS. As seen in Figure 6, for the same simulation as in Figure 5 only two LBTS computations are required, versus the six required for lockstep synchronization.

2.1.2 Optimistic Synchronization

One of the first approaches using optimistic time synchronization for simulation was Time Warp [9]. In this algorithm, LPs process events with the assumption that they are safe. If a message is received with a timestamp less than the current simulation time, the simulation is rolled back to a time before the timestamp of the message, using a checkpoint of the simulation state that was saved previously. When a *rollback*

is performed, the state of the simulation is reverted back to a state that is known to be correct from a time in the past. In addition, all the messages that have been sent by the LP being rolled back since the timestamp of the checkpoint it is reverting to must be cancelled, through the use of *anti-messages*. In order to keep memory usage to a reasonable level, a checkpoint can be deleted after it can be guaranteed that it is no longer needed. This is called *fossil collection*. To determine whether a checkpoint is no longer needed, the LPs compute *Global Virtual Time*(GVT). GVT is the minimum timestamp of all unprocessed events on all LPs. This is equivalent to LBTS with a lookahead of zero. As in LBTS, the algorithm must account for transient messages.

In order to improve the performance of a Time Warp simulation, the frequency with which checkpoints are made can be increased or decreased, with the stipulation that there must always be one checkpoint with a timestamp prior to the current GVT. When checkpoints are saved more frequently, the simulation will not have to roll back as far when a message is received with a timestamp less than the current simulation time. However, there will be a larger memory footprint required to store the checkpoints and there will be greater overhead for the time required to save the checkpoints. Conversely, if checkpointing is performed less frequently, the simulation will have to roll back further in the case of a message with a timestamp in the logical past, but both the memory and runtime overhead for checkpointing will be reduced.

In an effort to control the overhead of optimistic synchronization, a class of algorithms was developed that uses *limited optimism*. In these algorithms, a bound is put on how far ahead of GVT an LP can execute. An example of an algorithm using limited optimism is the Moving Time Window (MTW) [22]. In the MTW algorithm, a parameter w is specified. LPs then execute events with timestamps less than $GVT+w$. In this way, MTW limits the execution time spent on rollbacks, as well as the amount of saved state required in memory. However, it can also limit parallelism

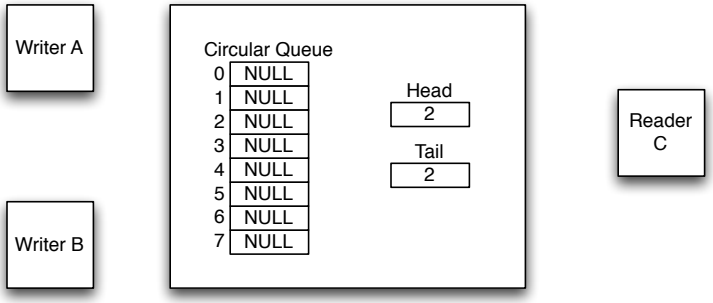
by forcing some LPs to block until GVT is advanced sufficiently.

2.2 Lock-Free Message Passing

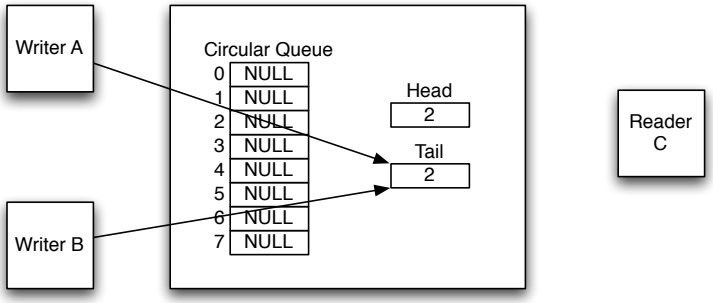
Message passing is another source of overhead in parallel discrete event simulation. When using shared memory, as in a multi-core system, contention for locks frequently introduces delays. There have been several proposals for lock-free message passing mechanisms.

The non-blocking queue proposed by Michael and Scott [14] is implemented using a linked list, the *compare-and-swap* atomic primitive, and two pointer variables, *Head* and *Tail*. The algorithm also handles the problem of pointer recycling. In this scenario, as illustrated in Figures 7 and 8, two writers check the value of the tail variable and read the same value. One writer then writes in the tail cell and updates the value of the tail variable, and a reader has also read the contents of the cell and updated the head variable. The second writer then mistakenly enqueues data in an inactive cell of the queue, as the tail value it read is outdated. In order to prevent pointer recycling, modification counters are used with the compare-and-swap operation. These modification counters are accessed whenever the pointers are read and incremented whenever compare-and-swap is called.

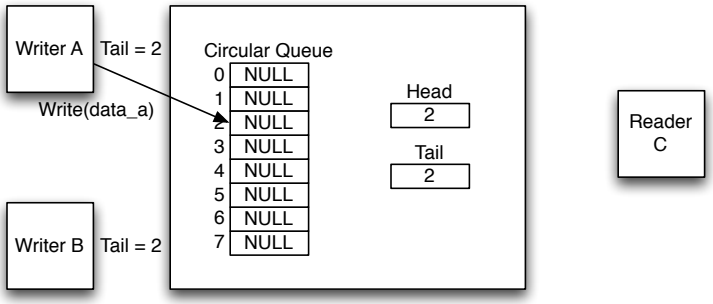
The non-blocking concurrent FIFO queue algorithm, as presented by Tsigas and Zhang [25], uses a circular array for message passing with the *compare-and-swap* atomic primitive and three variables: *head*, *tail*, and *vnull*, as seen in Figure 9. This algorithm uses two optimizations to improve performance over traditional circular queues. Firstly, it only updates the *head* or *tail* variables every m reads or writes. This approach requires more read operations in order to determine the true head or tail, but greatly decreases the number of more costly compare-and-swap operations. The parameter, m, can be optimized for the cost of read and compare-and-swap operations on the system. The second optimization uses the *vnull* variable to prevent



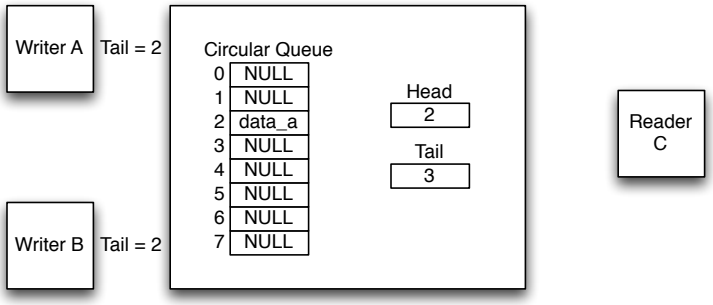
(a) The initial state of the circular queue.



(b) Writer A and Writer B both read the tail variable.

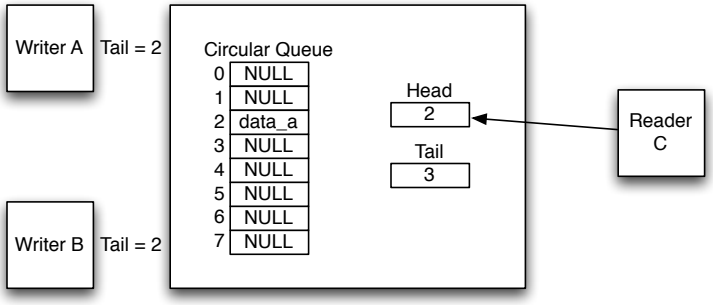


(c) Writer A writes data_a into slot 2 of the circular queue and increments the Tail variable.

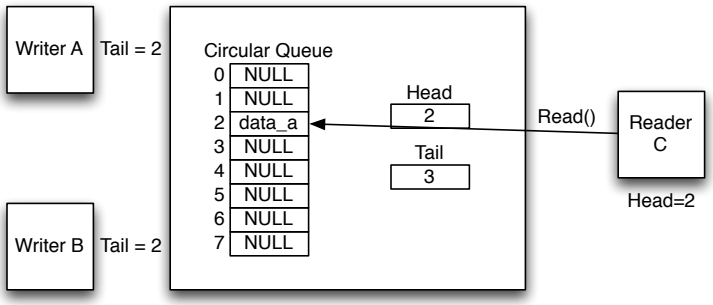


(d) The write is successful.

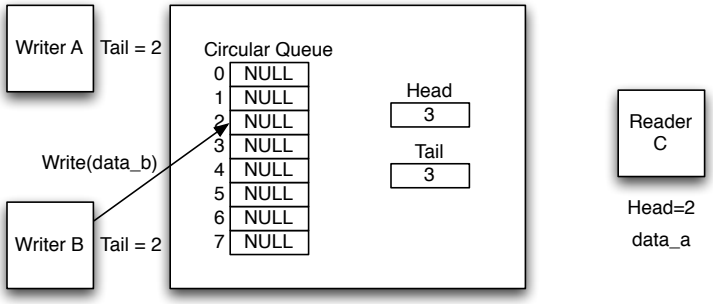
Figure 7: An illustration of the pointer recycling problem.



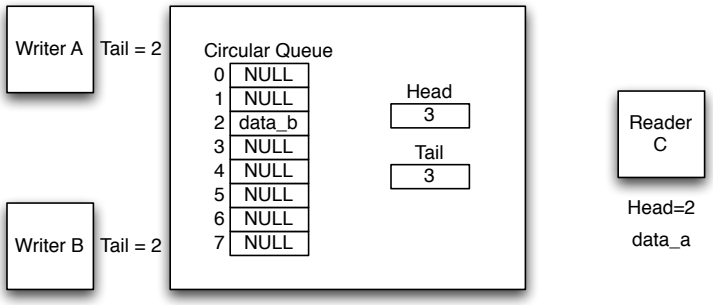
(a) Reader C reads the Head variable



(b) Reader C reads the data in slot 2 of the circular queue and increments the Head variable.



(c) Writer B incorrectly writes data_b into slot 2 of the circular queue, as it read the Tail variable before it was incremented.



(d) The data in slot 2 will not be read, as the Head variable has been incremented to 3.

Figure 8: A continuation of the illustration of the pointer recycling problem.

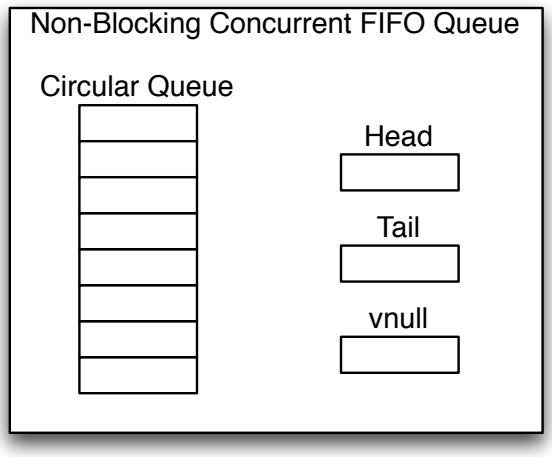


Figure 9: Data structures required for the non-blocking concurrent FIFO queue as designed by Tsigas and Zhang.

the problem of pointer recycling. By alternating two different values of NULL, always using the value contained in *vnull*, a writer can determine whether the cell has been used since the tail was determined, eliminating the pointer recycling problem.

Kim et al. proposed a *Non-Blocking Buffer* [10], which uses a circular queue with one producer and one consumer for message passing. This approach requires two counters, the *Update Counter* and the *Acknowledgement Counter*, as seen in Figure 10, and two access functions, *InsertItem* and *ReadItem*. The Update Counter contains the pointer to the next free slot for insertion and is modified by the producer. An odd value of the Update Counter indicates that an insertion is in progress, while an even value indicates that the insertion has been completed. The Acknowledgement Counter contains the pointer to the next slot to be read and is modified by the consumer. An odd value of the Acknowledgement Counter indicates that the slot is being read, and an even value indicates that the read operation has completed. These counters are also compared to determine if the queue is full or empty. When the producer calls InsertItem, the Update Counter is incremented, data is written into the buffer, and the Update Counter is incremented again. When ReadItem is called by the consumer, the Acknowledgement Counter is incremented, data is read from the buffer, and the

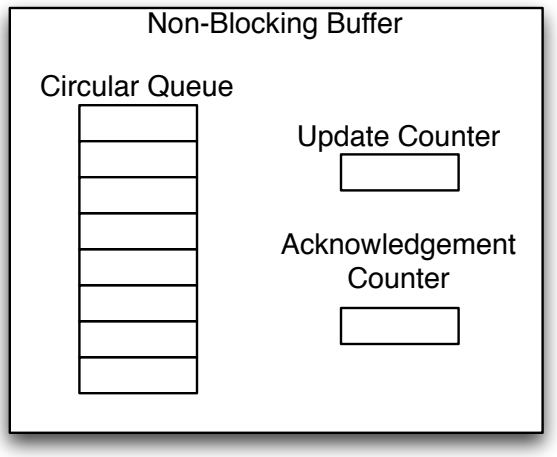


Figure 10: Data structures required for the Non-Blocking Buffer as designed by Kim et al.

Acknowledgement Counter is incremented again. The Non-Blocking Buffer approach requires that the producer and consumer each manage their own heap. The pointer written into the buffer by the producer points to the location of the data in the producer's heap. When the consumer reads the pointer in the ReadItem function, it copies the data from the producer's heap to its own heap and then returns the now-defunct pointer to the producer for re-use. In order for this approach to support multiple producers and consumers, there would need to be one Non-Blocking Buffer for each producer-consumer pair. This means that a system with N cores would need $N * (N - 1) = N^2 - N$ Non-Blocking Buffers.

2.3 Hardware Acceleration

One way to reduce the overhead of parallel simulation is to offload a task to specialized hardware. This hardware can be optimized for the particular function, allowing it to perform the task faster than a CPU. In addition, the specialized hardware can perform its task while the CPU continues processing events. Several different hardware accelerators have been proposed for use with parallel discrete event simulation.

