Optimistic Simulation of Parallel Architectures Using Program Executables *

Sashikanth Chandrasekaran and Mark. D. Hill
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
wwt@cs.wisc.edu

Abstract

A key tool of computer architects is computer simulation at the level of detail that can execute program executables. The time and memory requirements of such simulations can be enormous, especially when the machine under design—the target*—is a parallel machine. Thus, it is attractive to use parallel simulation, as successfully demonstrated by the* Wisconsin Wind Tunnel *(WWT). WWT uses a conservative simulation algorithm and eschews network simulation to make lookahead adequate. Nevertheless, we find most of WWT's slowdown to be due to the synchronization overhead in the conservative simulation algorithm.*

This paper examines the use of optimistic algorithms to perform parallel simulations of parallel machines. We first show that we can make optimistic algorithms work correctly even with WWT's direct execution of program executables. We checkpoint processor registers (integer, floating-point, and condition codes) and use executable editing to log the value of memory words just before they are overwritten by stores. Second, we consider the performance of two optimistic algorithms. The first executes programs optimistically, but performs protocol events (e.g., sending messages) conservatively. The second executes everything optimistically and is similar to Time Warp with lazy message cancellation. Unfortunately, both approaches make parallel simulation performance worse for the default WWT assumptions. We conclude by speculating on the performance of optimistic simulation when simulating (1) target network details, and (2) on hosts with high message latencies and no synchronization hardware.

1 Introduction

Simulation is a popular technique to study and evaluate proposed computer architectures. To simulate the complex interactions in the proposed design, researchers must be able to run applications (i.e., program executables) in addition to stochastic workloads. Unfortunately, the time and memory requirements of such simulations can be enormous, especially when the machine under design is a parallel machine. Therefore, researchers have begun using parallel simulation—i.e, using an existing parallel machine (the *host*) to simulate the parallel machine under study (the *target*). A technique known as *direct execution* [3] is used to execute program executables. With direct execution, the host simulates only those features in the target machine that it does not support (e.g., cache-coherence, synchronization operations). In this paper we use the term *event* only to refer to those actions that require simulation by the host. The common features, such as program instructions, are not simulated—instead, they are directly executed by the host. The Wisconsin Wind Tunnel (WWT) is a simulator that executes program executables on a Thinking Machines CM-5 host to simulate cache-coherent shared-memory computers [14]. WWT directly executes all instructions and memory references that hit in the target cache. Program executables are edited so that during WWT's direct execution they also keep track of target execution time (by incrementing a counter). WWT regains control on cache misses and simulates the target cache, directory, etc., by sending timestamped messages.

WWT uses the conservative *time bucket* synchronization mechanism [18] to coordinate simulation of the processor nodes. Simulation proceeds in parallel for quanta of duration Q. Each node must synchronize with all other nodes at the end of every quantum, after which all nodes proceed in parallel for another quantum Q. Without target network simulation, the minimum target time between events and the events that they can generate in remote nodes (also called the *lookahead*) is the *target network latency*. In order to have sufficient lookahead, WWT avoids network simulation and sets the quantum length to the fixed target network latency. Nevertheless, we find most

*This work is supported in part by NSF Grant MIP-9225097, Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, Sun Microsystems and Thinking Machines Corporation, Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School.

Step	Time	% of total time
Direct Execution	131	3.99%
Protocol Simulation	740	22.55%
Quantum Synchronization	2338	71.26%
Barrier Synchronization	72	2.20%

Table 1: Breakdown of total simulation time with conservative simulation.
The table shows the breakdown of simulation time when simulating the program Ocean, on a target machine to be described in Section 2. *Time* is in millions of host cycles. Direct Execution refers to the time spent executing target instructions on the host node. Protocol simulation includes simulation of the target cache and target directory events. While quantum synchronization refers to the time spent by host nodes waiting for other nodes to reach the end of the quantum and for the messages to reach their destinations, barrier synchronization is the time taken for nodes to complete the barrier after the network has delivered all messages and the last node has entered the barrier. The target program was executed for 16 million target cycles.

of WWT's slowdown to be due to the synchronization overhead in the conservative simulation algorithm (Table 1).

