Hardware Supported Time Synchronization in Multi-Core Architectures

Elizabeth Whitaker Lynch, George F. Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0250
{whitaker, riley}@ece.gatech.edu

Abstract—We present a design for a hardware supported global synchronization unit that would be implemented on-chip and directly accessible by all processors in a multi-core architecture. This global synchronization unit will provide all processors with access to global state information from all other processors in just a few clock ticks, and can be used to perform highly efficient and scalable time synchronization for parallel simulations. Further, our design takes into account the possibility of transient messages, and allows for non-uniform lookahead between processors in conservative synchronization methods. Simulating this hardware in a system simulator, we demonstrate its ability to decrease the runtime of a low-lookahead network simulation by a factor of two over a shared-memory barrier synchronization.

I. INTRODUCTION

As the desire to simulate larger systems at finer granularity has increased, the CPU demands of these simulation processes has grown beyond reasonable bounds for a single processor. One prior solution has been to parallelize the simulation onto multiple processors, using either a cluster or a shared-memory multiprocessor. With recent and near-future advances in multi-core technology, where multiple CPUs are put on a single chip, there are greater opportunities to perform distributed simulations with multiple CPUs in a tightly coupled environment. However, parallelizing these simulators imposes an additional, well-known constraint, which is that the current simulation times of the processes must not be allowed to diverge significantly. If they do, a process could receive messages with timestamps less than its current simulation time. To prevent these so-called causality errors, a frequent time synchronization protocol is needed.

Present day multi-core chip designs commonly utilize two to four independent central processing units (cores), with eight to sixteen cores planned for the near future. It is likely that in just a few years multi-core chips will have dozens or even hundreds of independent processors. It is natural to expect that such hardware platforms will lead to bigger and faster parallel simulations that take advantage of the large number of inexpensive processing units. However, past experience on similar tightly-coupled architectures indicates that, regardless of the amount of processor power available to a parallel simulation, the substantial overhead for interprocessor synchronization is often the performance bottleneck, particularly in conservative approaches with low lookahead.

We have designed a hardware-supported global synchronization unit (GSU) to be implemented on-chip and directly accessible by all processors in the multi-core architecture. This global synchronization unit is comprised of specialized register files that are read and updated using atomic hardware instructions that allow efficient time synchronization actions amongst the CPUs in a multi-core architecture. In addition, our design supports cases where messages are in transit between processors (cores) and have not yet been received and processed by the destination. These so-called Transient Messages have historically been a source of difficulty and additional overhead for traditional time synchronization algorithms. With the global synchronization unit accounting for transient messages is simply a matter of updating (atomically) the appropriate values in one of our register files. This approach will improve simulation performance on multi-core architectures, enabling larger and more detailed simulations.

II. BACKGROUND AND MOTIVATION

When parallelizing a simulation it is crucial to ensure that an instance of the parallel simulator, called a logical process (LP), never processes events out of timestamp order. This property is called causality. When an LP receives a message with a timestamp less than the current simulation time, a causality error occurs. Causality can be guaranteed by using synchronization between the LPs. Synchronization algorithms can be grouped into two main classes according to how they handle the issue of causality: conservative and optimistic.

Conservative synchronization algorithms enforce causality by prohibiting an LP from processing an event with timestamp \( t_1 \) until it can be guaranteed that no message will be received with a timestamp \( t_2 < t_1 \). This is done either by using null messages [1], [2], or by computing the lower bound on the time stamps (LBTS), using one of several existing algorithms to find the LBTS value [3]–[7]. Virtually all of the existing algorithms require numerous message exchanges between LPs and timely cooperation by all LPs. The performance and scaling of conservative synchronization algorithms has been studied in detail [8].