The *Rollback Chip*, as proposed by Fujimoto et al. [6, 8], is a specialized piece

of hardware intended to be coupled with a single CPU. The CPU then offloads the checkpointing operations required by the Time Warp optimistic synchronization algorithm to its rollback chip. A small number of instructions are used by the CPU to access the rollback chip, *READ*, *WRITE*, *MARK*, *RESET*, *ROLLBACK*, and *ADVANCE*. READ is used to read the most recent data stored at particular address and WRITE is used to write data at a particular address. MARK stores the current version of all the data associated with the LP, while RESET initializes the hardware at the beginning of a simulation. Finally, ROLLBACK restores the LP data to a previous state, and ADVANCE indicates to the chip the number of states that can be removed to free up memory (states with timestamps less than GVT). Calculation of the GVT is handled by the CPU, as usual.

Non-blocking checkpointing, as proposed by Quaglia and Santoro [17], partitions simulation data into three categories, *State Buffers*(SB), *Checkpoint Stacks*(CS), and *Other Data Structures*(ODS). The algorithm requires a piece of commodity hardware capable of copying data from SB to CS, such as the Direct Memory Access hardware for Myrinet. The CPU can then issue a request to this hardware to create a checkpoint, and the hardware will copy the required data from SB to CS while the CPU continues the processing required for the simulation.

Rosu et al. have proposed a *Virtual Communication Machine*(VCM) [21], which would be implemented on the network co-processor. The VCM would have access to the application address space and maintain consistent replicas of not only the local LP's event queue, but also the event queue of every other LP involved in the simulation, through communication with the VCMs of remote LPs. As this gives the LP knowledge of the state of every other participating LP, the GVT can be calculated using only local data, requiring no global communication. One downside to this approach is that the replication of all event queues can lead to a large memory footprint.

Targeting shared memory multiprocessors, Fujimoto has proposed a variation on the Time Warp algorithm [7]. In this approach, LPs can write directly into other LPs' event queues, which are stored in shared memory. This eliminates the need for message passing. To send a message, an LP obtains a lock for the event queue of the destination LP, enqueues the new event, and releases the lock. If the sending LP later needs to cancel the event, it again obtains the lock for the event queue of the destination LP, and checks the *Processed* flag of the event. If the flag is set to FALSE, the sending LP can simply delete the event from the queue. If the flag is set to TRUE, the sending LP sets the flag to FALSE and cancels all the events generated by the event being cancelled. The GVT calculation for fossil collection is performed in the traditional manner, using a global barrier computation.

As proposed by Srinivisan and Reynolds [23], the *Parallel Reduction Network*(PRN) is a tree of ALUs with a depth of $lg(n)$, where n is the number of CPUs. The PRN uses an auxiliary processor coupled with each CPU, which is responsible for passing data into the PRN, in the form of a state vector, and passing the result of the PRN back to the application. The reductions of the elements in the state vector are pipelined through the PRN. Although the PRN can be used for many different reduction operations, one use for simulation is calculation of the GVT. When using the PRN to compute the GVT, the auxiliary processor passes in a state vector which contains the current simulation time of the LP, as well as the minimum timestamp of all messages sent by the LP which have not yet been acknowledged. The auxiliary processors are also responsible for the acknowledgement of all messages received, as required by this algorithm.

Another use of the Parallel Reduction Network is the *Elastic Time Algorithm* [24], an adaptive synchronization protocol. The Elastic Time Algorithm uses *Near-perfect State Information*(NPSI) to calculate an *Error Potential*(EP). The goal of the Elastic Time Algorithm is to minimize the total cost of the optimistic synchronization, as

defined by the equation *total cost* = *rollback cost* + *memory management cost* + *lost opportunity cost*. The rollback cost is determined by the time required to perform the rollbacks in the simulation, the memory management cost is determined by the time required to perform the checkpointing for the simulation, and the lost opportunity costs are determined by the time LPs spent blocking when it was safe for them to process events. The EP is an estimate of the likelihood that a given LP will perform a rollback in the near future. The Error Potential for an LP is calculated based on the smallest timestamp of the LP's pending messages, as well as the smallest timestamp of all the messages that have been sent by the LP but not yet received or enqueued at their destinations. This EP is used to limit the optimism in the simulation through the introduction of wall clock delays proportional to the EP between the processing of events. Using a PRN as a NPSI calculator, the EP is calculated with Input State Vectors from the LPs.

CHAPTER III

THE GLOBAL SYNCHRONIZATION UNIT

In order to help alleviate the significant overhead of time synchronization discussed previously, we have designed the Global Synchronization Unit. This centrally located hardware unit performs the time synchronization for parallel conservative discrete event simulation on a multi-core chip. In this section, the design of the Global Synchronization Unit is described, along with a proof of its correctness, an example of its use, and an analysis of its performance. In addition, there is a discussion of how the Global Synchronization Unit could be implemented in software.

3.1 Description of the Global Synchronization Unit

The *Global Synchronization Unit*(*GSU*) is a centrally located hardware unit on a multi-core chip of N cores consisting of three register files of depth N and width determined by the size of the timestamp datatype, likely 32- or 64-bits, with five atomic instructions used to access it. The GSU can return an LBTS value to an LP at any point during the simulation without requiring any global communication. In addition, the GSU accounts for transient messages, eliminating the need for acknowledgements or processing all pending messages before computing the LBTS.

We will first describe the functionality of each component of the *GSU*, followed by a discussion of how they will be used to support time synchronization in a conservative distributed discrete event simulation. In this discussion, N represents the number of CPUs in the multi-core system. Also, we will assume that the pairwise lookahead is uniform between each pair of *LPs*. Although it is generally not true that the system being modeled has uniform lookahead, this is a common assumption, which is handled by treating all lookahead values as if they are equal to the minimum lookahead in

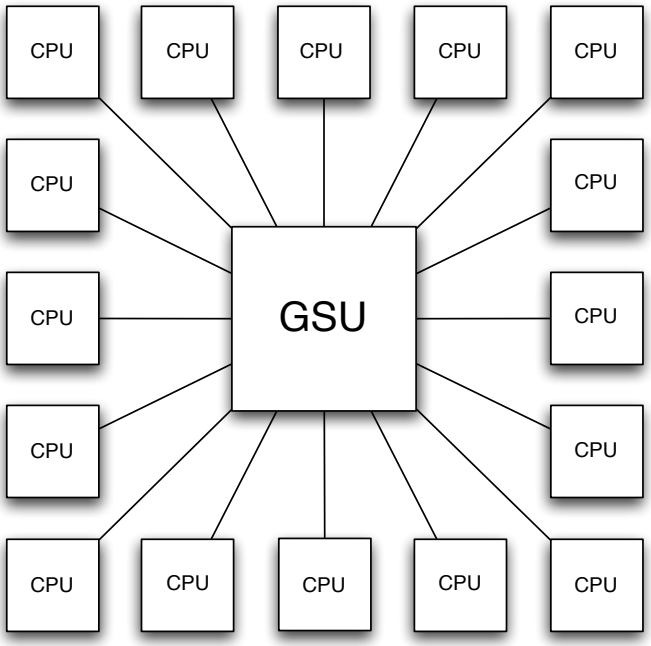


Figure 11: The Global Synchronization Unit is located on-chip with one unit per multi-core chip.

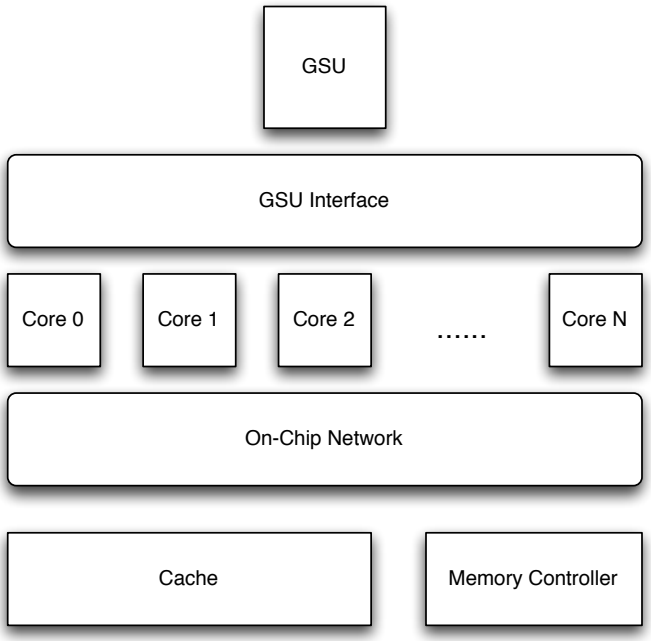


Figure 12: An example of how the GSU could be connected in a multi-core system

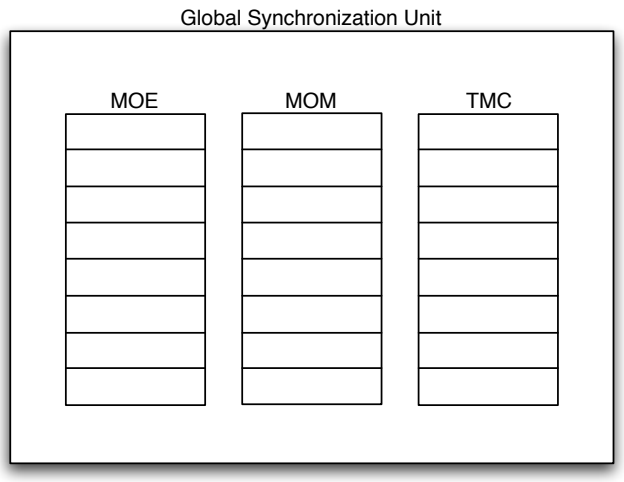


Figure 13: The Global Synchronization Unit

the system. However, the *GSU* can be extended to handle non-uniform lookahead, as described in section 3.9

3.1.1 Register Files

The *GSU* consists of three register files with a width of the size of the timestamp datatype and a depth equal to the number of cores on the chip. These three register files are the *Minimum Outstanding Event*(*MOE*), *Minimum Outstanding Message*(*MOM*), and *Transient Message Count*(*TMC*). A description of these register files follows.

- Minimum Outstanding Event (MOE) - Contains the timestamp of the earliest event in each LP's event queue.
- Minimum Outstanding Message (MOM) - Contains the minimum timestamp of all transient messages destined for each LP.
- Transient Message Count (TMC) - Contains a count of the transient messages destined for each LP.

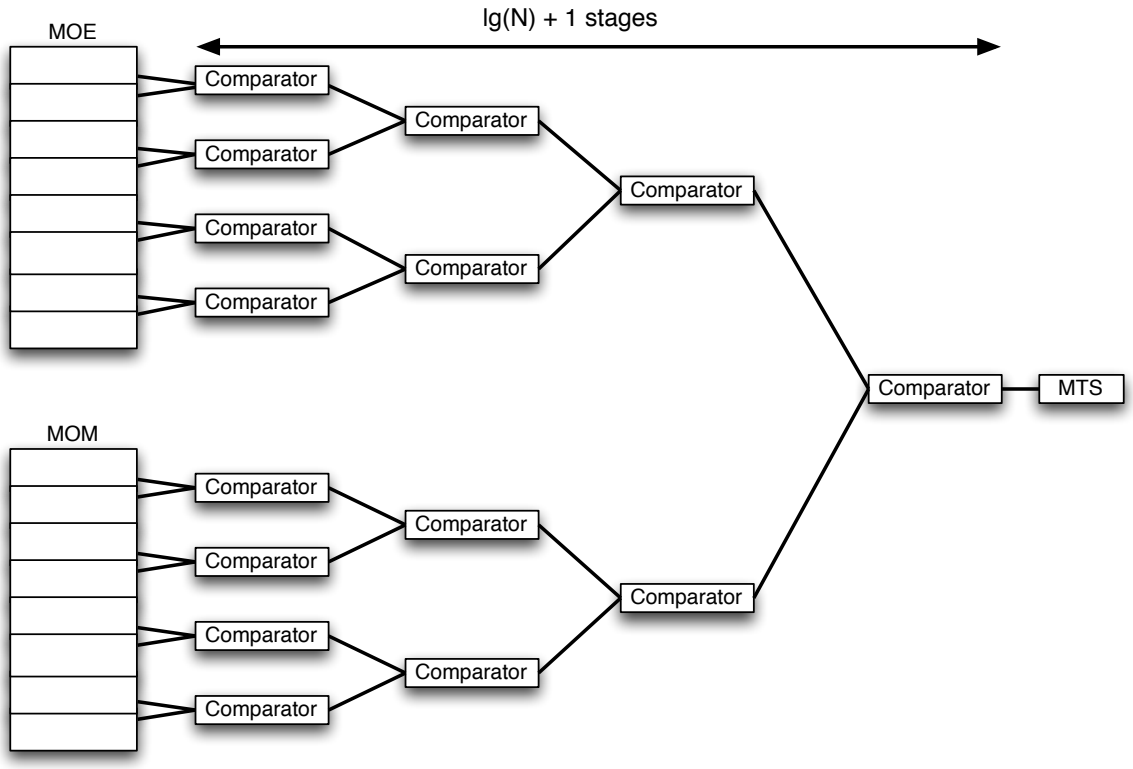


Figure 14: The tree of comparators used to find the minimum timestamp in both the Minimum Outstanding Event(MOE) and Minimum Outstanding Message(MOM) register files.

Each register file contains N values, one for each core. In addition, there are $N-1$ 32- or 64-bit comparators for both the *MOE* and *MOM* files that compute the minimum value in both files in $lg(N)+1$ stages, as seen in Figure 14.

3.1.2 Atomic Instructions

In addition to the three register files, there are five atomic instructions required to access the Global Synchronization Unit. These five instructions are *Minimum Timestamp*(*MTS*), *Increment*(*Inc*), *Decrement*(*Dec*), *Write If Less Than*(*WILT*), and *Write If Zero*(*WIZ*). A description of these atomic instructions follows.

