Performance of a Riskfree Time Warp Operating System

Steven Bellenot
Mathematics Department
Florida State University
Tallahassee, FL 32306
bellenot@math.fsu.edu

ABSTRACT

An optimistic method of synchronizing parallel discrete event simulations can be risky, that is, it could send (positive) messages (events) before they have been committed. Often a risky method will use anti-messages (negative messages) to cancel the wrong positive messages which have incorrectly sent. A riskfree method is more conservative and does not send messages until it is sure they are correct. The Time Warp Operating System (TWOS) was designed and implemented to use anti-messages. A riskfree version of TWOS is implemented and its run time is compared to TWOS on the standard TWOS benchmarks. The poor performance of these standard benchmarks is analyzed. Performance of the riskfree TWOS was highly dependent on the amount of lookahead in the simulation. Good lookahead was required for even reasonable performance. Limitations of this riskfree TWOS implementation caused the creation of Tracker, a simulation of the riskfree simulation. A simulation is constructed which has a speedup of n for Time Warp, but only a speedup of 2 for riskfree methods.

Introduction

As usual a simulation is divided into simulation objects which schedule events for each other via time-stamped messages. All of an object's variables are stored in a data structure called its state. Casualty requires that these events be executed in increasing virtual time order. Parallel simulation methods are usually divided into conservative and optimistic methods. In either case, the simulation objects are divided over the available nodes (processors). The main different between this two methods is that optimistic methods always execute ready events while conservative methods do not execute an event until it is certain no earlier event will arrive.

An optimistic method must handle out of order messages (such a message is called a strangler). When a strangler arrives, a mechanism called rollback is used to restore causality. Old copies of an object's state are saved so that rollback can restore the object to an earlier time. Rollback allows one to undo out of order event executions. Conservative and sequential methods need not store old copies of object states, but have other liabilities (see [F90], which is a good introduction to parallel discrete event simulation).

An optimistic mechanism can also be *risky* in the sense of [R88]. Execution of event can generate events (messages) for other objects. Risky simulation methods immediately send these messages. Thus an object executing in a wrong future could send a incorrect message (positive message) which will need to be unsent. A risky method must provide a mechanism, often anti-messages or negative messages, to unsend an incorrect (positive) message. When a negative and positive message meet in an object's queue they annihilate each other and rollback the object. A more conservative optimistic method can be *riskfree* just by not sending messages until it is certain they are correct.

The Time Warp Operating System (TWOS) is the operating system implementation of Time Warp [JBW87] done at the Jet Propulsion Laboratory (JPL). Time Warp is a method of optimistically synchronizing parallel discrete event simulations [J85]. TWOS is also risky, it sends messages as soon as they are generated.

Most optimistic simulations which have been implemented have been risky with the exception of SPEEDES ([S91] and [S92]). (The name SPEEDES is changed to breathing time buckets in [S92], but we will use the old name in this paper.) Steinman in [S92] adds an aggressive cancellation version of Time Warp to the riskfree SPEEDES and compares their performance. Here we do a similar exercise with TWOS. We implement a riskfree version of TWOS and compare it to the regular risky TWOS. Riskfree TWOS is not synchronous like the riskfree method in SPEEDES.

Generally, the performance of the riskfree TWOS was poor. The bulk of this paper is explaining why the performance was so bad. We show that some simulations like BeRisky below will not run well with any riskfree method. We find that good riskfree performance for the JPL benchmarks requires good lookahead in the simulation. The *lookahead* of an object is the minimum virtual receive time minus the virtual send time of each message (event) it schedules. A simulation has good lookahead if the lookahead value is large compared to the usual event spacing. Riskfree TWOS ran best when the list of uncommitted events was long (in terms of CPU time) and new events were scheduled at times which put them near the end of these uncommitted events. (Or in the language below, successive event horizons must contain lots of simulation work.) This particular implementation of a riskfree TWOS is slowed by using the usual TWOS code to compute GVT. To factor out this limitation, Tracker, a simulation of riskfree simulations was created.