In this paper, we examine the use of optimistic parallel simulation algorithms to simulate parallel machines using program executables. To the best of our knowledge, this is the first work to demonstrate how optimistic techniques can be used with directly executed programs. To correctly save the state of a target program, we checkpoint the processor state by copying the registers (integer and floating-point) and condition codes at the end of each quantum. Restoring the processor state only involves storing the values back into their respective registers. We incrementally save the target program's memory by editing the program executable to include instructions that log the value of memory words before every *store* instruction. To restore the memory state, we invoke a routine in the target program that replaces the old values in memory. Using these techniques, we study two optimistic approaches:

- A hybrid approach that optimistically executes target program instructions and memory references that hit in the target cache and conservatively simulates protocol events. We use Reynolds' terminology [15] and call this a *risk-free* optimistic approach.

- An aggressive Time Warp-like optimistic approach that executes all target program instructions, memory references (including those that miss in the target cache) and protocol events optimistically.

Optimistic approaches have been shown to perform better when simulating target systems such as queuing networks and when using stochastic workloads [12]. This paper, on the other hand, compares the performance of conservative and optimistic approaches by executing shared-memory applications. We find that for all the applications that we executed on a CM-5 host, the conservative technique performs better than either optimistic technique. The risk-free technique runs an average of 1.5 times slower than the conservative simulation. This technique improves the lookahead only by an average of 36 target machine cycles since shared-memory parallel programs spend significant time in communication and synchronization. The aggressive technique reduces the frequency of synchronization among the host nodes by up to 66%. However, rollback overheads dominate the execution and it is up to 2.5 times slower than the the conservative simulation.

We then speculate on simulation performance in two cases that could be more favorable to optimistic simulation: (1) Accurate simulation of the target network, which forces the lookahead to be much less than the constant network latency. At low lookaheads, synchronization overheads are exacerbated and optimistic techniques may yield better performance. (2) Simulation on hosts with high network latencies and no hardware support for synchronization. We speculate that the optimistic approaches would perform better on hosts where synchronization operations must be performed in software (at a higher cost).

The next section provides a brief background on our workloads. Sections 3 and 4 present the performance of the risk-free optimistic technique and the aggressive optimistic technique respectively. Section 5 presents the scenarios where optimistic techniques may offer better performance. Section 6 describes related work, Section 7 describes future work, and finally, Section 8 presents our conclusions.

2 Target Machines and Workloads

Our target machines are composed of nodes which contain a CPU, a 256KB 4-way set-associative data cache, and local memory. The nodes are connected by a network that has a fixed latency of Q (=100) cycles. An all-hardware directory-based coherence protocol is used to maintain a sequentially consistent view of shared-memory.

The three applications that we chose to present are *Ocean*, *Sparse*, and *Water*. Ocean and Water are from the SPLASH benchmark suite [17], and Sparse is a locally written shared-memory program. Ocean is a hydrodynamic simulation that models a two-dimensional cross-section of a cuboidal basin. The input data set used was a 98×98 grid. Sparse solves $AX = B$ in parallel for a sparse matrix A. The input matrix was a 256×256 dense matrix. Water is a water molecule simulation performed on 256 molecules for 10 iterations. We chose these benchmarks because they exhibit different computation/communication ratios and communication patterns. While Ocean has a very high communication overhead, Sparse has a moderate communication overhead and Water spends very little time in communication.

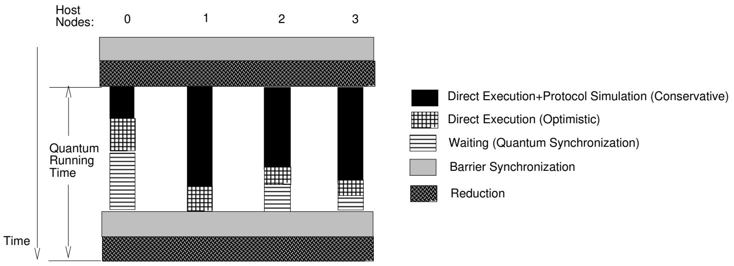

Figure 1: A pictorial view of the different steps in a quantum when using a risk-free optimistic technique.