- Minimum Timestamp (MTS) - Returns the minimum of all values in the MOM and MOE
- Increment (Inc) - Increments the TMC

- Decrement (Dec) - Decrements the TMC
- Write If Less Than (WILT) - Writes the value into the MOM if it is less than the value already in it
- Write If Zero (WIZ) - Writes the value into the MOM or MOE if the TMC has a value of zero

3.1.3 Simulator Loop

This section contains an example of how the instructions for the Global Synchronization Unit would be integrated into the main loop of a parallel discrete event simulator.

- If the earliest event in the event queue is safe to process (event time $\leq$ MTS() + lookahead)
  - Remove event from queue
  - Update local state
  - Send any messages generated by the event
    - Inc() receiver's TMC
    - WILT(timestamp of msg) in receiver's MOM
    - Send Message
    - Repeat until all messages are sent
    - WIZ(timestamp of earliest event in queue) into MOE
  - If there are pending messages
    - Put message in event queue
    - Dec() TMC
    - Repeat until TMC=0

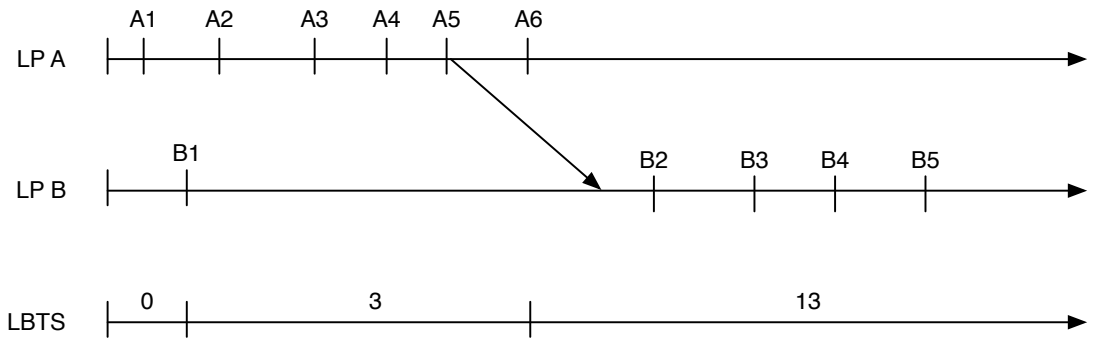


Figure 15: Timeline for the example

* WIZ(timestamp of earliest event in queue) into MOE
* WIZ(∞) into MOM

3.2 GSU Example

To illustrate the operation of our approach, we now present a simple example. This example will demonstrate how the approach accounts for transient messages and ensures the correctness of the *LBTS* computation at any point in the process. We will use two *LPs* for simplicity. Nevertheless, our approach is easily extended to k *LPs*. The state of the *GSU* at each step in the example can be seen in Figure 16, and the state of each *LP's* event queue can be seen in Figure 17. A timeline of the *LPs'* actions and the value of an *LBTS* computation at that time can be seen in Figure 15.

Example:

All *LPs* start at simulation time=0 with lookahead=10. Therefore, the initial LBTS is 0, and each *LP's* safe event window is equal to 10.

A1) LP_A populates its *MOE* register with the timestamp of the first event in its event queue, which is 3.

B1) LP_B populates its *MOE* register with the timestamp of the first event in its event queue, which is 19. At this point the *LBTS*, if calculated, would correctly return a value of 3. Prior to this point the *LBTS* would be 0 because MOE[B]

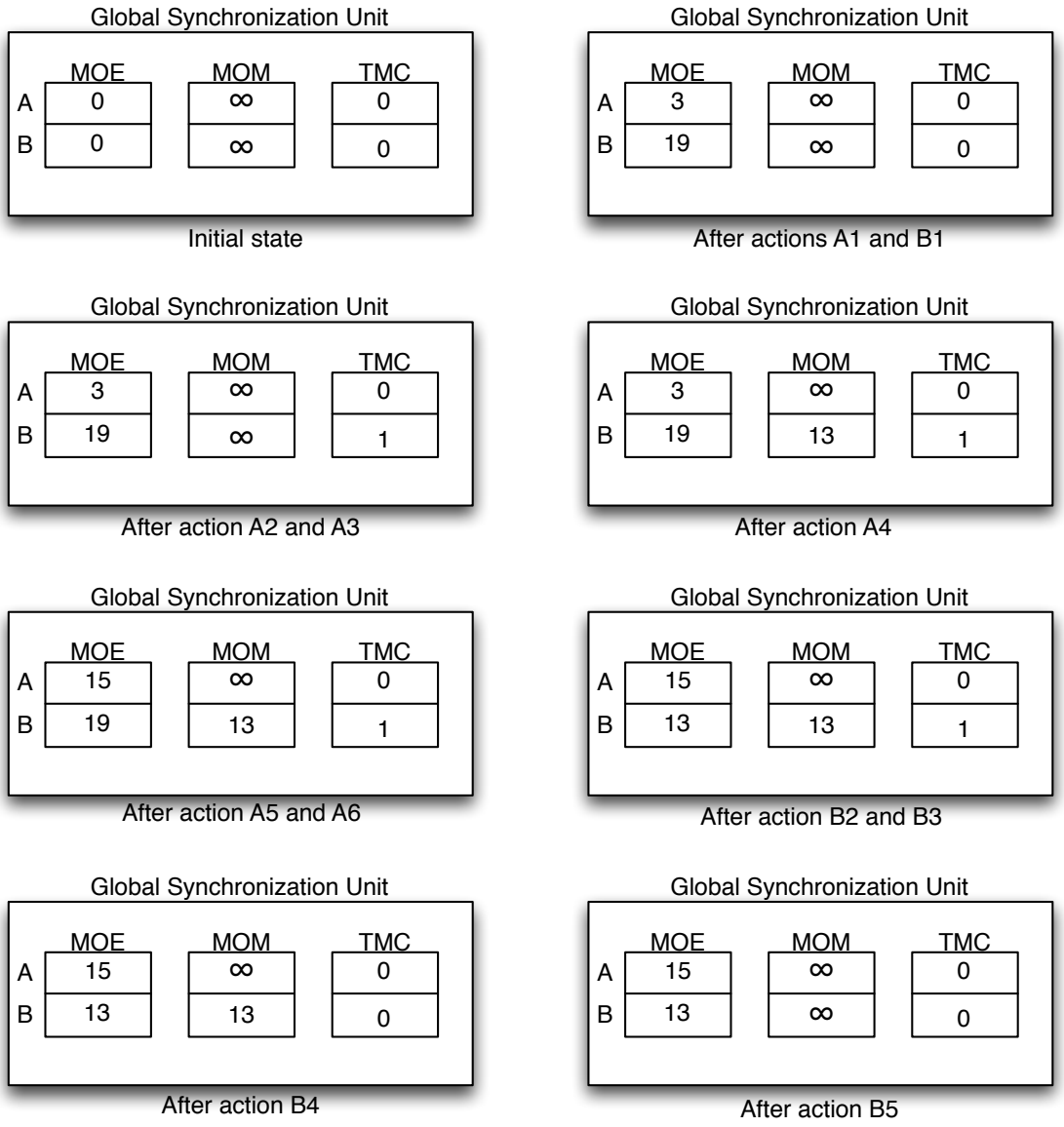


Figure 16: Global Synchronization Unit state for the example

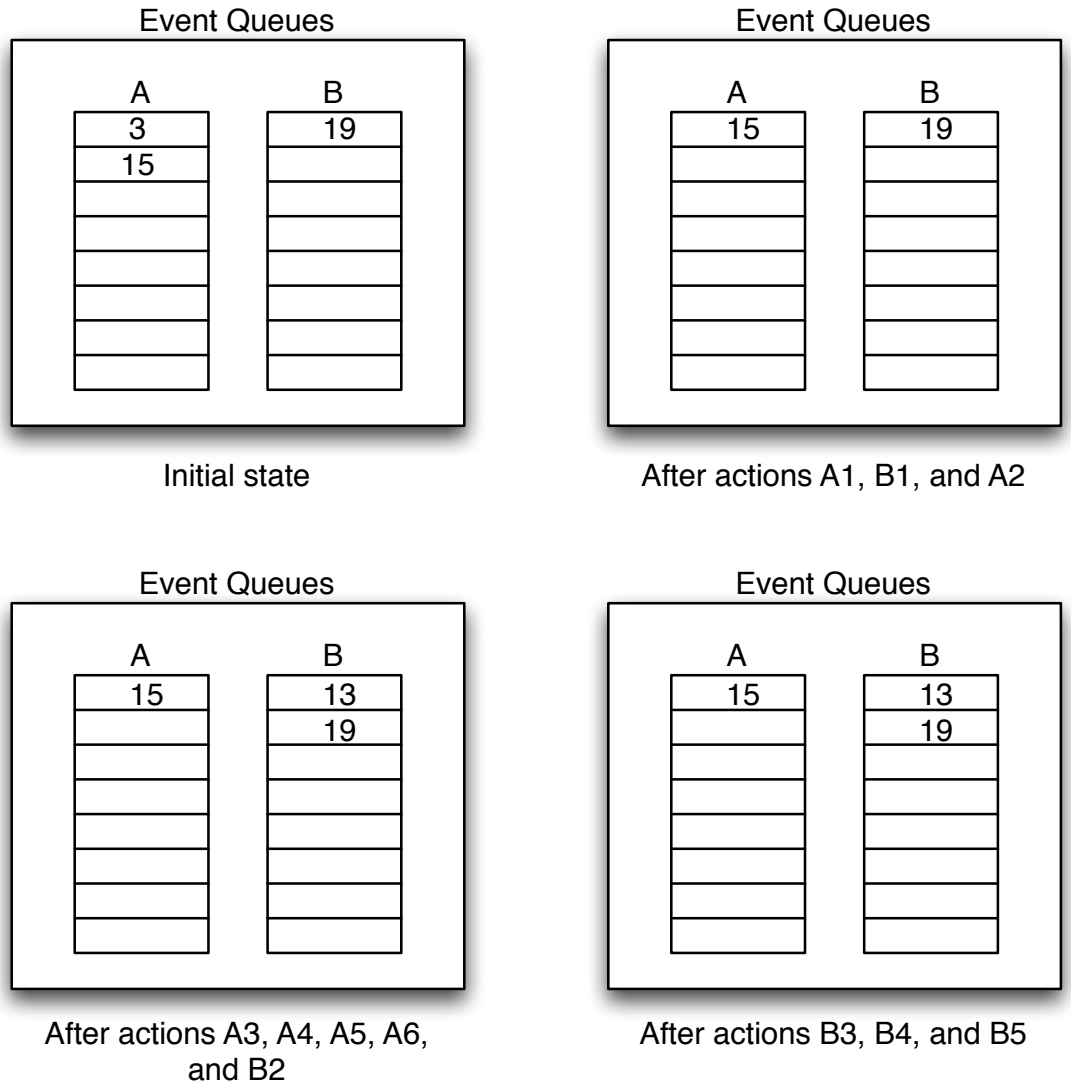


Figure 17: Event queue state for the example

still contained its initial value, 0.

A2) LP_A removes the first event off of its event queue and processes it. This event generates an event with timestamp=13 whose destination is LP_B.

A3) LP_A atomically increments the *TMC* register of LP_B.

A4) LP_A uses the *atomic write if less than* function to put the timestamp of the message in the *MOM* register of LP_B.

A5) LP_A sends the message to LP_B.

A6) LP_A updates its *MOE* register to the timestamp of the next event in its event queue, which is 15. At this point the *LBTS*, if calculated, would correctly return a value of 13, because the transient message is accounted for in the *MOM*.

B2) LP_B receives the message and puts it in its event queue.

B3) LP_B updates its *MOE* register to the timestamp of the next event in its event queue, which is 13.

B4) LP_B atomically decrements its *TMC* register.

B5) LP_B uses the atomic write if zero function to write a value of infinity in its *MOM* register.

3.3 Proof of Correctness

This section contains a proof of correctness for the Global Synchronization Unit. It begins by stating the assumptions used in the proof, followed by the algorithms used to process events using the *GSU*, and concluding with the proof itself.

3.3.1 Statements of Fact

1. MTS()=min(min(MOE), min(MOM))

2. Lookahead - The minimum simulation time between the current time at the sending LP and the timestamp of the message being sent.

3. Corollary of Lookahead - For any LP with simulation time T, T $\leq$ MTS()+lookahead.

4. Causality Requirement - For any message with timestamp t_1 and a receiving LP with simulation time T, $t_1 \geq T$

5. Corollary of Causality Requirement - For any message with timestamp t_1 and sending LP with simulation time T at the time the message is sent, $t_1 \geq T + lookahead$

6. Non-negative Timestamps - For any timestamp t_1, $t_1 \geq 0$

7. Corollary of Non-negative Timestamps - For any set of timestamps $t_1...t_k$, $min(t_1...t_k) \geq 0$

8. Correctness requirement - $MTS() \leq$ actual minimum timestamp of simulation at any given point. If the sequence of values of MTS() are not monotonically non-decreasing, the correctness requirement has been violated.

3.3.2 Event Processing Algorithms

There are four different algorithms required to process events using the *GSU*. The first is the initialization of the *GSU*, performed before the simulation begins. The second algorithm is used when actually processing events in the event queue. The third is used when sending messages generated by events when they are processed. The fourth algorithm is used to receive messages sent by other logical processes.

1. Initialization

   - Before the simulation begins, every register in the MOE and TMC is set to zero and every register in the MOM is set to infinity.

- When the simulation begins, each LP initializes its MOE register to the timestamp of the earliest event in its event queue before entering the simulation loop for the first time.