The Event Horizon

The event horizon is a term from SPEEDES, but it has meaning in a sequential simulation. At a given simulation time T, the *event horizon* is the virtual time of the earliest event generated by an executing an event at virtual time T or greater. The main loop of SPEEDES executes object until the next event horizon is known. Once all nodes know the new event horizon, all output messages with send time less than the new event horizon are sent. Due to the nature of its sending mechanism, all nodes will know when all messages have been safely received. The object execution resumes to determine the next event horizon. In SPEEDES, the sequence of event horizons is repeatable and can be determined from a sequential run. The calculating the event horizon has a lot in common with calculating global virtual time (GVT) in TWOS. But the next event horizon calculation requires the all events with virtual time less than the new event horizon be completed, where as a GVT calculation can be done at anytime.

Tracker

In light of the limitations of our riskfree TWOS we designed and implemented Tracker. Tracker simulates a no overhead SPEEDES simulation. Tracker has one delay parameter which makes it also a simulation of our riskfree TWOS.

Tracker requires two steams of input data which we collected in to separate runs of the sequential simulator. One run just collected a list of all messages (events) send during the simulation. Just four fields were needed, sender, send time, receiver, and receive time. The other run collected event execution times, the wall clock time, the event's virtual time and the name of the executing object. (This needed a separate run so that the message logging time was not included into the event execution time.)

Since no wrong events are sent in a riskfree method, the only (possible) wrong calculations would be when a correct event is executed early. We have the execution time of this event when it is executed in the correct order and we assume that this out of order event execution CPU time is the same for early (wrong) executions as it is for the correct executions. This could be false for some simulations, but it seems reasonable.

The second assumption deals with calculation of event horizons. Since Tracker input contains a log of all messages sent by the simulation, Tracker can figure out the sequence of event horizons. However, the riskfree parallel can't compute the next event horizon H until after all events with virtual time less than H have executed. There is a delay between the real time the last event less than H completes and the real time when all the nodes know H is the next event horizon. We all this delay the *commit delay*, and we made it a parameter to Tracker. This delay is similar to the GVT calculation time in our riskfree TWOS.

Tracker assumes there is no other run time overhead. Tracker interrupts events in progress if an earlier event becomes available. Tracker just keeps track of which event is executing where, which allows the collection of node idle times as well as overall run times. A the list of Tracker assumptions is given below.

Tracker Assumptions

1. The event execution real time is the same for early (wrong) executions as it is for the correct (sequential) event execution time.
2. Event horizons are calculated "exactly" at the instant they can be known. The delay to spread this event horizon to the nodes is the Tracker parameter commit delay.
3. There is no synchronization overhead and messages have zero communication delay time.

Effect of the commit delay parameter in Tracker

Between the real time the event horizon is known and the time all the nodes learn the new event horizon could be called the twilight zone. While in the twilight zone, each node is executing events which could be rolled back. If there is not enough events available to execute, this twilight zone of time will be idle CPU time. That is if the commit delay parameter is large enough, then Tracker (and a riskfree simulation) will run out of simulation work to do while waiting for the new event horizon. Idle CPU time, if long enough, would rob the simulation of any speed up.

The estimated commit delay for the current riskfree TWOS implementation is very high, around $30-60$ milliseconds for $10-70$ nodes (see Ping below). This long commit delay can lead to idle CPU time. To get around this long possibly idle time, the simulation designer can do several things. If event execution time is long then few events would be needed to keep the processors busy during the twilight zone. Another approach is to have lots of events that can be executed in the twilight zone. However the best design method seems to minimize the number of event horizons with good lookahead. (See Bank below.)

The BeRisky simulation

The BeRisky simulation consists of n objects named 0 through $n-1$. Each object only sends messages to itself and every event execution takes the same amount of real CPU time. At time $i*n+j$, object j sends a messages to itself at time $i*n+j+1$. And at time $i*n+j+1$ object j sends a message to itself at time $(i+1)*n+j$. Time Warp (and conservative methods) have no problem (in theory) of obtaining the full n-fold parallelism of BeRisky. However, at each virtual time T, the next

event horizon is $T+1$. Hence a riskfree simulation could at most obtain only a two fold parallelism. As a model this simulation is really the same as n objects which at time T just send a message to itself at a time $T+L$ in the future. Here all the objects are synchronized to the virtual time axis and in theory the riskfree method could obtain the n-fold parallelism of the new simulation. (See also Pucks below.)