3 A "Risk-Free" Optimistic Mechanism

In this section, we present the implementation and performance of a *hybrid* approach that adds optimism to the existing conservative mechanism. This strategy is similar to the Breathing Time Buckets strategy supported in SPEEDES [18]. Like the conservative technique, events are processed in quanta. However, these quanta do not have the constant length, Q. Instead, the *event horizon* determines the quantum length. The event horizon is the minimum event time of all events generated during optimistic simulation of the previous quantum plus the target network latency. We refer to the logical time of the event horizon as the *safe time*. The safe time is computed by performing a synchronous *min reduction* of the event times. Processing events beyond the safe time *may* cause time accidents, i.e., events processed beyond the safe time may have to be rolled back. Thus a quantum length is the logical time between two successive event horizons ($\geq$ Q).

The conservative technique (combining direct execution and simulation of events) is used to process *safe* events—events with timestamp less than the safe time. When a host node has no events that it can safely process, it checkpoints the target processor state and optimistically executes target program instructions. However, to confine rollbacks to the local processor, cache misses and other events beyond the safe time that may cause communication are not processed optimistically. Instead, the node participates in the global synchronization to check for causality errors and to compute the next event horizon. The probability of optimistically executed instructions being incorrect tends to increase the further simulation proceeds beyond the safe time. To prevent such incorrect computations from proceeding too far ahead into the logical time and increasing the probability of a rollback, we insert a *mega-quantum* expiration event that puts an upper bound on the optimistic phase in each cycle. Scheduling the mega-quantum expiration event at $3 \times Q$ cycles gives the best performance. Figure 1 illustrates the different steps in a quantum when using a risk-free optimistic technique.

We now discuss the techniques needed to save and restore the state of a directly executed program (Sections 3.1 and 3.2). Directly executed programs can manipulate any part of a target node's state. However, since the risk-free mechanism prevents the target program from executing beyond an event, we only need to save and restore the target program's register and memory state.

3.1 Saving Processor State

WWT schedules a checkpoint event at the end of each quantum. Whenever the target program returns control to WWT (due to a quantum expiration, or an event), the executive interface to the CM-5 kernel [13] saves the target global registers (including the program counter) and the condition codes in a buffer. We only need to copy the buffer and save the floating-point registers and the floating-point status register to *partially* checkpoint the target processor state. On a machine with no register windows (e.g., a MIPS-like architecture) the *entire* processor state could be constructed by making a copy of all the registers. On a SPARC processor, we need to save the registers in all the active register windows that were used by the target program before starting optimistic execution of the target program. When a time accident is detected, we copy the global integer registers, floating-point registers and the condition codes from the checkpoint buffer into the host node's registers. The target register window is restored before returning to the target program for restarting direct-execution from the checkpoint.

3.2 Saving Memory State

The simplest solution to saving memory state is to copy the entire target address space; however this involves very high overhead. Instead, the target memory is saved incrementally by logging all changes to it. The target program changes the state of memory by executing *store* instructions. We use EEL [9], an executable editing library, to instrument the target store instructions with a small piece of code (e.g., four instructions before a *store-word*) that loads the old value from memory and saves it in a log in the target address space.[1] Note that stores to both private and shared memory locations are logged. When a time accident is detected, we invoke an unroll procedure in the target program that restores the values in the target memory starting from the end of the log. This unroll procedure is linked to the target program along with the instrumentation of the store instructions. An important concern in optimistic techniques is the memory required for saving state. Our technique uses 4K bytes (a page) for logging changes to memory and less than 300 bytes to save the processor state. This overhead is less than 1% of the memory required by a program that has a data set size of 1MB per node.

[1] Actually, stores to target memory during the conservative phase are also logged, since it is more expensive to detect the state of the simulation and avoid logging. When a checkpoint is taken the log pointer is reset to the start of the log.

Application	Procs	% Computation	Conservative Simulation Time (million cycles)	Risk-Free Optimistic Simulation Time (million cycles)	Average Quantum Length (cycles)	Slowdown
Ocean	8	16	7097	9276	112	1.31
Ocean	16	14	4398	6127	107	1.39
Ocean	32	12	3245	4505	104	1.39
Sparse	8	67	1775	2880	136	1.62
Sparse	16	46	2098	2902	128	1.38
Sparse	32	25	3329	4249	120	1.28
Water	8	59	6733	10918	202	1.62
Water	16	51	5100	8800	170	1.73
Water	32	41	4029	7759	145	1.92

Table 2: Performance of conservative vs. risk-free optimistic simulation.