2. To Process an Event

   - If timestamp of earliest event in queue is $\leq$ MTS() + lookahead
     - Remove event from the event queue
     - Update local state
     - Send any messages generated
     - Update MOE to next unprocessed event time

3. To Send Messages

   - Increment TMC of the receiver
   - Update MOM of the receiver using atomic Write If Less Than
   - Send message
   - Repeat until all messages are sent.

4. To Receive Messages

   - Read message from input queue and place in the event queue
   - Decrement LP's own TMC
   - Repeat until TMC=0
   - Update MOE using atomic Write If Zero
   - Write ∞ in MOM using atomic Write If Zero

3.3.3 Proof

3.3.3.1 Initialization

- At the beginning of the simulation, each MOE register contains a value of zero. According to the definition of MTS() in (A1), $MTS() = 0$. Following from (A7), $MTS() \leq$ actual minimum timestamp, satisfying the Correctness Requirement, (A8).

- When some, but not all, LPs have written the timestamp of their earliest event into the MOE, some MOE registers will still contain a value of zero. Following from (A7) and (A1), $MTS() = 0$. Following from (A7), $MTS() \leq$ actual minimum timestamp, satisfying the Correctness Requirement, (A8).

- Once each LP has written the timestamp of their earliest event into the MOE, following from (A1), $MTS()$ = min(min(all events on LP_j), min(∞)) for j=1 to N, where N is the number of LPs. As $MTS()$ =min(all timestamps on each LP), $MTS() \leq$ actual minimum timestamp, satisfying the Correctness Requirement, (A8).

3.3.3.2 Process an Event and Send Messages

- Check if $t \leq$ (MTS() + lookahead), where t is the timestamp of the earliest event in the queue. This satisfies the Corollary of Lookahead, (A3), indicating that the event is safe to process.

  - If the earliest event is safe, the event is processed, and the local simulation time is updated. It may generate messages to send to other LPs. These messages must have a timestamp t_j such that $t_j \geq t + lookahead$, as specified by (A5). The correctness requirement, (A8), is still satisfied, as the MOE contains t and $t < t_j$ for all j, as follows from (A5).

– The sending LP calls the atomic Increment function for the receiving LP's TMC.

  * If there is no contention, the TMC for the receiving LP is atomically incremented, ensuring that the MOM will not be overwritten with infinity by the receiver.

  * If there is contention for the TMC register, another LP is also preparing to send a message to the same receiving LP or the receiving LP is writing infinity into its MOM. The Increment completes after the other LP has also incremented the TMC or infinity has been written into the MOM. The atomicity of the instruction ensures that the TMC gets incremented the correct number of times.

  As no changes have been made to the MOE or MOM registers, the MTS() value remains the same, satisfying the Correctness Requirement, (A8).

– Then, the sending LP calls the atomic Write If Less Than function for the receiving LP's MOM with the timestamp of the message being sent, t_j.

  * If there is no contention and $t_j <$ the current value of the receiving LP's MOM, the timestamp is written into the receiving LP's MOM. Following from (A1), $MTS() \leq t_j$, satisfying the Correctness Requirement, (A8).

  * If there is no contention and $t_j \geq$ the current value in the receiving LP's MOM, then the Correctness Requirement (A8) is still satisfied, as $MTS() \leq$ the value present in the MOM, which we also know is less than the timestamp of the message being sent.

  * If there is contention for the TMC or MOM registers, then another LP is also preparing to send a message to the same receiving LP, or the receiving LP is attempting to write infinity into its MOM using

atomic Write If Zero.

- If the receiving LP was attempting to write infinity into its MOM using atomic Write If Zero, it would not have succeeded, as the TMC has a non-zero value. As no changes have been made to the MOE or MOM registers, the MTS() value remains the same, satisfying the Correctness Requirement, (A8).
- If the other LP was attempting to write the timestamp of its message into the receiving LP's MOM, it either wrote the value in, or the value in the MOM was smaller than the timestamp. When the sending LP's atomic Write If Less Than call completes, either $t_j <$ the current value in the receiving LP's MOM and gets written in, or $t_j \geq$ value in the receiving LP's MOM and t_j is not written in. If a smaller value is already present in the MOM, then the Correctness Requirement (A8) is still satisfied, as the value returned by $MTS() \leq$ the value present in the MOM, which we also know is less than the timestamp of the message being sent. Also, a larger value cannot be written into the receiving LP's MOM until all pending messages have been read from the input queue, put into the event queue, and the timestamp of the earliest event in the queue, which is $\leq t_j$, is written into the MOE.

– After all the messages are sent, the sending LP calls the atomic Write If Zero function for its MOE with the timestamp of the earliest event in the queue, t'.

  * If the TMC has a value of zero, the MOE is updated to t', allowing the simulation to move forward
  * If the TMC does not have a value of zero or there is contention, the LP has pending messages which need to be processed.

- If $t > (MTS() + lookahead)$, no changes are made to the MOE or MOM, and the Correctness Requirement, (A8), is satisfied.

3.3.3.3 Receiving a Message

- When an LP has pending messages, its MOM contains the minimum timestamp of all its unprocessed messages, $min(t_j)$.

- The LP receives a message, puts it into the event queue, and calls the atomic Decrement function on its TMC.

  - If there is no contention, the LP's TMC is decremented.

  - If there is contention, another LP is sending a message to this LP. The atomicity of the instruction ensures that the TMC gets decremented the correct number of times. This continues until the TMC has a value of zero, indicating that there are no more outstanding messages.

  As no changes have been made to the MOE or MOM registers, the MTS() value remains the same, satisfying the Correctness Requirement, (A8).

- The LP then calls the atomic Write If Zero instruction for its MOE with the timestamp of the earliest event in the queue, t. As follows from (A4), $t \leq min(t_j)$.

  - If the TMC has a value of zero, the MOE will contain the timestamp of the earliest event in the queue. As all the messages are now in the event queue, this time must be less than or equal to the timestamps of all the messages received, preserving the Correctness Requirement, (A8).

  - If the TMC does not have a value of zero or there is contention, another message has been posted for this LP. The LP will continue to process incoming messages as above until the TMC value is again zero. As no

changes have been made to the MOE or MOM registers, the MTS() value remains the same, satisfying the Correctness Requirement, (A8).

- Finally, the LP calls the atomic Write If Zero function for its MOM with ∞.
  - If the TMC has a value of zero, the MOM will contain a value of ∞, allowing the simulation to progress. The value in the MOE is now less than or equal to the value that was previously in the MOM, per the last step, preserving the correctness requirement (8).
  - If the TMC does not have a value of zero or there is contention, another message has been posted for this LP. The LP will continue to process incoming messages as above until the TMC value is again zero. As no changes have been made to the MOE or MOM registers, the MTS() value remains the same, satisfying the Correctness Requirement, (A8).

3.3.3.4 Deadlock-free

- Given a set of N timestamps, one or more must be the minimum, and, therefore, safe to process, preventing deadlock.
- It is impossible for an LP to keep receiving messages indefinitely, preventing it from writing infinity in its MOM, as the sending LPs must eventually run out of safe events to process.

3.4 Experimental Setup

The Global Synchronization Unit has not yet been designed at a gate-level and fabricated. Due to this limitation, we cannot measure its performance in a real system, but must instead simulate the hardware to predict its performance when used by a parallel simulation. We have used several simulation tools to conduct our performance analyses, Simics, RandomSim, and the Georgia Tech Network Simulator (GTNetS).

3.4.1 Simics

For our experiments, we implemented a model of our GSU in Simics [12, 1]. Simics is an instruction-set level, full-system, discrete event simulator. It can model several ISAs, including x86, PowerPC, UltraSparc, and MIPS. It is also capable of simulating multi-core or multi-processor machines and networks of machines. In addition, Simics is capable of running full, unaltered operating systems and executables. Simics also allows users to make tradeoffs between accuracy and efficiency by providing operating modes at several granularities. The default, *Normal Mode*, provides an emulation-like simulation, with each instruction executed in order and assumed to take one cycle. Additionally, *Normal Mode* assumes that any memory and I/O operations take zero time. When running in *Memory timing mode*, Simics interfaces with user-supplied memory timing models. Finally, the *Micro-architectural Interface* allows Out-of-Order execution and interfacing with processor timing models. The Simics framework is highly extensible. An API is provided allowing for user modules to interface with the simulator. These modules can be models of devices, or even detailed processor and memory timing models. An example of an extensive user-developed set of modules for Simics is the GEMS project [13].

3.4.2 RandomSim

For our first performance benchmark we used *RandomSim*, a simple multi-process distributed simulation we developed. The RandomSim program is a simple distributed simulation, with an arbitrary number of LPs. Each LP generates an arbitrarily chosen small number of initial events and schedules those in a sorted pending event queue. It then enters the main event processing loop and removes and processes safe events. When processing each event, each LP randomly chooses a number of new events to schedule. This number is chosen randomly from an exponential distribution with a mean of 1.05. For each new event, a random destination LP (which possibly can

be itself) is chosen from a uniform distribution. Then a random future time for the event is chosen from a uniform distribution between 0 and 5 seconds. The current simulation time and the lookahead value are added to the chosen timestamp, which ensures that event will satisfy lookahead constraints. The event is then sent to the target destination for later processing. Finally, to prevent an unbounded growth in the total number of events, the pending event list for each LP is capped at 500. For these simulations, the lookahead was set to 2 seconds and the simulation runs for 100,000 seconds of simulation time.

3.4.3 The Georgia Tech Network Simulator

For the sensitivity analysis, we adapted the Georgia Tech Network Simulator to use our GSU for synchronization. The Georgia Tech Network Simulator(GTNetS) is a discrete event network simulator that was designed with scalable parallel simulation in mind. GTNetS includes models for both wired and wireless network protocols at the application, transport, and link levels, as well as stock objects for creating common topologies. GTNetS is also highly extensible, allowing users to write new models as C++ objects that inherit from the base class of that network element, such as *Queue*, *Node*, or *Link*. GTNetS provides network traces at user-specified levels of granularity, and also includes built-in data collection to gather network statistics. GTNetS includes several features to improve scalability. These include NIxVector routing, which eliminates the need for routing tables, instead calculating routes on demand [20]. Another optimization is the use of *Ghost Nodes*. In parallel simulations, the logical process maintains the full state of the nodes it is responsible for simulating, but a reduced state representation, or ghost, of the nodes assigned to other LPs. This gives every LP a representation of the entire topology, allowing it to make routing decisions without having to maintain the full state of the simulation on each LP [19]. GTNetS has been scaled to hundreds of thousands of simulated nodes [18].

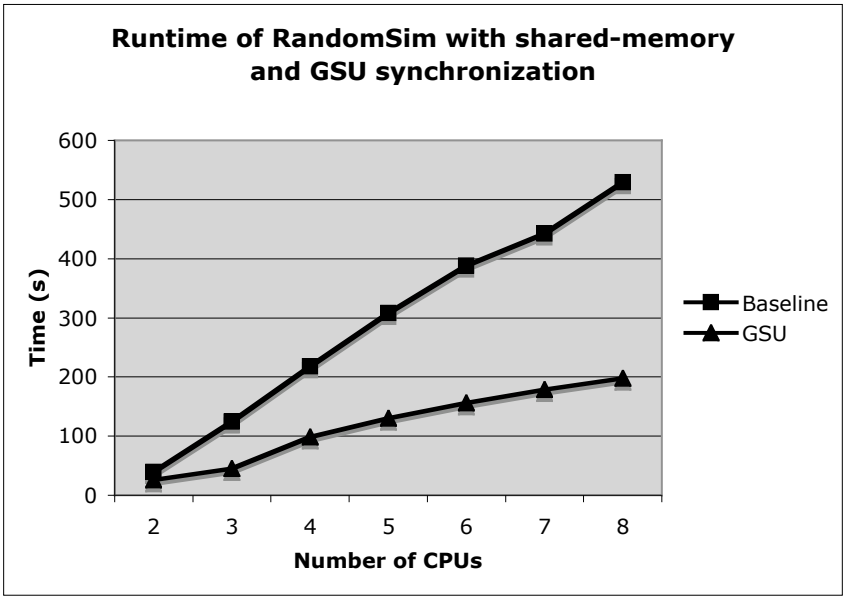


Figure 18: The runtime of RandomSim with and without the GSU for synchronization.

3.5 Results

In order to assess the performance of the Global Synchronization Unit, we conducted experiments in Simics using the GSU model to perform the synchronization in two different discrete event simulators, RandomSim and GTNetS. The results from these experiments are described below.

3.5.1 RandomSim

Figure 18 shows the runtimes of the baseline and GSU versions of RandomSim on our simulated x86 system. Using the GSU, not only is the total runtime significantly less than that of the baseline version, but the rate of growth is less than half that of the baseline.