TWOS – Riskfree TWOS trade offs

The cost of anti-messages in TWOS includes increased message traffic, increased memory use to store the anti-messages and the increased logic which is needed to handle the two kinds of messages (positive and negative). The advantage of anti-messages to TWOS include faster delivery of messages and possibly fewer global synchronization's (global virtual time calculations versus event horizons). An advantage of riskfree methods which is not shown in the benchmarks below is the ability of riskfree TWOS to throttle self-propelled memory hog simulation objects. Note that the synchronuous nature of SPEEDES is advantage over TWOS which the riskfree TWOS does not share.

Riskfree TWOS implementation

The two main changes to TWOS needed to make it riskfree were to eliminate anti-messages and changes to GVT. The original code modifications was done by a student Pi-chiang Chang. Actually anti-messages were kept and positive messages were removed from the code. In TWOS anti-messages were already put in output queues. Riskfree rollback is similar to TWOS with aggressive cancellation, except anti-messages are destroyed rather than sent. Finally, the commit operation at a GVT update sends the anti-message rather than destroys it. Anti-messages have priority in the low level message sending in TWOS on the butterfly.

The changes to GVT are less clear cut, and we tried several approaches. In TWOS, GVT is calculated every second. This was not often enough for the riskfree version. Eventually, each riskfree node will run out of events to run and the node will be idle until the next GVT. The next GVT will release the messages which can be committed. A periodic GVT with too long a period wastes too much CPU time and the performance is bad.

We first modified the GVT periodic machinery so that the period was in milliseconds rather than seconds. However, doing GVT too often also has its costs. If the new GVT is the same as the old GVT there is no need to try to commit any output messages. But each GVT calculation required searching all the output queues and the several other places TWOS can put messages in transit. Now GVT calculation isn't a large part of the run time in TWOS but you only do it every second in TWOS. Finding a good GVT period for any of the benchmark simulations and for a multiple number of nodes proved impossible.

The original design of riskfree TWOS changed GVT from a regular periodic activity to a GVT on demand system. Each node kept track of its own idea of the next event horizon. When a node ran out of objects to execute or, more likely, when it started executing objects at virtual times later than its current idea of the next event horizon, the node would request a GVT calculation. However, on multiple node runs, it was not unlikely for one of the nodes to be idle immediately after a GVT update. This node would immediately request a GVT calculation. On those times when a single node had little or nothing to do after a GVT calculation it would become a pest. Such a node would constantly request GVT calculations.

A modified request scheme was implemented. Since in TWOS a GVT calculation is always started by node zero. Node zero became the only node which would start a GVT on demand. When a node, say number $i > 0$, wanted to request a GVT, it would instead pass its request along with its current estimate of the next event horizon to the node numbered $(i+1)/2 - 1$. This lower numbered node would incorporate the received upper bound on the next event horizon into its own current estimate. If the received estimate was lower than the receiving node's estimate, then the receiving node might discover it was executing events beyond its new idea of the next event horizon and in turn request a GVT.

Since SPEEDES is synchronous, it can tell when there are no messages in transit and the calculation of the next event horizon is more straightforward. However this scheme cannot be used in TWOS. TWOS has no easy way to determine if all messages have been received. Since GVT happens so infrequently, the GVT implementation in TWOS is not optimized for speed. Indeed. For example, each GVT message goes through the complete message allocation process. Timing tests showed that it took longer to allocate a GVT message then it took to send a GVT message and have it received on the receiving node.

Note that a riskfree TWOS simulation could do more GVT calculations then strictly required to compute event horizons. Thus riskfree TWOS may do more event horizon calculations then SPEEDES. However, for our simulations this is not unreasonable to estimate the number of GVT calculations with the number of event horizons.

Benchmark Performances

All measurements were made on a BBN Butterfly GP-1000 owned by the Jet Propulsion Laboratory. TWOS version 2.7 was the baseline. The benchmarks are organized by worst (for riskfree) first. The three

standard JPL benchmarks of Pucks, STB88 and Warpnet were measured as well as two artificial simulations Ping and Bank.