This table shows the percentage of time spent by the applications in computation, the simulation times, the average quantum length, and the slowdown of the optimistic technique when compared to the conservative simulation. The target network latency was set to 100 cycles. While the simulation times refer to the *host* cycles (in millions), the average quantum length refers to the *target* cycles.

3.3 Performance Evaluation

Table 2 compares the performance of the risk-free optimistic technique and the conservative simulation for the target system parameters described in Section 2. The average quantum length is the average number of target cycles simulated between barriers and indicates how often synchronization is performed between host nodes. The results clearly demonstrate that the risk-free optimistic technique fails to increase the lookahead and performs worse than the conservative simulation. Simulation of applications that have a significant communication overhead is mostly conservative since protocol events are simulated conservatively. The net result is that the average quantum length is only slightly greater than the lookahead in the conservative simulation (100 cycles). Synchronization is performed nearly as often and the additional overheads of saving and restoring state and computing the event horizons result in a slightly longer execution time. Water and Sparse have a significant computation component and accordingly, optimistic execution improves the lookahead. However, an increase in lookahead alone is not sufficient to speedup the execution. In particular, optimistic execution results in a higher load imbalance between the host nodes. For example, a node could be executing target instructions until the mega-quantum while all other nodes are waiting at the barrier. The increased lookahead (or parallelism) must outweigh the load imbalance and overheads to achieve speedups.

Table 3 presents the breakdown of the simulation time for Ocean. The results show that the overhead of restoring target state and computing the global event horizon is minimal (about 3% for Ocean and less than 10% for all the applications that we simulated). Saving target registers requires less than fifty instructions and accounts for about 2% of the simulation time. The overhead of saving target memory is harder to segregate since the target memory is incrementally saved as the program is directly executed. Although the time for direct execution has increased by 75 million cycles (from 131 to 206 million cycles), it still accounts for less than 6% of the total simulation time.

Step	Time	% of total time
Conservative Execution	123	3.26%
Protocol Simulation	808	21.44%
Optimistic Execution	83	2.20%
Quantum Synchronization	2581	68.49%
Barrier Synchronization	64	1.72%
Reduction	92	2.44%
Undo	17	0.45%

Table 3: Breakdown of total simulation time with risk-free optimistic simulation.

The table shows the breakdown of simulation time when simulating Ocean. *Time* is in millions of host cycles. Conservative Execution refers to the direct execution of the target program instructions until the checkpoint. Optimistic execution refers to the direct execution of target program instructions beyond the checkpoint. Protocol simulation includes simulation of the target cache and target directory events. Quantum Synchronization refers to the time spent waiting for nodes to arrive at the barrier and for the the network to deliver all the messages at the end of each quantum. Barrier Synchronization is the time taken for nodes to complete the barrier after all messages have been delivered and the last node has entered the barrier. Reduction is the time spent in calculating the new event horizon and Undo refers to the time spent unrolling the target registers and memory. The target program was executed for 16 million target cycles.

4 A Time-Warp-like Optimistic Technique

In this section we present an aggressive technique that executes both target program instructions and protocol events optimistically. We call it Time-Warp-*like* since a few target synchronization operations such as *swap* are simulated conservatively.

4.1 Rolling Back Protocol State

Processing protocol events optimistically poses new problems: protocol events send host messages in order to simulate the parallel machine. Mechanisms would hence be required to undo incorrectly sent messages and incorrectly simulated protocol events. We implement the classical solution proposed by Jefferson [8] and send *anti-messages* that annihilate incorrectly sent messages.[2] All protocol events essentially perform one or more of the following actions:

- Modify the state of a block (either in the target cache or directory). For example, an invalidation event would mark a cache block *invalid*; a directory event could set a bit associated with the block to indicate that a target node has a cached copy.
- Send one or more messages to other host nodes. For example, an invalidation event would send an acknowledgement message to (the host node that simulates) the directory indicating that the block has been invalidated; a directory event could send a copy of a block to (the host node that simulates) the cache in response to a request.
- Update the logical clock of the hardware that services the event, i.e., cache or directory.