3.5.2 Sensitivity Analysis

As we have not designed our hardware at a gate-level, we have had to estimate the number of cycles required to access the GSU with our new atomic instructions. With

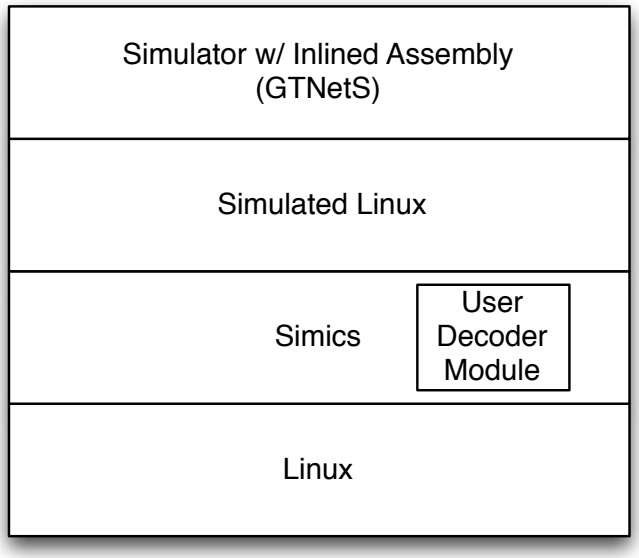


Figure 19: The software stack for the sensitivity analysis

the sensitivity analysis, we are determining the impact of GSU access times on the runtime of a low-lookahead network simulation and comparing it with a baseline version of the simulator using a shared-memory barrier synchronization algorithm. The baseline synchronization algorithm uses a master/slave approach where LP zero is the master process. When LP zero enters the synchronization barrier, it immediately begins checking for the LBTS information reported by other LPs and calculating the minimum of those that have been reported. The synchronization is completed when all LPs have entered the barrier and the master has computed the minimum of all the reported values. To measure the runtime, we ran the network simulator, GTNetS on top of Fedora Core 5 running in Simics, as seen in Figure 19. Using Simics, we modeled a multi-core system of 2 to 32 cores. The GSU is modeled by using unused x86 opcodes for the atomic instructions. We then wrote a module in Simics that decodes the new instructions, executes them, contains the state of the GSU, and introduces the appropriate delays for each instruction. In our implementation of GTNetS using the GSU, we used inlined assembly calls in C code to call the new atomic instructions.

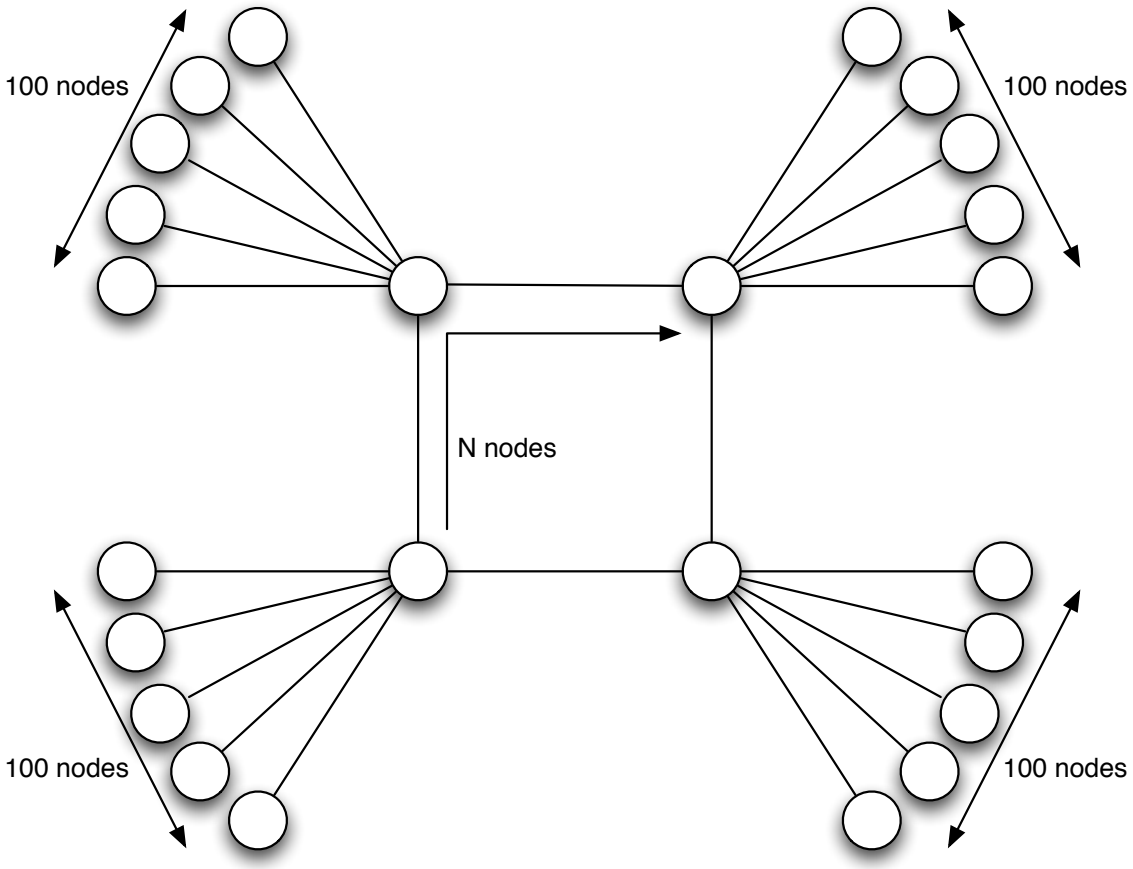


Figure 20: The star topology used in the GTNetS simulation

The network simulation we ran in GTNetS for both the baseline and GSU versions is a star topology with 100 leaf nodes on each logical process, as seen in Figure 20. The link delay for each link in the simulation is $1\mu s$. This translates to a uniform lookahead of $1\mu s$ for the simulation. Located on each leaf node is a TCP application which sends packets to a leaf node on another logical process. The duration of the simulation is 10 seconds of simulation time. The problem size of the simulation scales up linearly with the number of LPs, so the ideal speedup would correspond to a constant runtime. For the sensitivity analysis we started with GSU access times of 2 cycles for calculating the *Minimum Timestamp*, 3 cycles for *atomic Increment/Decrement*, 4 cycles for *atomic Write If Less Than*, and 5 cycles for *atomic Write If Zero*. We then ran the simulation with access times of 10x, 100x, 150x, 250x and 500x the original values.

As seen in Figure 21, the version of GTNetS using the GSU for synchronization had runtimes about 40 percent less than the version using traditional shared memory synchronization for up to 32 CPUs. In addition, there was virtually no impact on the runtime with access times of 10x, 100x, and 150x the original values. It was not until the access times were increased to 500 or more cycles that the runtime increased to times greater than the baseline shared-memory synchronization version. This indicates that any likely values for the access times of the GSU will give us significant performance improvements over a shared-memory implementation.

In the sensitivity analysis in Figure 21, we see that not only does the runtime suddenly increase between the 150x delays and the 250x delays, but the shape of the curve also changes from linear to exponential. We believe that this is because the GSU instructions with smaller delays can be overlapped with message passing and event processing by the other LPs, making the effect of the GSU instruction delays negligible. The time spent waiting on GSU instructions would otherwise have been spent blocking, waiting for more events to be safe to process. This means that for delays of 150x or less, the shape of the curve is determined by the message passing and event execution. However, when the delays are increased to 250x, they begin to dominate over the message passing and event processing costs. The LPs are now waiting longer for the GSU instructions to complete than they would have naturally been blocking due to the limits of parallelism in the simulation.

In these experiments, we assumed that each memory operation takes zero time, which is, of course, not realistic, but it results in a considerable decrease in the overall execution of our lengthy simulation-based experiments. This seems a reasonable assumption because regardless of the actual memory delays, we expect that both the baseline shared memory approach and the GSU approach would be affected equally, other than the memory costs of synchronization. During the synchronization portions of the simulation, the shared memory version will incur the cost of memory delays,

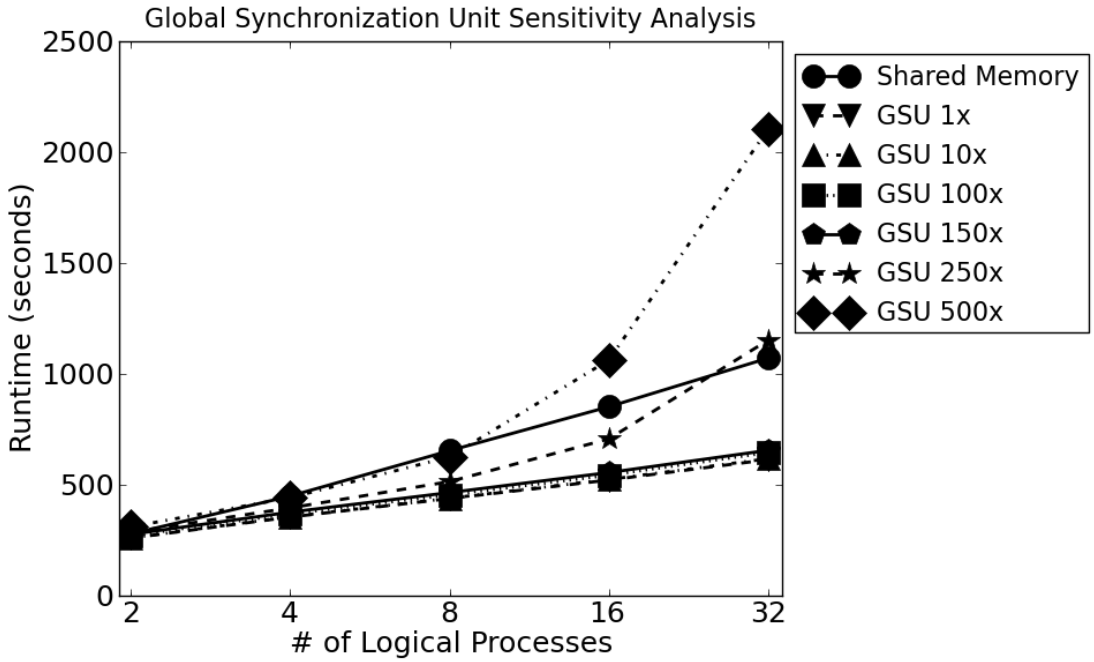


Figure 21: A graph of runtime vs # of CPUs for the baseline shared memory synchronization and the GSU version with GSU access delays of 1x, 10x, 100x, 150x, 250x and 500x the original delays

while the GSU approach does not. In order to demonstrate that this assumption does not significantly impact our performance comparison, we ran a single test case using a delay of 20 cycles for each memory operation. We used the same GTNetS simulation as in the sensitivity analysis, however it only ran for one second of simulation time instead of the ten seconds used previously. We ran this GTNetS case on 4 cores, comparing the shared memory baseline version to the GSU with the original delays. We found that with the memory delays added, the runtime of the shared memory version was 82 minutes, and the runtime of the GSU version was 65 minutes. This gives us a 20% reduction in runtime with the GSU version. In the previous experiment with zero-cost memory operations, we saw a runtime reduction of 21% using the GSU. The small difference may be explained by the fact that the initialization of the simulation is a greater percentage of the runtime for a one second simulation than it is for the ten second version. This initialization has a fixed cost that is unaffected by

the time synchronization approach.

3.6 An Estimate of Time Delays for the Global Synchronization Unit

Although the GSU has not been designed at a gate level, we have based our estimates for the instruction delays off of the functional-level design. For the Minimum Timestamp instruction, if we assume that each comparison takes one cycle, the number of delay cycles is equal to the depth of the tree of comparators, $lg(N) + 1$. For the Increment and Decrement Instructions, we perform one 32- or 64-bit add, taking one cycle. For both the Write If Less Than and Write If Zero instructions, we perform one comparison, taking one cycle, and one 32- or 64-bit write, taking one cycle, giving us a delay of two cycles for each instruction.

3.7 An Estimate of the Area for the Global Synchronization Unit

In order to estimate the area that our Global Synchronization Unit would take on-die, we used CACTI 5.3, a tool designed to provide power and area estimates for cache and memory configurations. For our estimates, we assumed a system of 1024 cores, to give us an upper bound on the area that would be used. This means that the GSU consists of 3 register files, with 1024 entries each, and each entry is 64-bits wide. We also assume that that the area for the logic is negligible compared to the area used by the registers. Based on these assumptions, we used CACTI to estimate the size of 3 caches in 65nm technology, one for each of the register files in the GSU, each with 1024 8-byte lines, direct associativity, and a single bank. Using these parameters, one of the register files will have an area of $.146mm^2$ and the entire GSU will have an area of $.439mm^2$. As a comparison, the Intel Pentium Dual-Core, "Conroe", has a die-area of $143mm^2$ in 65nm technology. This means the GSU would take up about 0.3% of the CPU area.

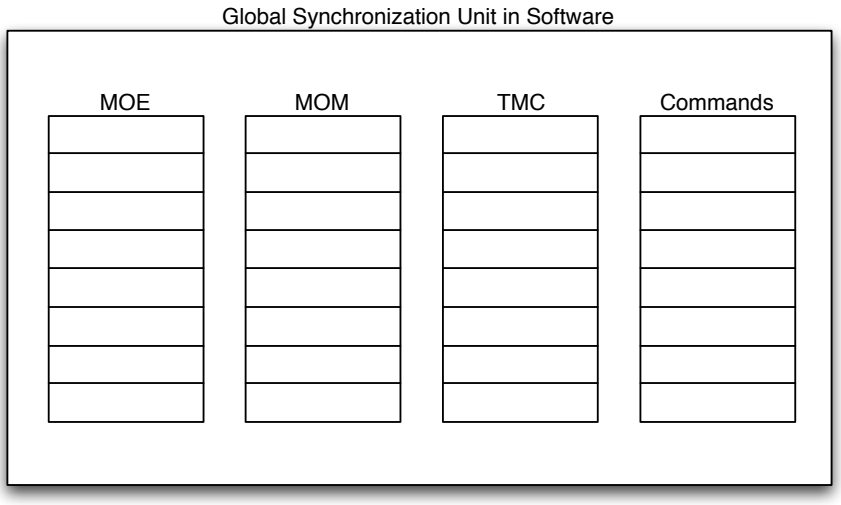


Figure 22: The data structures for the Global Synchronization Unit as implemented in software.

3.8 The Global Synchronization Unit in Software

For circumstances where the cost of this specialized hardware would be prohibitive, or to use this approach on pre-existing hardware, the Global Synchronization Unit can be implemented in software, as a separate thread or process. A description of this approach follows.

The key to implementing the GSU in software is maintaining the atomicity of the GSU instructions without introducing any locks that could result in contention. This can be achieved by creating an array of structures with an entry for each logical process, into which the LP writes calls to the GSU, as seen in Figure 22. In addition, when the GSU has executed the instruction, it writes any return values into the structure and changes the instruction type of the structure to REPLY. The GSU thread or process services these calls in a round robin manner.