Ping Benchmark

Ping has also been called Pong. Ping is not really a simulation but a debugging aid. There are two objects in Ping, named ping and pong. All ping (pong) does is schedule an event for pong (ping) at time now plus one. The simulation has one message which bounces back and forth. Figure 1 shows the run times of Ping when run by TWOS (the flat line at the bottom), Ping when run by riskfree TWOS and a logarithm curve which fits this second plot. Note that there is no parallelism available in Ping, it is a sequential simulation. When run in TWOS, Ping has no anti-messages nor rollbacks. For SPEEDES and riskfree TWOS, every message is a new event horizon. This Ping sent 2500 messages and so there were 2500 event horizon calculations in the riskfree TWOS. TWOS on the other hand does almost no GVT calculations since PIng runs so fast. Ping is used to estimate the GVT calculation time (roughly $14ln(x)$ milliseconds, where x is the number of nodes) and hence the event horizon commit delay time for use in Tracker. The logarithm shape of Figure 1 is consistent with GVT calculation times in [B90].

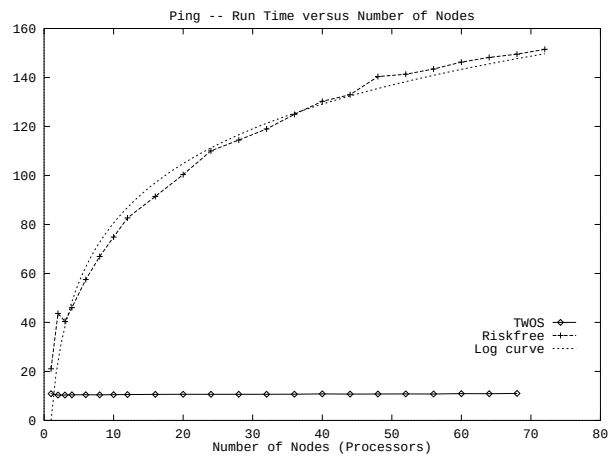

Figure 1. Ping run times.

Pucks Benchmark

Pucks has been a standard JPL benchmark since it was introduced [HBD89]. Pucks has 304 objects simulating colliding pool balls and usually runs for 400 simulated time units where it has 362824 committed events and 405802 committed messages. Pucks objects average 2.7 milliseconds per event and has a TWOS speedup near 14. Figure 2 shows the comparison in run times of Pucks when run by TWOS and when run by riskfree TWOS. The usual Pucks benchmark runs ten times longer than the version timed here. (The short version was for 40 simulated time units.) The poor performance of the riskfree Pucks was expected due to the way the Pucks simulation is constructed. Below we will explain why pucks is highly sequential when run riskfree.

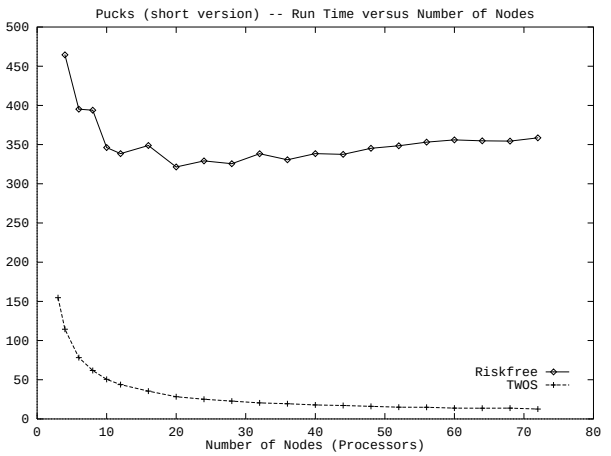

Figure 2. Pucks run times.

Figure 3 shows Tracker run times with Pucks. The length of the Pucks benchmark used in the runs was much shorter than the length of the standard Pucks benchmark, four times shorter than the ones in Figure 1 and forty times shorter than normal. (The very short version was for 10 simulated time units.) Even the case with no commit delay, the idealized no overhead run times are twice as long as an actual TWOS run times. This cuts the speed up by more than half. With a commit delay of 10 milliseconds, speed up drops by almost one fifth. And with the actual riskfree TWOS estimated delays, Tracker Pucks runs slower as the number of nodes increase. So Pucks is unlikely to perform well in SPEEDES either.