The procedure used by a typical conservative synchronization algorithm is as follows. Each LP has an upper bound on the timestamp of events that are currently safe to process.
The LP processes events until it has no more events that are safe, with timestamps less than the upper bound. When this occurs, the LP must compute the LBTS of all $N$ LPs in the simulation. The LBTS for an LP $j$ with a minimum event timestamp (including all unprocessed messages it has sent) of $T_j$ is defined as the $\min(T_j + \text{lookahead}_{i,j})$ for $j = 0$ to $N$. $T_j$ is the minimum event timestamp of LP $j$, including any unprocessed messages sent by LP $j$. The lookahead is the minimum simulation time possible between the current time at LP $j$ and the timestamp of an event with a destination of LP $j$. Lookahead is determined by the physical properties of the system being simulated. Such algorithms are, however, complicated by the existence of transient messages. Transient messages (messages that have been sent but not yet processed by their receiver) are handled at each LP by keeping track of the number of messages it has both sent and received. When an LBTS computation is performed, each LP disseminates not only the timestamp of its next event but also the difference of its count of sent messages and received messages. Before computing the minimum timestamp of all the LPs, an LP must sum the differences of the messages received and messages sent for all LPs. Only if these differences sum to zero can the LBTS be determined. Otherwise, there are transient messages in the system and the communication must be performed again, resulting in an additional overhead penalty.

Conventional conservative synchronization, as described above, requires global communication between all $N$ LPs to calculate the LBTS. However, this global communication scales poorly. The best implementations on typical hardware scale with $O(N \times \log(N))$ [7]. Also, using this type of synchronization an LP that has run out of events that are safe to process must sit idle waiting for all other LPs to participate before the LBTS computation is performed. As the number of LPs increases, this idle time becomes prohibitive.

Optimistic algorithms permit events to be processed before causality is guaranteed. However, if a message is received with a timestamp less than the current simulation time (a causality error), the simulation is rolled back to its state at some time prior to the timestamp of the message. In order to perform this rollback, the state of the simulation must be saved periodically. To prevent the state from accumulating indefinitely, the Time Warp optimistic synchronization algorithm periodically calculates the global virtual time (GVT). GVT is calculated as the lowest timestamp of all unprocessed events and transient messages in the system. Any state prior to the GVT is no longer needed and can be discarded. This is called fossil collection [9].

We are specifically addressing conservative parallel simulation applications on multi-core architectures. We assume that each of the LPs in the parallel simulation is executing on an independent processor (core) in the multi-core system, and has some method of sending event messages to the other LPs. In a tightly-coupled shared-memory environment such as a multi-core chip, this is likely accomplished via shared memory queues, but the actual method used is not relevant to our synchronization approach. Using our hardware-supported approach, the global minimum computation needed to determine a lower-bound timestamp can be accomplished in just a few clock cycles, and can be done by each processor independently from the others. The details of this are discussed in the next section.

III. THE GLOBAL SYNCHRONIZATION UNIT

In this section, we describe the functionality of the various components in our global synchronization unit (GSU). The GSU consists of several cooperating register files and atomic read/write instructions and is centrally located, with one GSU per multi-core chip. The use of this synchronization approach for parallel discrete event simulation eliminates the need for costly global communications, replacing them instead with communication with the GSU on the order of a few clock cycles. In addition, this approach allows each LP to calculate a new LBTS as needed, without waiting for all the other LPs to participate in the global communication. This greatly reduces the amount of time each LP spends waiting for other LPs.

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 the system. We relax the uniform lookahead assumption later in this paper.
1) The Minimum Outstanding Event (MOE) register file consists of N floating point values. Our design is such that these can be read and written in just a few clock cycles. In addition, there are N − 1 comparators that compute the minimum value in the MOE file in \(lg(N)\) stages.

2) The Minimum Outstanding Message (MOM) register file consists of N floating point values, identical to the MOE, also including N − 1 comparators computing the minimum value found in the file. Again, this register file can be read and written to in just a few clock cycles.

3) The Transient Message Count (TMC) register file consists of N integer values. Each of these can be atomically incremented and decremented.

4) The Minimum Timestamp (MTS) instruction computes the smaller of the two minima in the MOE and MOM register files.

5) An atomic increment instruction increments any value in the TMC register file.

6) An atomic decrement instruction decrements any value in the TMC register file.