Since directly executed programs can modify any part of their virtual address space, copying the entire state of the cache or the directory would be prohibitively expensive. Protocol events, however, modify memory in block sizes (typically 32-128 bytes). It makes sense, therefore, to save the state of the target block that was modified by each optimistically simulated event. Analogously, restoring the target state is also done incrementally and is a four-step process:

1. Restore the state of each directory entry that was modified optimistically. For instance, this might result in sending an *anti-response* message to the node that requested a copy and clearing a bit to restore the bit vector of owners.
2. Restore the state of all optimistically modified cache blocks. For instance, this could involve marking a cache block as *valid* to undo an invalidation event.
3. Restore the state of the memory modified by directly executing the target (Section 3.2).
4. Copy the target program registers from the checkpoint buffer (Section 3.1).

This technique requires about 4K bytes of memory for saving protocol state in addition to the memory required to save target processor and memory state. The additional memory is still less than 1% of the memory used by an application that has a data set size of 1MB.

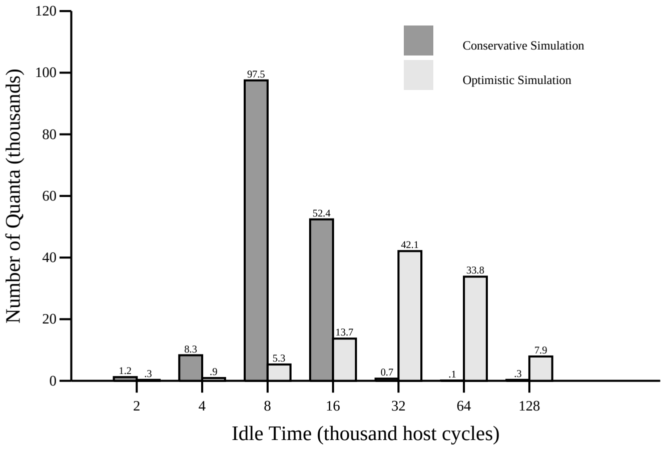

Figure 2: Histogram of idle times.

This figure shows a histogram of the average host idle time in each quantum when executing Ocean. The idle time is the time spent by a host node waiting for other nodes to synchronize. In the conservative scheme the average idle time is mostly between 8,000 and 16,000 cycles while the optimistic scheme results in an average idle time mostly between 32,000 and 64,000 cycles.

4.2 Performance

Table 4 compares the performance of the aggressive optimistic technique and the conservative simulation. For the sake of convenience, we reproduce the time taken for simulation using conservative simulation from Table 2. The results clearly demonstrate that the conservative simulation outperforms the aggressive simulation. The increase in average quantum length denotes that the aggressive simulation is able to synchronize less often. The performance of the aggressive simulation is sensitive to the mega-quantum length. Large optimistic windows increase the scope for useful work to be done before synchronization and hence allows the simulation to synchronize less often (up to 60% less often). However, this comes at a high cost; the probability of an event being simulated incorrectly increases greatly as nodes move further from the global virtual time. Undoing incorrectly processed events leads to an avalanche of anti-messages. For example, for Ocean and Sparse the message traffic increased by up to 66% due to the aggressive simulation. Computation-bound applications such as Water do not suffer from the effect of numerous anti-messages. Instead, such applications tend to rollback and redo their computations. For example, with Water the time spent in executing instructions in the target program increased by up to 300%. Figure 2 illustrates the difference in load imbalance using a histogram of idle times. The load imbalance could be reduced if a host node can detect that one or more host nodes are waiting at the barrier and stop optimistic simulation. Unfortunately the CM-5 does not

[2] Although anti-messages are generated immediately, they are buffered and *sent* only when the logical clock sweeps past the timestamp of the incorrect message without regenerating the same message (i.e., we implement *lazy cancellation*).

Application	Procs	Conservative Simulation Time (million cycles)	Aggressive Optimistic Simulation Time (million cycles)	Average Quantum Length (cycles)	Slowdown
Ocean	8	7097	14867	187	2.09
Ocean	16	4398	9951	173	2.26
Ocean	32	3245	8509	154	2.62
Sparse	8	1775	3406	195	1.92
Sparse	16	2098	4023	186	1.92
Sparse	32	3329	6012	193	1.80
Water	8	6733	11090	248	1.65
Water	16	5100	9876	244	1.93
Water	32	4029	8805	238	2.18

Table 4: Performance of conservative vs. aggressive optimistic simulation.