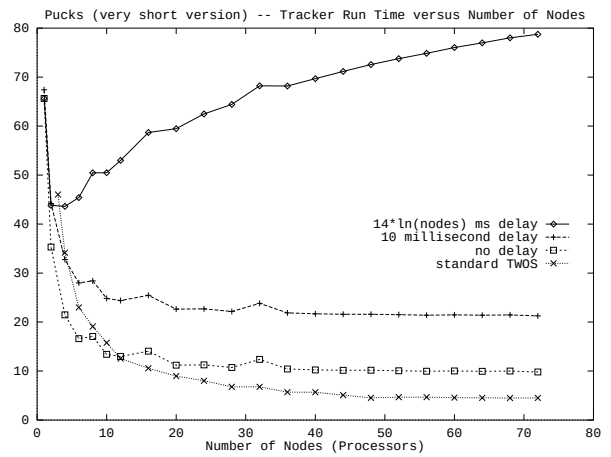

Figure 3. Tracker Ping run times.

What does Pucks do that ruins the riskfree performance? First Pucks has no lookahead. When two puck objects collide, both pucks gain new velocity vectors. Both pucks send information messages containing their new velocities to the sector(s) they are currently in. These information messages are almost for now, they are guaranteed to be the next events in the simulation. The sector(s) respond to these information messages by sending informational messages to its neighboring sectors, cushions and the pucks in its sector. These information messages are also almost for now, they are also guaranteed to be the next events in the simulation. This is a disaster for riskfree performance, there are two event horizons after every collision with essentially no simulation work. Thus a riskfree Pucks essentially sequentializes the pucks collisions losing most of the parallelism.

We believe that Pucks with lazy cancellation TWOS beats the critical path because of lazy cancellation's ability to obtain temporal parallelism (See [RBJ91]). Pucks is known to run $30-60\%$ slower with TWOS with aggressive cancellation [B89] for similar reasons. Although Pucks is a disaster for SPEEDES, apparently other proximity detection models can obtain a high degree of parallelism [S92]. (But note the Time Warp which Steinman uses to compares with SPEEDES uses the less than optimal aggressive cancellation.)

STB88 Benchmark

Often STB88 [WHF89] falls between Pucks and Warpnet as it does here. STB88 is a war game simulation with 380 objects, 389382 committed events and 603472 commited event messages. STB88 objects average 8.1 milliseconds per event execution and gets a TWOS speedup above 26. The results for STB88 is given in Figure 4. Compared with Figures 2 and 5, STB88 is in the middle again. STB88 also has a proximity model (although not exactly the one in Pucks). STB88 events only happen at integer virtual times, and like Warpnet it has events for now plus one (although perhaps not as often).

Warpnet Benchmark

The Warpnet benchmark is described in [HBD89]. Warpnet simulates a communication network with 169 objects, 35027 committed events and 45319 committed event messages. Warpnet objects average 51.5 milliseconds per event execution and has a TWOS speedup of almost 30. We were surprised by the poor riskfree Warpnet performance. Figure 5 shows the run time for Warpnet comparing riskfree TWOS with TWOS. Events in Warpnet only happen at integer virtual times and as Figure 6 shows there is a good deal of parallelism available at each virtual time. However, the lookahead in Warpnet is poor. At most virtual times T, there is an event scheduled for virtual time $T+1$. The poor lookahead and the lengthy GVT calculation times for riskfree TWOS lost this potential parallelism.

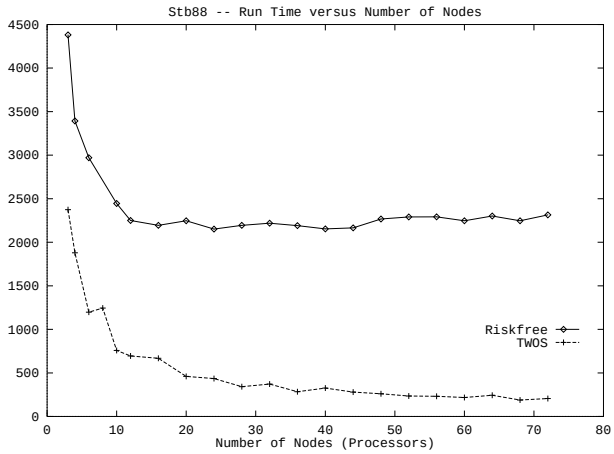

Figure 4. STB88 run times.