7) An atomic write if less than instruction that will write a new value in either the MOE or the MOM but only if the new value is less than the value already present in the register file. This instruction returns true if successful (the value was less than the existing value and was written).

8) An atomic write if zero instruction that will write a new value in either the MOE or MOM but only if the corresponding value in the TMC register file is zero. This instruction returns true if successful (the value was zero) or false if not.

The approach works by keeping track of not only the timestamp of the minimum event in each LP’s event queue but also the timestamp of the minimum unprocessed message destined for each LP. Each LP processes events until all received messages have been placed in the event queue and no events safe to process remain in the queue and then requests an MTS computation from the GSU. After an LP completes the processing of an event, it updates its MOE register in the GSU, reflecting its new minimum event timestamp. When an LP generates a message for another LP, it first atomically increments the TMC register for the destination LP, then uses the atomic write if less than instruction to update the destination LP’s MOM register to reflect the new minimum unprocessed message for that LP, sends the message, and, finally, updates its own MOE register to reflect its new minimum event timestamp. After an LP receives a message, it first inserts it into its event queue, then updates its MOE register to reflect the timestamp of the earliest event in the event queue, which might be earlier due to the new message, and atomically decrements its TMC register. The LP repeats this process until its TMC register contains zero and uses the atomic write if zero instruction to update its MOM register to infinity. The atomic write if zero may fail if a message has been sent to the LP since it atomically decremented its TMC register. In this case, the LP will process the pending messages and then retry. The procedures followed by LPs are described in more detail below.

The action taken by an LP to determine the lower bound on timestamp (including lookahead) is as follows:

1) Read and enqueue in the LP’s event list all messages that have been sent but not received. This is done as follows:

   ```
   while(TMC[myLpIndex] !=0)
   Read an event message from the pending message queue.
   Enqueue the event in the LP’s event list
   Atomically decrement TMC[myLpIndex]
   end
   ```

2) Use the atomic write if less than instruction to update the MOE register if the timestamp of any of the new events is less than the current MOE value.

3) Use the atomic write if zero instruction to write a value of infinity in the MOM register, if the TMC register file has a zero entry. If this atomic write fails, return to step 1 above. This means that a new message has arrived from another LP after detecting that there are no more messages, but before updating the MOM register.

4) Use the Minimum Timestamp instruction to find the smallest outstanding timestamp. The Minimum Timestamp instruction returns the minimum value for all entries in both the MOE and MOM registers. This represents the timestamp of the globally smallest event in the distributed simulation, including Transient Messages.
5) Add the constant lookahead value to the global minimum; the result is the timestamp of the upper bound of the safe event window for a given LP.
6) The LP can safely de–queue and process any event in the local event list with a timestamp less than or equal to the global minimum plus lookahead.

When processing events and sending events to other LPs, each LP does the following:

1) If the earliest event in the event queue is not safe, check for pending messages and perform the MTS computation above, and repeat until an event is safe for processing.
2) Remove the earliest event from the event list, but do not update the corresponding MOE value.
3) Advance local simulation time and process the event.
4) Prior to sending any event message to another LP, use the atomic increment instruction to increment the TMC register file entry for the receiving LP.
5) Use the atomic write if less than instruction to update the MOM entry for the receiving LP.
6) Write the event message to the receiving LPs pending message queue.
7) After all processing for an event has completed, update the MOE entry with the timestamp of the new smallest pending event in the event queue.
8) Return to step 1 above.

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 2 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 4, and the state of each LP’s event queue can be seen in Figure 5. A timeline of the LPs’ actions and the value of an LBTS computation at that time can be seen in Figure 3.

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.

![Fig. 3. Timeline for example](image)

![Fig. 4. Global Synchronization Unit state for example](image)

![Fig. 5. Event queue state for example](image)
because MOE[B] 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.

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 [10]. However the computation is only completed with the participation of all LP, unlike our GSU, which can return an MTS on demand.

A. Non-uniform Lookahead

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 the addition of an $N \times N$ register file containing the pairwise lookahead values of the LP, as seen in Figure 6. 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, the GSU will have to compute the $\min(\min(MOE_j, MOM_j) + \text{lookahead}_{j})$ for $j = 0$ to $N$. Our design will again accomplish this with $\log(N)$ stages of comparators.