3.8.1 Algorithm for the Global Synchronization Unit in Software

In this section, we describe the algorithm used in the software implementation of the Global Synchronization Unit. The algorithm that follows would run in a separate thread or process for the duration of the simulation and would be accessed by the

- While the simulation is not complete
  - For each core
    - Check the command field
    - If command is REPLY
      - Skip
    - If command is Increment or Decrement
      - Increment or decrement the specified TMC register
      - Write the current value of the TMC register into the return field
      - Change the command to REPLY
    - If command is Write if Less Than
      - If the new value is less than the current value in the specified MOM register, write the new value into the specified MOM register
      - Change the command to REPLY
    - If command is Write if Zero
      - If the specified TMC register contains zero, write the new value into the corresponding MOE or MOM
      - Change the command to REPLY
    - If command is Minimum Timestamp
      - Write the min(min(MOE), min(MOM)) into the return field
      - Change the command to REPLY

logical processes for use in synchronization.

3.9 Non-uniform Lookahead and Global Virtual Time

Non-uniform lookahead results when the properties of the connections between the physical systems being modeled are irregular. An example of this would be an irregular network topology. An advantage of supporting non-uniform lookahead is that those *LPs* with greater lookahead can take advantage of this property and process events further into the future before synchronizing. However, the complications of supporting non-uniform lookahead often outweigh the benefits, and uniform lookahead is approximated by assuming that the smallest lookahead in the simulation is the lookahead for each *LP*.

Our approach can be extended to simulations with non-uniform lookahead with

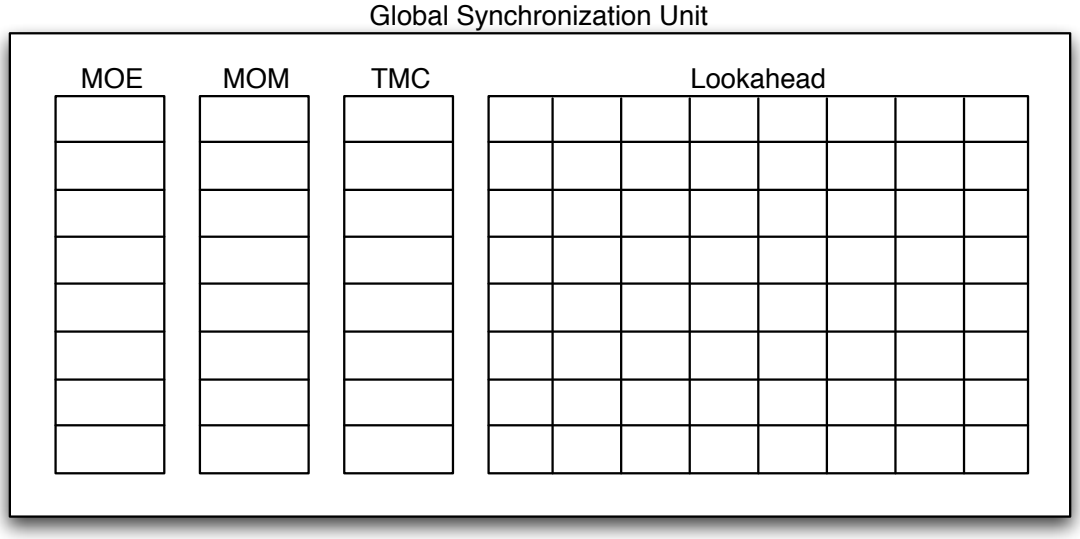


Figure 23: The components of a global synchronization unit with non-uniform lookahead

the addition of an NxN register file containing the pairwise lookahead values of the *LPs*, as seen in Figure 23. Non-uniform lookahead must be accounted for in the *GSU* instead of at the *LP* as it is when lookahead is uniform. This is because it is insufficient to calculate the *MTS* and then add the appropriate lookahead. The safe event window is bounded by the minimum of the sums of the *LP's* smallest event timestamps (the minimum of the values in the MOE and MOM registers) and their associated lookahead. Therefore, when calculating the *LBTS* for LP_i, the *GSU* will have to compute the $min(min(MOE_j, MOM_j) + lookahead_{i,j})$ for $j = 0$ to N. Our design will again accomplish this with $lg(N)$ stages of comparators.

While the intended application for this approach is conservative synchronization, it is also applicable to optimistic synchronization. The calculation of *global virtual time* (*GVT*) in the Time Warp algorithm is conceptually similar to an *LBTS* calculation without any lookahead. An approach to *GVT* calculation in shared-memory has been presented in [5]. However the computation is only completed with the participation of all *LPs*, unlike our *GSU*, which can return an *MTS* on demand.

CHAPTER IV

HARDWARE SUPPORT FOR MESSAGE PASSING

A common difficulty for distributed applications running in separate address spaces in a multi-CPU processing environment is management of messages to be passed from one process to another. Each process has its own local view of the physical address space, and manages its own memory allocation and deallocation. When the need arises to pass information from one process to another, typically called a *message*, existing solutions involve expensive copying of message contents into a *shared memory region*, along with interlocking mechanisms to insure simultaneous updates do not occur. Depending on the size of the messages and the frequency of message exchanges, this process can and often does become the primary limiting factor in the overall application performance.

In order to address the overhead of parallel discrete event simulation incurred by message passing, we have designed two complementary hardware units, an *Atomic Message Passing* hardware unit and an *Atomic Shared Heap* hardware unit, at a functional level. When used together, these units can be used to implement a lock-free, zero-copy message passing system for a multi-core system. The solution approach must solve two independent but related problems. First, the messages contents must be stored in a memory area directly addressable by both the message sender and message receiver. While the commonly used *shared memory* approach solves this problem partially (the sending and receiving process do not necessarily have a common view of the *virtual address* of the shared memory region), a second and more difficult problem must be addressed. That is, when messages are passed from one process to one or

more other processes, it becomes problematic for the memory allocation and deallocation mechanisms to know exactly when all references to a given allocated memory area are no longer needed. In our novel design, the AMP unit handles the lock-free passing of message metadata (not complete messages) in an efficient manner, and our ASH unit handles the actual memory allocation and required reference counting approach to allow memory re-use. We demonstrate that this design leads to as much as a 16% improvement in overall execution time for distributed message-passing applications. Further, we show that the overall die area required for our design is minimal, excepting in extreme cases of a large number of cores on a single die.

4.1 Atomic Message Passing

The design for our Atomic Message Passing(AMP) unit requires a circular queue of size k for each of N logical processes(LPs). Each of these circular queues contains pointers to messages destined for a single LP. In addition, there is a register file which contains the input and output offsets for each LP's circular queue. The pointers to messages can either point to the sender's heap or to a shared heap, depending on the implementation. Our design also includes an atomic Write and an atomic Read instruction to access the AMP.

4.1.1 Atomic Instructions

1. Read
    - If the queue is empty ($In = Out$)
        - Return immediately with an indication of failure.
    - Else
        - Read the pointer in the Out slot.
        - Increment the Out offset ($Out = (Out + 1)\%k$).
2. Write

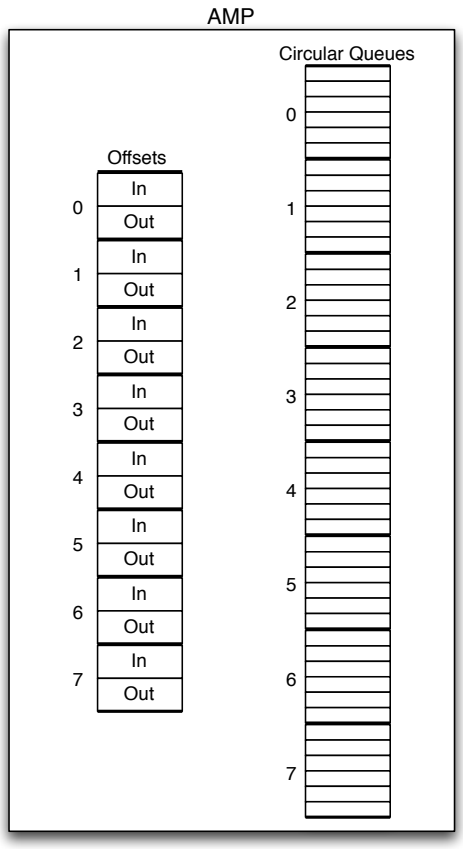


Figure 24: The Atomic Message Passing unit

- If the queue is full ($In = (Out - 1)\%k$)
  - Return immediately with an indication of failure.
- Else
  - Write the pointer to the message in the In slot.
  - Increment the In offset ($In = (In + 1)\%k$).

Although this unit has only been designed at a functional level, the delay for each instruction has been estimated to be 2 cycles. The Write instruction consists of writing a 32-bit value in one register and incrementing another register. Each of these operations should take 1 cycle, giving a total of two cycles for the instruction. Likewise, the Read instruction consists of reading a 32-bit value from a register and incrementing another register. Again, these operations each take 1 cycle, giving a total of two cycles for the instruction.

In order to estimate the area used by the Atomic Message Passing unit on-die, we used CACTI 5.3. In order to get an upper bound on the area used by the device, we assumed a system of 1024 cores. There is one circular queue for each core, and each circular queue has 1024 entries, so it can contain a message from each of its peers at any time. Each entry is 64-bits wide, in order to accommodate a 64-bit address system. In addition to the circular queues, themselves, there are also 1024 In and Out variables, each 64-bits wide. Like the GSU, we assume that the area for the logic is negligible compared to the area for the registers. Based on all of these assumptions, we used CACTI to estimate the size of two caches in 65 nm technology. The first contains the circular queues, and contains 1024x1024 8-byte lines, with direct associativity and a single bank. It is estimated to have an area of $92.436mm^2$. The second cache is the In and Out variables for each of the 1024 circular queues, and contains 1024x2 8-byte lines, with direct associativity and a single bank. This cache is estimated to have an area of $.270mm^2$. This gives us a total area for the AMP of $92.706mm^2$. Comparing

Table 1: Areas On-Die for the Atomic Message Passing Unit

Cores	Circular Queue Area (mm^2)	In/Out Area (mm^2)	Total Area (mm^2)	Percentage of CPU Area
32	.146	.028	.174	0.1
64	.470	.046	.516	0.4
128	1.937	.067	2.004	1.4
256	6.391	.110	6.501	4.5
512	22.404	.146	22.550	15.8
1024	92.436	.270	92.706	64.8

this to the area of the Intel Pentium Dual-Core,"Conroe", we see that an AMP this size would take about 65% of the entire CPU area of $143mm^2$. However, it is clear that 1024 cores could not fit in the same area using existing technology. We have included area estimates for the Atomic Message Passing unit with core counts from 32 to 1024 in Table 1. For each of these estimates, we used circular queues with a number of entries equal to the number for cores. We can see that the area for an AMP with 32 cores would be $.174mm^2$, about 0.1% of the total area for the "Conroe" CPU.

4.2 Atomic Shared Heap

The design for our Atomic Shared Heap(ASH) unit requires one register for each allocatable unit of the heap. For a 512MB heap with a page size of 16KB, the ASH would contain $512MB/16KB = 2^{15}$ entries. For a system of N cores, each register will be $lg(N)$ wide. Each register contains a count of the number of cores accessing that unit of the heap and is indexed according to the offset from the start of the heap. This allows cores passing messages through the shared heap to send offsets, allowing translation between the different virtual addresses of the heap on different cores. The ASH also requires several atomic instructions to access it.

4.2.1 Atomic Instructions

1. Allocate(Bytes) - Allocates a region of slots to the caller

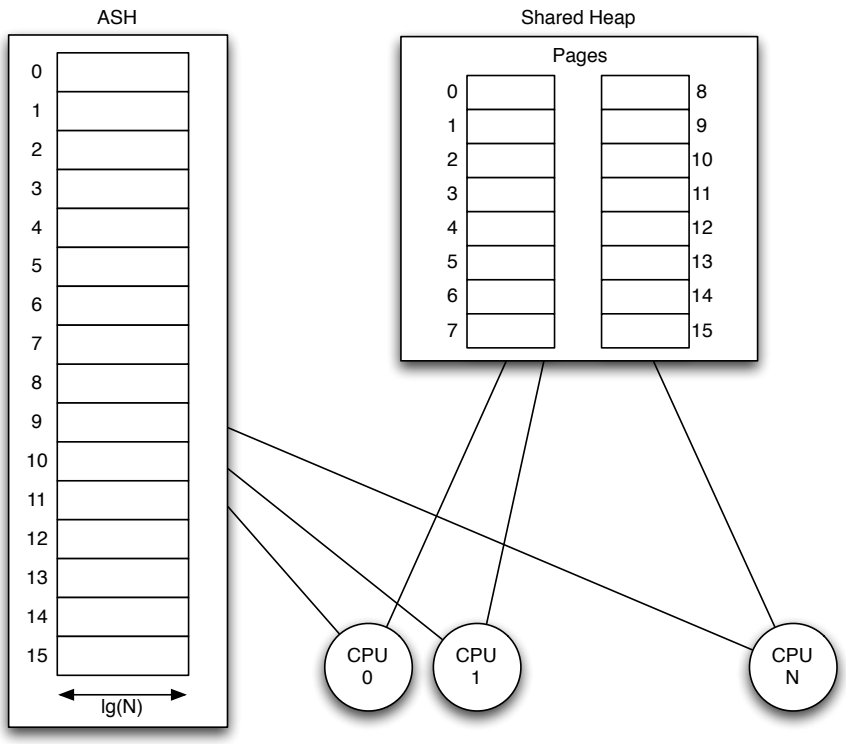


Figure 25: The Atomic Shared Heap

- Find a set of $Bytes/pagesize$ contiguous free slots
- Increment usage count of each slot in the set
- Return offset of the first slot in the set

2. IncrementUsage(Offset, Bytes) - Increments the usage count of the region so that another processor may access it
   - Increment the usage count of each slot from $Offset$ to $Offset+(Bytes/pagesize)-1$
3. Free(Offset, Bytes) - This is called by each reader of the region after the read is complete
   - Decrement the count of each slot from $Offset$ to $Offset+(Bytes/pagesize)-1$

The delays for the ASH have also been estimated based on the functional level design. The delay for the Allocate instruction is estimated as *pages in the heap*/2 cycles, as Allocate requires a search through the list of pages in the ASH for free slots. For some searches, the free slots will be at the beginning of the ASH, requiring only 1 cycle to find. For other searches, the free slots may be at the end of the ASH, requiring as many cycles as there are pages in the heap. This roughly averages out to half the number of pages int he heap. For IncrementUsage, the delay is estimated to be the number of pages being incremented. This instruction increments the register representing each page of the allocation, and each increment takes 1 cycle. The delay for Free, like that of IncrementUsage, is estimated to be the number of pages being freed. The Free instruction decrements the register representing each page of the allocation, and each decrement takes 1 cycle.