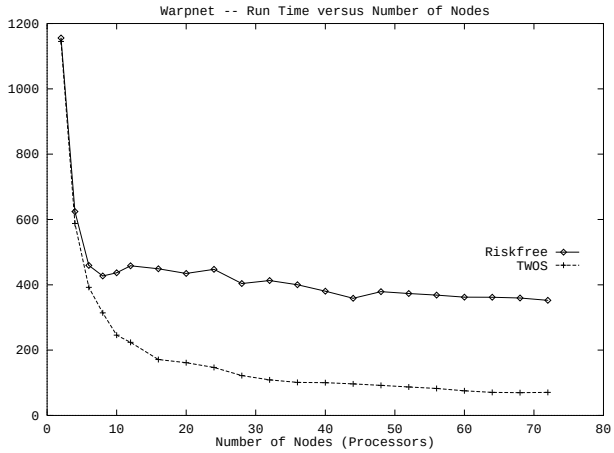

Figure 5. Warpnet run times.

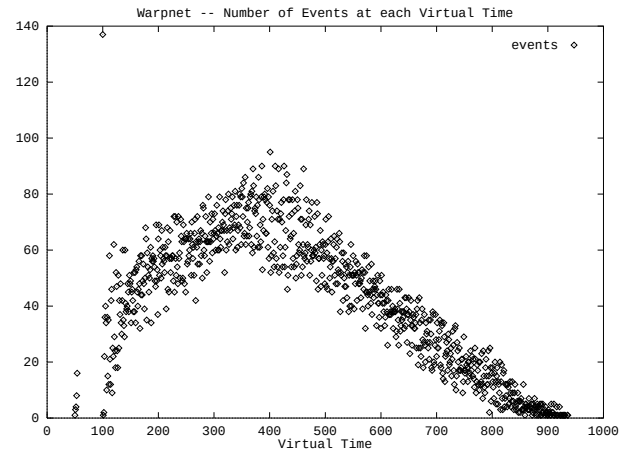

Figure 6. Warpnet objects executing in parallel.

Figure 7 shows the various Tracker speedups for Warpnet, with various commit delay values compared to the real TWOS and riskfree TWOS values. Note that even the idealized no overhead no delay Tracker speedup curve flattens before the actual TWOS speedup curve peaks. Indeed at each increased level of commit delay in the Tracker curve, has the Warpnet speedup curve flatten earlier. When the commit delay is $14ln(nodes)$ the speedup becomes horizonal at about 36 nodes, essentially the same shape as the real riskfree TWOS speedup curve. These Tracker speedup curves were what we expected Warpnet would perform under riskfree TWOS. The flattening of these Tracker speedup curves is due to increased idle time, the simulation runs out of work to do and most wait for the next event horizon for more work.

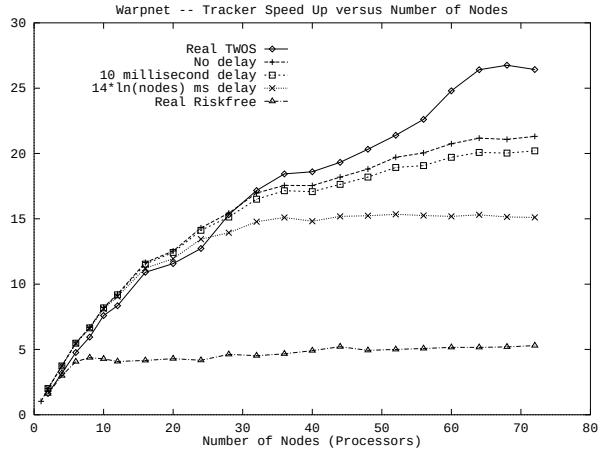

Figure 7. Tracker Warpnet speedups.

Bank Benchmark

The Bank benchmark was introducted in [B92]. Each of the 16 bank objects reacts to a message by generating a message with a random receiver and random exponential interarrivial time with mean 10 (but see below). We modified Bank so to discover if the riskfree TWOS implementation could ever obtain parallelism. There were three ways which we attempted to modify bank to obtain good results with riskfree TWOS. First was message density, message density is the number of active events per object. Bank already had message density as a parameter and we varied this parameter from 5 to 25 active messages per object. (Changing the message density by a factor of S changes the number of events in the Bank benchmark by essentially the same factor S.) Second was event execution time. Bank events are fast taking under a millisecond to execute. Delays loops were added to make this execution time range from 1 to 41 milliseconds. Finally we varied lookahead. The usual bank message sent at time T arrives at $T + D$ where D is an exponential random variable with mean 10. A new bank parameter of lookahead L changed this arrival time to $T+L+D$ where D is now an exponential random variable with mean $10 - L$. This changes the simulation slightly, but the delta (interarrivial) time between send and receive times always has mean 10.