![Global Synchronization Unit](image)

Fig. 6. The components of a global synchronization unit with non-uniform lookahead.

IV. PERFORMANCE

For our performance benchmarks we ran two simulators, RandomSim and GTNetS, in parallel inside the Simics system simulator [11]. Using Simics, we simulated an Pentium 4 multi-core system with 2 to 8 cores and added new instructions using unused opcodes in the x86 instruction set to read from and write to the register files of our GSU. Because our synchronization hardware has not been specified at a gate level, the delays required for these atomic instructions are not hard-coded into the model, but specified as parameters. For these simulations, the delay for calculating the Minimum Timestamp was 2 cycles, atomic increment/decrement was 3 cycles. Write if Less Than was 4 cycles, and Write if Zero was 5 cycles. Both the baseline and GSU versions of RandomSim and GTNetS use circular queues for message passing. For time synchronization the baseline versions use barrier LBTS computation in shared memory, while the experimental versions use the atomic instructions to access the synchronization hardware.

One metric commonly used to analyze performance of parallel programs is speedup. The speedup of a parallel algorithm running on $p$ processors is defined as $S_p = T_1/T_p$, where $T_1$ is the runtime of the serial algorithm and $T_p$ is the runtime of the parallel algorithm on $p$ processors. Generally considered to be ideal, linear speedup is where $S_p = p$. The problem size of both simulations grows linearly with the number of LP, so linear speedup would correspond to a constant runtime.

A. 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.

Figure 7 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.

**B. The Georgia Tech Network Simulator**

Our second benchmark use The Georgia Tech Network Simulator (GTNetS). The network simulation used in GTNetS is a star topology with 100 leaf nodes on each LP. Each link has a delay of $1\mu s$, which gives the simulation a uniform lookahead of $1\mu s$. This low-lookahead increases the need for synchronization to levels comparable to wireless network simulation. Each leaf node has a TCP application which sends messages to a leaf node on another LP at a constant rate. The simulation runs for 10s of simulation time.

In Figure 8 the runtimes of the baseline and GSU versions of GTNetS show that the GSU greatly decreases the runtime and cuts the rate of growth by half. The baseline data points for 7 and 8 CPUs are missing because of memory constraints in the system simulator.

**V. RELATED WORK**

A variety of hardware accelerators for distributed synchronization have been proposed in the past. Srinivisan and Reynolds proposed a Parallel Reduction Network (PRN), a tree of ALUs with $\log(n)$ levels, where $n$ is the number of CPUs. Each CPU has an auxiliary processor responsible for passing a state vector into the PRN from the simulation and for passing the results of the PRN back to the simulation. The reductions for different elements in the state vector are pipelined. To use the PRN to compute the GVT, the auxiliary processor passes in a state vector consisting of the current simulation time of the LP and the minimum timestamp of all unreceived messages sent by the LP. In addition, the auxiliary processor must acknowledge receipt of all messages [12]. Srinivisan and Reynolds also proposed the Elastic Time Algorithm, which is an adaptive synchronization protocol using Near-perfect State Information (NPSI) to calculate an Error Potential (EP). The EP is then used to control optimism by introducing wall clock delays proportional to the EP between processing events. The EP is calculated by a NPSI calculator (NPSIC), with Input State Vectors from LPs. This NPSIC is realized using a Virtual Communication Machine (VCM) implemented on the network interface co–processor. The VCM is granted access to the address space of the application and is responsible for replicating the event queue of each LP on every other LP and maintaining consistency between the local copy and the remote copies. This knowledge of the state of every LP allows an LP to calculate GVT using only local data [14]. Our approach does not require replication of event queues, thus reducing the memory footprint.

Another solution, proposed by Rosu et al., is a Virtual Communication Machine (VCM) implemented on the network interface co–processor. The VCM is granted access to the address space of the application and is responsible for replicating the event queue of each LP on every other LP and maintaining consistency between the local copy and the remote copies. This knowledge of the state of every LP allows an LP to calculate GVT using only local data [14]. Our approach does not require replication of event queues, thus reducing the memory footprint.