This table shows the simulation times, the average quantum length and the slowdown of the optimistic technique when compared to the conservative simulation. The mega-quantum length for Ocean and Sparse was set to 200 cycles while the mega-quantum length for Water was set to 300 cycles. While the simulation times refer to the *host* cycles (in millions), the average quantum length refers to the *target* cycles.

provide a fast broadcast mechanism that can enable all host nodes to synchronize as soon as the first node enters the barrier.

We performed experiments using four other shared-memory benchmarks (*Barnes*, *Cholesky* and *Mp3d* from the SPLASH benchmark suite [17] and a parallelized version of *Appbt* [1]) and obtained similar results. We restricted the presentation to three applications in the interest of brevity.

5 Scenarios Favorable to Optimistic Simulation

The results presented in the earlier sections demonstrate that for the default WWT assumptions (i.e., no target network simulation, constant target network latency of 100 cycles), the conservative technique has adequate lookahead to offer better performance than both optimistic techniques when executing our shared-memory applications on the CM-5. We now consider two scenarios that are more favorable to optimistic simulation.

Less Lookahead Due to Network Simulation: Without network simulation, the quantum length was equal to 100 cycles for the conservative algorithm and $\geq$ 100 cycles for optimistic algorithms. Depending on the desired accuracy of simulation of the target network contention and topology, the quantum length must be less than or equal to the message latency. The fixed network latency assumption results in an error of over 20% in several cases [2]. Reducing the quantum length results in a further increase in the synchronization cost. Since optimistic techniques improve the lookahead and reduce the frequency of synchronization, they may perform better if the network simulation messages are not rolled back frequently.

Greater Host Message Latencies: Parallel systems such as a network of workstations are becoming popular since they provide low-cost alternatives to the current generation of parallel machines (such as the CM-5). We expect many of these systems to have high latency messages (10-100s of μs) and little or no hardware support for synchronization. The CM-5, on the other hand, provides fast messages and hardware support for fast barriers and reductions (the latencies of these operations are all less than 10μs on the CM-5). Hyder and Wood study the implications of latency and synchronization tradeoffs using a variety of applications [7].

How would high latency networks and no hardware synchronization affect the trade off between conservative and optimistic simulation? Both techniques must ensure that all messages sent during a quantum are received before the beginning of the next quantum. In the absence of synchronization hardware, each host node must send an acknowledgment message for each message that it receives and perform a simple software barrier once all acknowledgments have been received [7]. On a host with a network latency of 100μs, synchronization at the end of the quantum would be an order of magnitude more expensive than on the CM-5. This favors optimistic techniques since they need to incur the high cost of this software synchronization less often. Unfortunately, this saving does not come for free—the costs of rolling back incorrect messages also increases. Fortunately, this will not be a dominating factor in applications that do not communicate often. Figure 3 illustrates that for applications with a low communication overhead, optimistic techniques may be a better choice on future parallel systems.

6 Related Work

This paper presented techniques to integrate direct execution with optimistic simulation and studied the performance of three simulation techniques. Unger et al. [21] present an incremental state saving scheme in the Jade simulation environment. A state manager exports an interface to the application and calls this interface before each change of a block of state. The backtrace of memory snapshots are saved in a buffer that is unwound on rollback. Incremental state saving is performed in SPEEDES using two techniques [19].

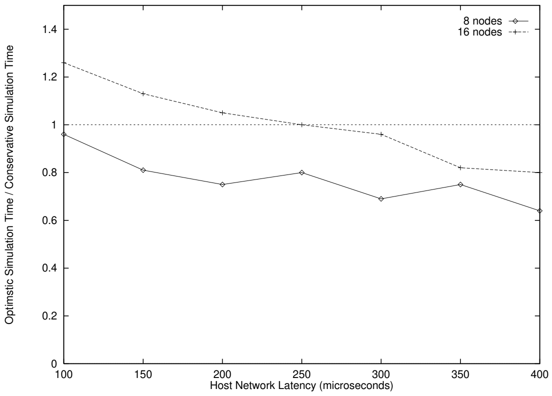

Figure 3: Effect of host network latency on simulation time.