For our estimate of the area required for the Atomic Shared Heap on-die, we again used the CACTI tool, and assumed a system of 1024 cores to give us an upper bound. The size of the ASH is dependent on the size of the shared heap it corresponds to. We chose a generously sized shared heap of 16 Mbytes, with a page size 512 bytes. This gives us an ASH with 32,768 entries, and we assume that each entry is 64-bits wide. We again assumed that the area required by the logic is negligible compared to the area for the registers. Based on this set of assumptions, we used CACTI to estimate the size of a cache containing 32,768 8-byte lines, with direct associativity and a single bank in 65nm technology. This gives us an estimate of $3.343mm^2$ for the total area of the ASH. Comparing this to the size of the Intel Pentium Dual-Core, "Conroe", which is $143mm^2$ in 65nm technology, we find that the ASH would take roughly 2% of the area of the CPU for this large shared heap. The size of the ASH is dependent on the size of the shared heap, so it could also be adjusted if space was a concern.

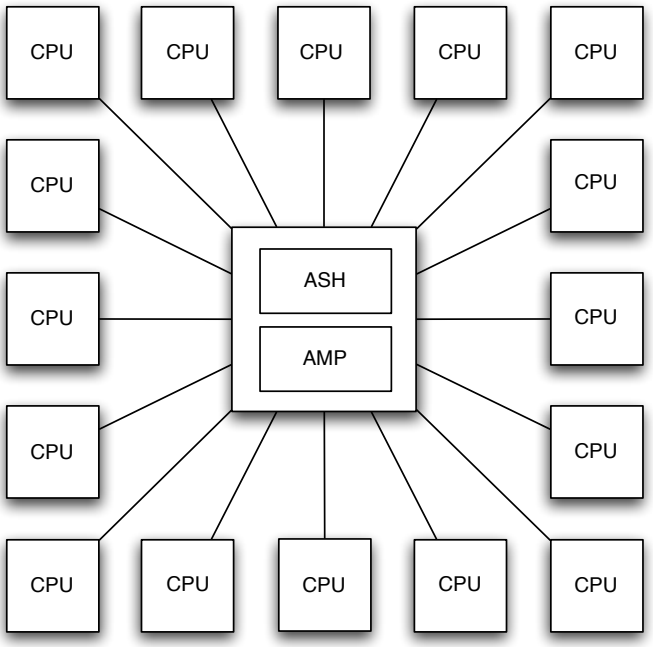


Figure 26: The Atomic Shared Heap and Atomic Message Passing units are centrally located on-chip.

4.3 Message Passing Algorithm

4.3.1 Initialization

- Before the simulation begins, the register for each page in the ASH is set to zero.
- In addition, the In and Out offsets for each circular queue in the AMP are set to zero.
- Allocate shared heap.
- Attach shared heap to each LP.

4.3.2 To Send Messages

- Call offset = Allocate(size of message) on the ASH.
- Write message in Heap[offset]
- For each receiver of the message

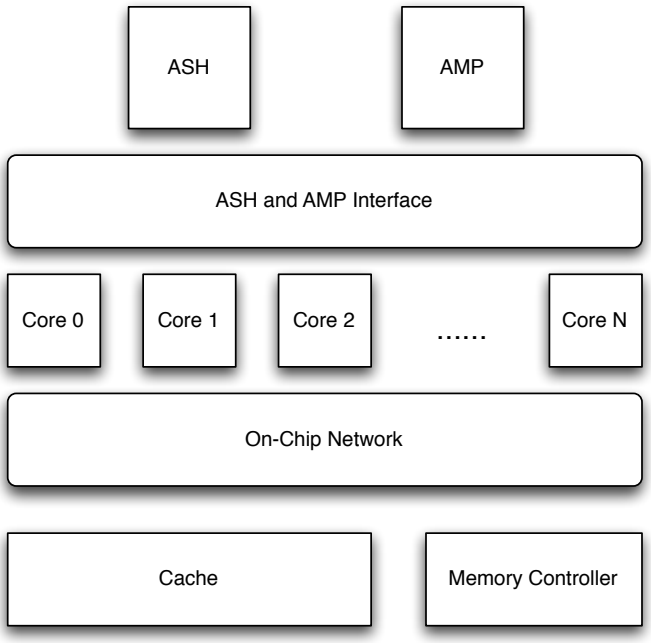


Figure 27: An example of how the ASH and AMP could be connected in a multi-core system.

  - Call IncrementUsage(offset, size of message) on the ASH.
  - Call Write(receiving LP ID, offset) on the AMP.

- Call Free(offset, size of message) on the ASH.

4.3.3 To Receive Messages

- Call offset=Read() on the AMP.
- While(offset != -1)
  - Process message at Heap[offset]
  - Call Free(offset, size of message) on the ASH.

4.4 Experimental Setup

In order to evaluate the performance of the Atomic Shared Heap(ASH) and Atomic Message Passing(AMP), we have written modules simulating these pieces of hardware in the Simics system simulator. To make a reasonable comparison against a

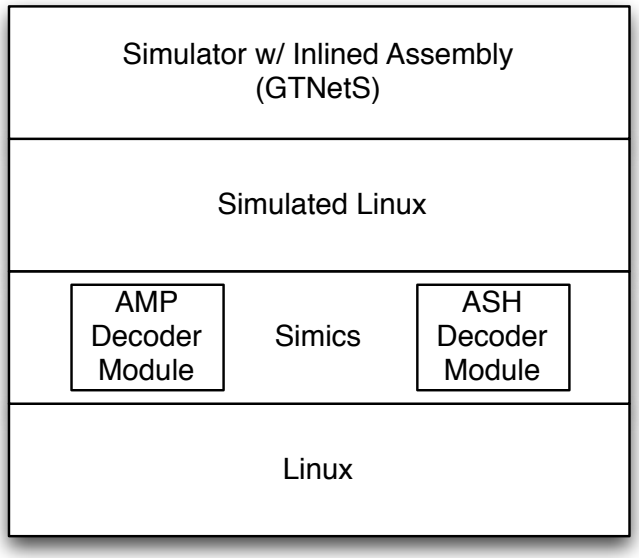


Figure 28: The software stack for the performance analysis of the Atomic Shared Heap and Atomic Message Passing.

shared memory implementation, we have estimated that each memory access takes 20 cycles. We have modified the code for the Georgia Tech Network Simulator to use a combination of ASH and AMP for copy-free message passing, using the new atomic instructions. For the performance analysis, we will compare the runtime in Simics of GTNetS with ASH and AMP, to that of the shared memory implementation of GTNetS using traditional message passing. For these experiments, we used a shared heap of $128kB = 131072$ *bytes* with a page size of 512 *bytes*. This gives us an ASH size of $131072/512 = 256$, which gives us a delay for the Allocate instruction of $256/2 = 128$ *cycles*. The delay for both the Free and IncrementUsage instructions is the number of pages in the allocation being freed or incremented. For the AMP, we used a Circular Queue size equal to the number of CPUs and delays of 2 cycles for both the Read and Write instructions.

4.5 Results

As seen in Figure 29, the version of GTNetS using ASH and AMP for message passing has runtimes that are 16 percent less than the version of GTNetS using traditional

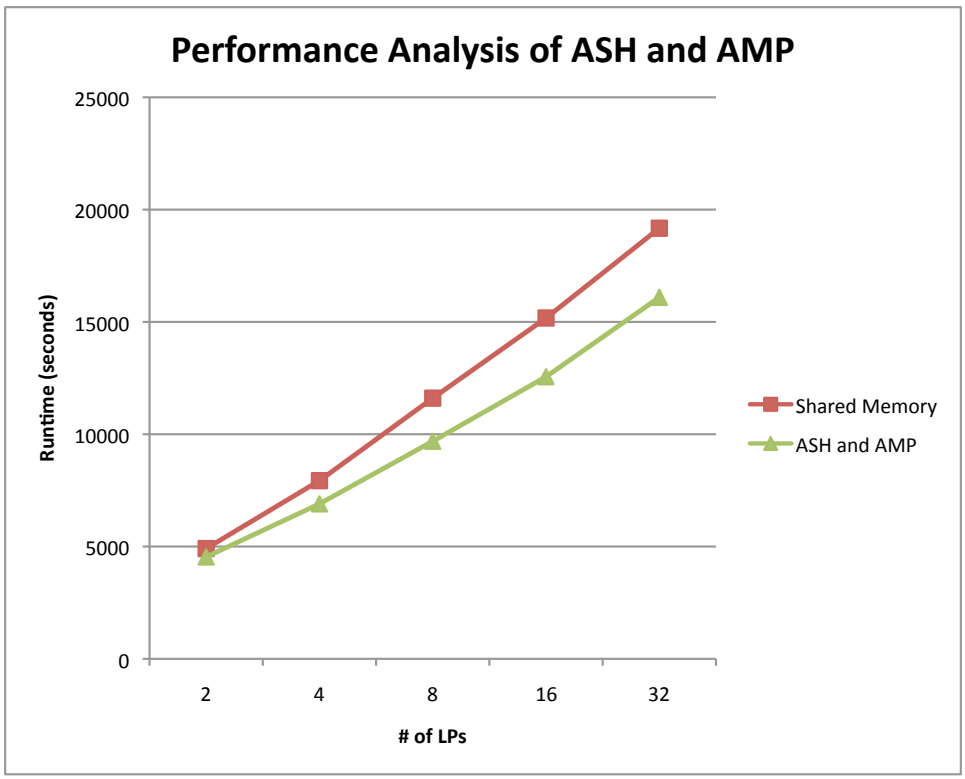


Figure 29: The results of the performance analysis of GTNetS using the Atomic Shared Heap and Atomic Message Passing versus the traditional shared memory implementation of GTNetS.

shared-memory message passing for up to 32 CPUs. We expect this performance improvement to continue for larger numbers of CPUs, as well.

4.6 The Atomic Shared Heap and Atomic Message Passing in Software

Much like the Global Synchronization Unit, the Atomic Shared Heap and Atomic Message Passing can be implemented in software to save the cost of fabricating specialized hardware or to take advantage of existing systems. In these cases the Atomic Shared Heap and Atomic Message Passing Units would each be implemented as a separate thread or process, as described below.

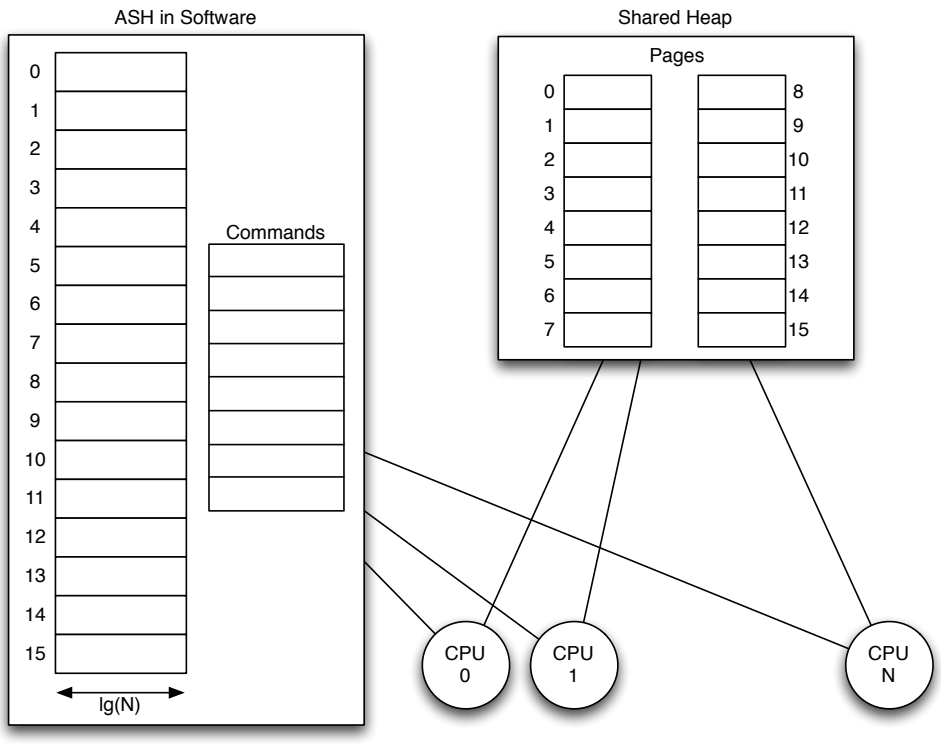


Figure 30: The Atomic Shared Heap implemented in software.