Other proposals have been made for the hardware acceleration of state-saving. Fujimoto et al. proposed a Rollback Chip, which would be coupled to a single CPU and handle the state saving required for the Time Warp optimistic synchronization protocol. The CPU uses a small number of operations to access the rollback chip, including READ, WRITE, ROLLBACK, and ADVANCE. Computation of the GVT to determine which
states are no longer required is performed by the CPU [15].
Quaglia and Santoro have proposed a solution for nonblocking
checkpointing in optimistic simulations. The simulation data is
partitioned into three categories, State Buffers (SB), Checkpoint
Stacks (CS), and Other Data Structures (ODS). The presence of
a device which can copy data from SB to CS, such as the
DMA hardware for Myrinet, is assumed. The use of this device
allows checkpointing while the CPU continues to process
events [16]. As opposed to [12]–[16], our solution does not
require a chip for each CPU/core, but only one for the entire
multi-core system. In addition, as we are targeting conservative
simulation and not optimistic, checkpointing/state-saving as in
[15], [16] is not required.

In addition, Fujimoto has proposed a Time Warp variation
for shared memory multiprocessors. In this algorithm, LPs
have direct write access to the other LPs’ event queues. To
send a message, the LP obtains the lock for the destination
event queue, writes the event in, and releases the lock. If
the LP subsequently wants to cancel the event, it obtains the
lock for the destination event queue and checks the event’s
Processed flag. If the flag is set to FALSE, the LP simply
deletes the event. If the flag is set to TRUE, the LP sets the
event’s flag to FALSE, and cancels all the events generated
by that event. The GVT is performed periodically using a
global barrier calculation, for fossil collection [17]. However,
as our work is restricted to conservative synchronization, event
cancellation is not required.

VI. FUTURE WORK

Future work will include extending the performance study
out to at least 64 processors, reflecting expected work–loads of
the near–future. We will also compare the performance of the
GSU to that of synchronization using the Message Passing
Interface (MPI) reduction operations. In addition, we will perform a sensitivity analysis of the impact on performance
of the time delays for each of the atomic instructions. We
will also write a gate–level design of the hardware in VHDL,
thus measuring the actual delays resulting from our hardware
instructions. Finally, we will extend our design to accommodate simulations with non-uniform lookahead and measure its
effect on performance.

VII. CONCLUSIONS

We presented a design for hardware-assisted conservative
time synchronization, a Global Synchronization Unit. The
Global Synchronization Unit yielded smaller runtimes than its
counterpart using shared-memory synchronization. In addition,
the rate of increase with each additional CPU is less. These
results indicate that the GSU has potential for making parallel
simulation much more efficient, especially for simulations with
low–lookahead, such as wireless simulations, executing on a
large number of multi-core CPUs.

REFERENCES

design and verification of distributed programs.” in IEEE Transactions
on Software Engineering, September 1979.


[4] Y. B. Lin and E. D. Lazowska, “Determining the global virtual time in
a distributed simulation,” in Proceedings of International Conference on
Parallel Processing, 1990, pp. 201–209.

virtual time approximation,” in Journal of Parallel and Distributed

[6] ——, “Experience with a new distributed termination detection algo-
rium,” in Proceedings of the 2nd International Workshop on Distributed


of large-scale network simulations,” Parallel and Distributed Simulation,


a time warp system for shared memory multiprocessors.” Simulation
1994.


asynchronous global reductions,” Simulation Conference Proceedings,


on clusters of workstations: The intelligent network interface approach,”

of the rollback chip: special purpose hardware for time warp.” Compute-

[16] F. Quaglia and A. Santoro, “Nonblocking checkpointing for optimistic
parallel simulation: description and an implementation,” Parallel and
Distributed Systems, IEEE Transactions on, vol. 14, no. 6, pp. 593–610,
June 2003.

Proceedings of the International Conference on Parallel Processing,