This figure compares the performance of conservative and optimistic simulations as the host network latency is increased. All synchronization operations are performed in software. The application program was Water running on 8 and 16 host nodes. Points above 1 indicate that the conservative simulation is better while points below 1 indicate that the optimistic simulation performs better. With 8 nodes, optimistic simulation is always faster, while with 16 nodes it performs better when the latency is more than 250 μs.

In the delta exchange method event processing is divided into two steps —The first step does the basic event processing while the second step exchanges the new state values. In the rollback queue mechanism the C++ assignment operators are overloaded to automatically save state information.

Shah et al. [16] simulate a shared-memory target machine on a shared-memory host. Their technique is optimistic with respect to timing correctness and reconciliation is performed only at synchronization points. Rollback of the target program is never required because the underlying host machine keeps the memory consistent and only data-race-free programs are executed. However, only the target cache is simulated and the technique has been shown to scale only up to 8 processors. WWT models the parallel machine more accurately and could use the above techniques when the host is a shared-memory machine.

Falsafi and Wood [4] propose simulating multiple target nodes in a single host node. The advantage of this approach is that it reduces load imbalance. Unfortunately, less memory is available to each target node and architects may be restricted to using smaller input data sets.

7 Extensions and Future Work

The most important deficiency of our work is a precise understanding of why optimistic simulations performed worse than the conservative technique. This may enable us to refine optimistic schemes and eliminate the drawbacks of the two techniques that we studied. We have also not attempted to tradeoff accuracy in the simulation for performance. We find that incorrect computations almost always affect the timing behavior of the system being simulated, but rarely affect the functional behavior. Introducing approximations may reduce rollbacks of target program execution and message sends.

We have used a simple algorithm for computing the global virtual time(GVT). This algorithm requires that all host nodes perform a barrier before the GVT can be computed. Researchers have recently proposed efficient algorithms to perform global synchronization with optimistic simulation [11]. Since a large fraction of the simulation time is spent in global synchronization, incorporating the new algorithms would improve the performance of optimistic simulation.

Finally, we are in the process of porting WWT to a cluster of workstations connected by a Myrinet switch. The network latency of this host system is about 100μs and it has no synchronization hardware. We expect that this implementation would give us new insights and help us better understand the performance of various parallel simulation techniques.

8 Conclusions

This paper presented new techniques to integrate direct execution of a parallel application and optimistic parallel simulation. We used checkpointing and incremental logging to correctly save and restore the state of directly executed program executables. We used these techniques to develop two optimistic strategies for parallel simulation that represent the ends of the spectrum of the degree of optimism. A risk-free technique executed only target program instructions optimistically and resorted to a global virtual time calculation before simulating protocol events. An aggressive technique used a Time-Warp-like algorithm to simulate protocol events and send messages optimistically.

We compared the performance of these techniques with the conservative simulation algorithm. We found that for the shared-memory applications that we executed on a CM-5 host, the conservative technique offered better performance. The behavior was similar for three different target system sizes—8, 16, and 32 nodes and three different application programs each having a different communication pattern. Optimistic simulations are plagued by two main problems when used for simulating parallel machines using program executables:

- Simple optimistic techniques that avoid the complexity of undoing protocol events and message sends are unable to improve the lookahead in the simulation. The additional overhead accompanied by little gains make them perform up to twice as slow as the conservative simulation.

- Aggressive optimistic techniques improve the lookahead and are able to reduce the frequency of synchronization. However, these gains come at a very high cost; lowering the costs by restricting the size of the optimistic window also relinquishes chances of increasing lookahead. We found that the Time Warp like technique performs up to 2.5 times slower than the conservative simulation.

Several researchers have reported successes with using optimistic simulation [5, 12, 22]. The two notable reasons for the sharp contrast in our conclusions are:

- Our workloads are directly executed programs. Previous studies of simulation strategies have been mostly performed using synthetic workloads, not program executables.
- Our target system is a parallel machine with an interconnection network that provides full connectivity. An implication of this is that a rollback in a node could potentially affect any other node in the system within a logical time equal to the latency of the target network.

We surmise that optimistic techniques may perform better when the host machine has a high network latency (100s of μs), no hardware support for synchronization and the parallel application being executed does not communicate often. Unfortunately, it is debatable whether parallel simulation itself is cost-effective with such host system parameters. Decreasing the quantum length for detailed simulation of the target network may also favor optimistic simulation when the network simulation messages are not rolled back frequently.