4.6.1 The Atomic Shared Heap in Software

The thread or process implementing the Atomic Shared Heap contains the same data structure as the hardware implementation, an array with an entry for each allocatable unit of the shared heap. The ASH is accessed using the same atomic instructions, Allocate, IncrementUsage, and Free. In order to maintain the atomicity of the instructions without the introduction of locks, an array of structures is added with an entry for each logical process, as seen in Figure 30. An LP writes an instruction for the ASH into its slot in the array, along with any required arguments. When the ASH has completed the execution of the instruction, the return value is written into the same entry in the array, and the instruction type variable in the structure is changed to REPLY. The ASH thread or process executes the calls written into the array using a round robin approach.

4.6.1.1 Algorithm for the Atomic Shared Heap in Software

In this section we describe the algorithm used in the software implementation of the Atomic Shared Heap. The algorithm that follows would run in a dedicated thread or process, separate from Atomic Message Passing. This thread would run for the duration of the simulation and would be accessed by the logical processes for use in message passing.

- While the simulation is not complete
  - For each core
    - Check the command field
    - If command is REPLY
      - Skip
    - If command is Allocate
      - Find the requested number of contiguous free slots
      - Write the index of the first of those slots into the return value
      - Change the command to REPLY
    - If command is IncrementUsage
      - Starting at the specified index, increment the usage count for each of the slots in the allocation
      - Change the command to REPLY
    - If command is Free
      - Starting at the specified index, decrement the usage count for each of the slots in the allocation
      - Change the command to REPLY

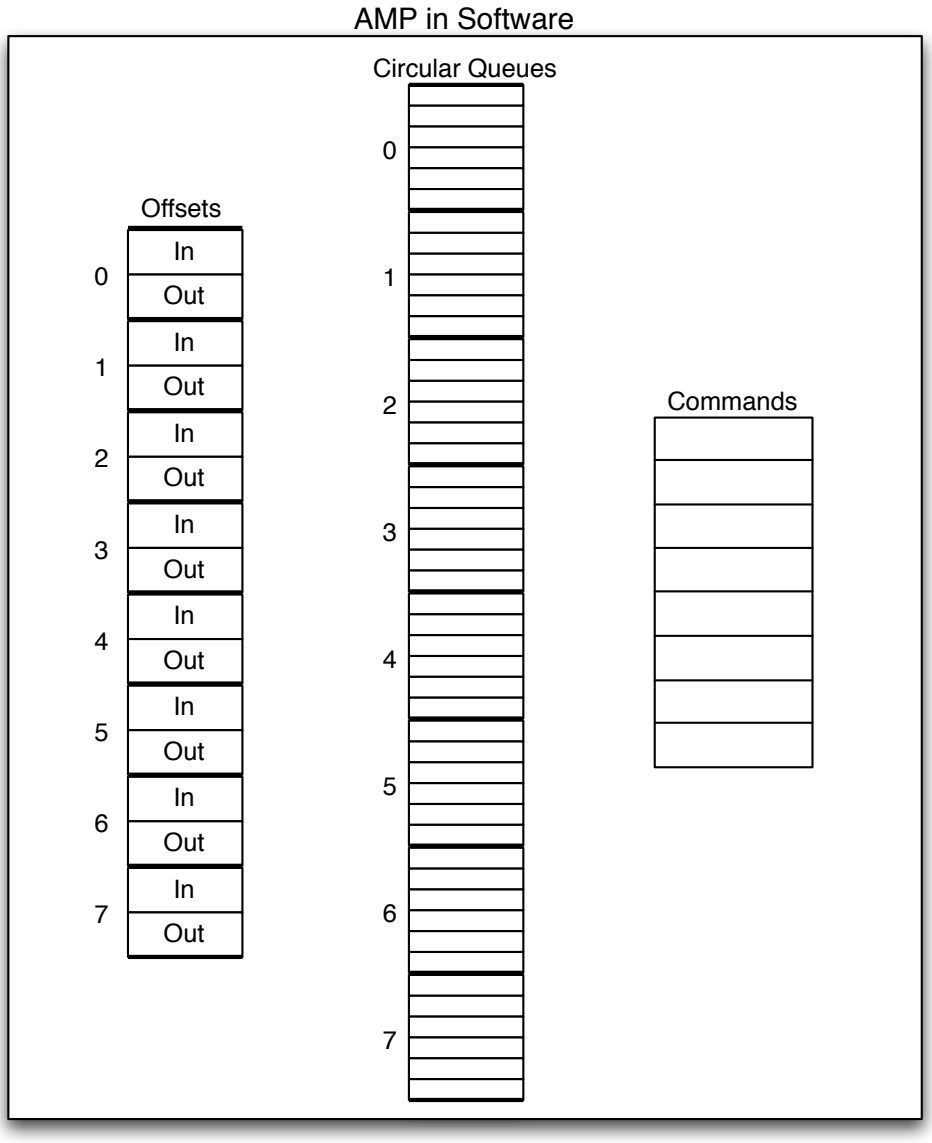


Figure 31: The Atomic Message Passing implemented in software.

4.6.2 Atomic Message Passing in Software

The Atomic Message Passing unit can be implemented in a separate thread or process in a similar manner. Like the hardware implementation, the AMP thread or process contains a circular queue for each logical process, as well as the In and Out offsets for each. The software AMP uses the same atomic instructions, Read and Write. Eliminating contention for locks, each LP has an entry in an array of structures which is used to write the instructions and arguments for the AMP, as seen in Figure 31. In addition, the result will be written in the array when an instruction is executed, along with the REPLY instruction type. The AMP thread or process will also service the calls in the array in a round robin order.

4.6.2.1 Algorithm for Atomic Message Passing in Software

In this section we describe the algorithm used by the software implementation of Atomic Message Passing. This algorithm would run in a separate thread or process from both the logical processes and the Atomic Shared Heap for the duration of the simulation, and would be accessed by the LPs for use in message passing.

- While the simulation is not complete
  - For each core
    - Check the command field
    - If command is REPLY
      - Skip
    - If command is Read
      - If the queue is empty ($In = Out$), write -1 in the return value as an indication of failure.
      - Otherwise, copy the pointer in the Out slot into the return value, and increment the Out offset ($Out = (Out + 1)\%k$).

- Change the command to REPLY

* If command is Write
  - If the queue is full ($In = (Out - 1)\%k$), write -1 in the return value as an indication of failure.
  - Otherwise, write the specified pointer to the message into the queue entry specified by the In value, and increment the In offset ($In = (In + 1)\%k$).
  - Change the command to REPLY

CHAPTER V

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

In this thesis, we have presented several hardware units to accelerate parallel discrete event simulation on multi-core systems. We have targeted the two primary sources of overhead, time synchronization and message passing. To accelerate the time synchronization for the simulation, we have presented a Global Synchronization Unit. Through our experiments, we have demonstrated that the GSU can reduce the runtime of a low-lookahead simulation running on up to 32 cores by approximately 40 percent. In addition, we have presented a hardware solution for lock-free, zero-copy message passing using two devices, the Atomic Shared Heap and Atomic Message Passing. Through the use of these two hardware units together, we have demonstrated that the use of ASH and AMP for message passing can reduce the runtime by approximately 16 percent for a low-lookahead simulation running on up to 32 cores. The use of these these specialized hardware units will greatly reduce the overhead of parallel discrete event simulators on multi-core systems, allowing users to run longer and more detailed simulations. These improvements will have the greatest impact on low-lookahead systems, such as wireless and on-chip networks.

5.2 Future Work

- In Section 3.9, we presented a design for the Global Synchronization Unit that takes into account non-uniform lookahead between the LPs. However, in our experiments, we only measured the performance of the GSU with the uniform

lookahead assumption. Using the GSU with non-uniform lookahead may increase the possible parallelism for some cases. A performance analysis of the GSU with non-uniform lookahead should provide an indication of which cases can benefit the most from the extension.

- In Section 3.8, we presented a software design for how the Global Synchronization Unit could be implemented as a separate thread or process. This software implementation is a way to use the GSU approach on existing systems or without the cost of the GSU hardware. A performance analysis of this software implementation will determine how this approach compares to both traditional shared memory synchronization algorithms and to the hardware implementation of the GSU. This software implementation of the Global Synchronization Unit is the subject of ongoing research.

- We presented a design for a software implementation of both the Atomic Shared Heap and Atomic Message Passing in Section 4.6. This software implementation is a way to use the ASH and AMP approach for zero-copy message passing both on existing systems, and in circumstances where the cost of custom hardware is prohibitive. A set of experiments comparing this software approach to both the hardware implementations of ASH and AMP and to traditional shared-memory message passing will determine its effects on application performance. This software implementation is the subject of ongoing research.

- In Chapters 3 and 4 we presented functional-level designs for the Global Synchronization Unit and the Atomic Shared Heap and Atomic Message Passing. An area of future research will be a gate-level design of the GSU, ASH and AMP. This gate-level design will allow an analysis of the area and energy-usage required by these devices.

- In Chapters 3 and 4 we presented separate performance analyses of the Global

Synchronization Unit and of the Atomic Shared Heap with Atomic Message Passing. Given the performance improvements these devices provide separately, it would be desirable to measure their impact when used complementarily.

- For the experiments presented in Section 4.4 we used a shared heap of 128kB with a page size of 512 bytes, giving us an Atomic Shared Heap size of 256 entries. Because the delay for the Allocate instruction for the ASH is proportional to the size of the ASH, it would be worthwhile to conduct an investigation of the impact of the size of shared heap and the page size on the performance of the ASH.

- In Section 4.3 we presented a basic algorithm for using the ASH in a simulator loop. However in some cases, there may be ways to optimize the ASH use. A performance analysis of these techniques would help application writers to decide which optimizations would be best suited to their applications. Some examples are described below:

  - If there are a few messages that are sent over and over again, leave them in the ASH and increment/free each time they are sent, rather than re-allocating and writing each time. This not only saves the time to write the message into the shared heap, but also the delay from the Allocate instruction on the ASH, which is the most costly instruction.

  - If the messages sent are frequently the same size, senders can re-use the same allocation in the ASH for messages, rather than freeing and allocating for each message sent. This saves the delay from the ASH Allocate instruction, which is the longest delay.

REFERENCES

[1] "http://www.virtutech.com."

[2] Bryant, R., "Simulation of packet communication architecture computer systems," *portal.acm.org*, Jan 1977.

[3] Chandy, K. and Misra, J., "Distributed simulation: A case study in design and verification of distributed programs," *Software Engineering, IEEE Transactions on*, vol. SE-5, pp. 440 – 452, Sep 1979.

[4] Chandy, K. and Misra, J., "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, Apr 1981.

[5] Das, S., Fujimoto, R., Panesar, K., Allison, D., and Hybinette, M., "Gtw: a time warp system for shared memory multiprocessors," *Simulation Conference Proceedings, 1994. Winter*, pp. 1332–1339, 11.

[6] Fujimoto, R., Tsai, J.-J., and Gopalakrishnan, G., "Design and performance of special purpose hardware for time warp," *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, Jun 1988.

[7] Fujimoto, R. M., "Time warp on a shared memory multiprocessor," *Proceedings of the International Conference on Parallel Processing*, vol. 3, pp. 242–249, Aug 1989.

[8] Fujimoto, R., Tsai, J.-J., and Gopalakrishnan, G., "Design and evaluation of the rollback chip: special purpose hardware for time warp," *Computers, IEEE Transactions on*, vol. 41, pp. 68–82, Jan 1992.

[9] Jefferson, D., "Virtual time," *Transactions on Programming Languages and Systems (TOPLAS*, vol. 7, Jul 1985.

[10] Kim, K., Colmenares, J., and Rim;, K.-W., "Efficient adaptations of the non-blocking buffer for event message communication between real-time threads," *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, pp. 29 – 40, Apr 2007.

[11] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, Jul 1978.

[12] Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B., "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, Feb 2002.

[13] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A., "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[14] Michael, M. M. and Scott, M. L., "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 51, pp. 1–26, May 1998.

[15] Misra, J., "Distributed discrete-event simulation," *Computing Surveys (CSUR*, vol. 18, Mar 1986.

[16] Nicol, D. M., "Noncommittal barrier synchronization," *Parallel Comput.*, vol. 21, no. 4, pp. 529–549, 1995.

[17] Quaglia, F. and Santoro, A., "Nonblocking checkpointing for optimistic parallel simulation: description and an implementation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 14, pp. 593–610, Jun 2003.

[18] Riley, G. F., "The georgia tech network simulator," in *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, MoMeTools '03, (New York, NY, USA), pp. 5–12, ACM, 2003.

[19] Riley, G. F., Jaafar, T. M., Fujimoto, R. M., and Ammar, M. H., "Space-parallel network simulations using ghosts," in *Proceedings of the eighteenth workshop on Parallel and distributed simulation*, PADS '04, (New York, NY, USA), pp. 170–177, ACM, 2004.

[20] Riley, G., Ammar, M., and Zegura, E., "Efficient routing using nix-vectors," in *High Performance Switching and Routing, 2001 IEEE Workshop on*, pp. 390 –395, May 2001.

[21] Rosu, M., Schwan, K., and Fujimoto, R., "Supporting parallel applications on clusters of workstations: The intelligent network interface approach," *High Performance Distributed Computing, 1997. Proceedings. The Sixth IEEE International Symposium on*, pp. 159–168, Aug 1997.

[22] Sokol, L., Weissman, J., and Mutchler, P., "Mtw: an empirical performance study," *Simulation Conference, 1991. Proceedings., Winter*, pp. 557 – 563, Nov 1991.

[23] Srinivasan, S. and Reynolds, P., "Non-interfering gvt computation via asynchronous global reductions," *Simulation Conference Proceedings, 1993. Winter*, pp. 740–749, Dec 1993.

[24] Srinivasan, S. and Paul F Reynolds, J., "Elastic time," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 2, pp. 103–139, 1998.

[25] Tsigas, P. and Zhang, Y., "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, Jul 2001.

[26] Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., and Borkar, S., "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *Solid-State Circuits, IEEE Journal of*, vol. 43, pp. 29 – 41, Jan 2008.