Acknowledgment

We thank Doug Burger, Babak Falsafi, Alain Kägi, Rahmat Hyder, Steve Reinhardt and David Wood for helping us understand WWT and for useful suggestions that improved this paper. David Wood contributed to the genesis of some of the optimistic techniques described in this paper. We thank James Larus for generously supporting EEL, Rahmat Hyder for providing code that simulates high latency host machines and the anonymous referees for comments on an earlier draft of this paper. We thank David Wood and Steve Reinhardt for pointing out an error in the network simulation experiments. We thank all the members of the Wisconsin Wind Tunnel project for technical support and encouragement that made this research possible.

References

[1] David Bailey, John Barton, Thomas Lasinski, and Horst Simon, *The NAS Parallel Benchmarks*, Technical Report RNR-91-002, Revision 2, Ames Research Center, August 1991.

[2] Doug Burger and David A. Wood, *Accuracy vs. Performance in Parallel Simulation of Interconnection Networks*, International Symposium on Parallel Processing, April 1995.

[3] Helen Davis, Stephen R. Goldschmidt, and John Hennessy, *Multiprocessor Simulation and Tracing Using Tango*, In Proceedings of the 1991 International Conference on Parallel Processing (Vol. II Software), pages II99–107, August 1991.

[4] Babak Falsafi and David A. Wood, *Cost/Performance of a Parallel Computer Simulator*, In Proceedings of PADS, 1994.

[5] Richard M. Fujimoto, *Performance of Time Warp Under Synthetic Workloads*, Proceedings of the SCS Multiconference on Distributed Simulation, January 1990.

[6] Richard M. Fujimoto, *Parallel Discrete Event Simulation,* Communications of the ACM, 33(10):30-53, October 1990.

[7] Rahmat Hyder and David A. Wood, *Synchronization Support for Networks of Workstations*, In Proceedings of the International Conference on Supercomputing (ICS), 1996.

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

[9] James R. Larus and Eric Schnarr, *EEL: Machine-Independent Executable Editing*, Programming Languages Design and Implementation (PLDI), 1995.

[10] David Nicol, *Conservative Parallel Simulation of Priority Class Queuing Networks*, IEEE Transactions on Parallel and Distributed Systems, 3(3):398–412, May 1992.

[11] David Nicol, *Global Synchronization for Optimistic Parallel Discrete Event Simulation*, Proceedings of the seventh workshop on Parallel and Distributed Simulation, July 1993.

[12] Presley, M., Ebling, M., Wieland, F., and Jefferson, D. R., *Benchmarking the Time Warp Operating System with a computer network simulation,* In Proceedings of the SCS Multiconference on Distributed Simulation, 21, 2 (March 1989), pp. 8-13.

[13] Steven K. Reinhardt, Babak Falsafi, and David A. Wood, *Kernel Support for the Wisconsin Wind Tunnel*, Proceedings of the Second USENIX on Microkernels and Other Kernel Architectures, September 1993.

[14] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood, *The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers*, ACM SIGMETRICS, 1993.

[15] P. Reynolds, *A Spectrum of Options for Parallel Simulation*, Proceedings of the 1988 Winter Simulation Conference, pages 325-332.

[16] Gautam Shah, Umakishore Ramachandran, and Richard Fujimoto, *Timepatch: A novel technique for the parallel simulation of multiprocessor caches*, TR-94-52, GIT, October 1994.

[17] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta, *SPLASH: Stanford Parallel Applications for Shared Memory*, Computer Architecture News, 20(1):5-44, March 1992.

[18] Jeff S. Steinman, *SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation*, International Journal in Computer Simulation, Vol. 2, Pages 251-286.

[19] Jeff S. Steinman, *Incremental State Saving in SPEEDES using C++*, In Proceeding of the 1993 Winter Simulation Conference, Pages 687-96.

[20] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, 1991.

[21] Brian W. Unger, John G. Cleary, Alan Covington, and Darrin West, *An External State Management System for Optimistic Parallel Simulation*, In Proceedings of the 1993 Winter Simulation Conference.

[22] Wieland, F. et al., *Distributed combat simulation and Time Warp: The model and its performance*, In Proceedings of the SCS Multiconference on Distributed Simulation 21, 2 (March 1989), pp. 14-20.