Figure 8 shows the best speedups we obtained for Bank. The message density was 25 and the execution delay is 40 making the average event execution time around 41 millieseconds. The best riskfree Bank speedup was almost 9.5, the same simulation using TWOS got a speedup over 13. Lookahead had the biggest effect on banks riskfree performance. Good lookahead greatly reduces the number of GVT calculations (clicks) made by riskfree TWOS. Figure 9 shows the number of GVT clicks versus lookahead for the various message densities.

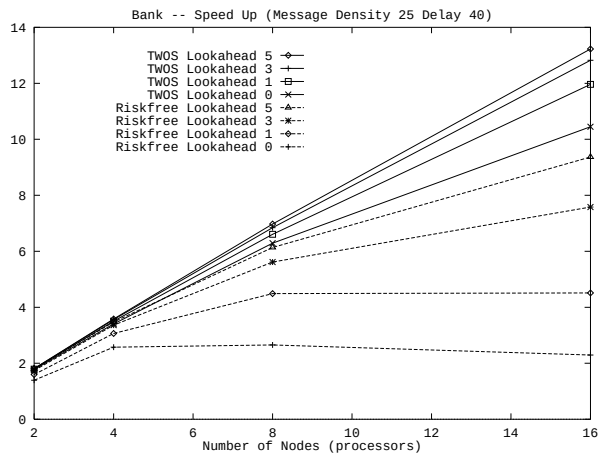

Figure 8. Best Bank speedups.

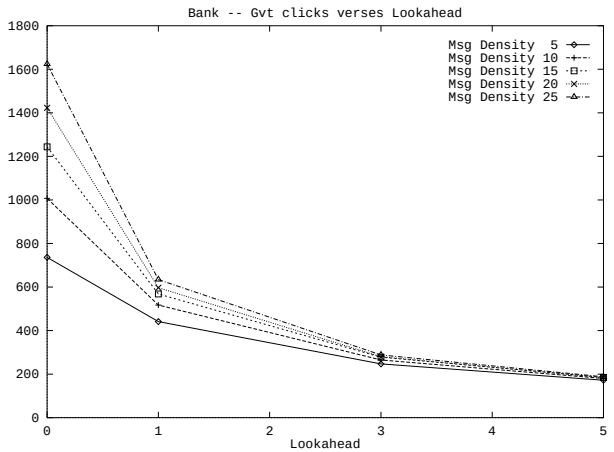

Figure 9. Bank: GVT calculations versus lookahead.

The effects of granularity had significant effects as well. Figure 10 shows the worst performance of Bank

where the delay is 0 and message density is 5. With a delay of zero, the average event execution time is under one millisecond. This small granularity robs the speedup of both methods. (The horizontal and vertical scales of Figures 8, 10, 11, 12 and 13 are all the same.) Moving the delay to 10 makes the average event execution about 11 milliseconds (slightly more than STB88) and improves the speedups (Figure 11).

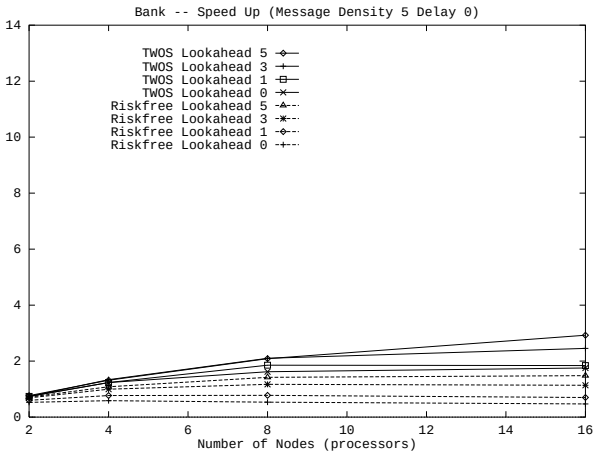

Figure 10. Bank worst speedups.

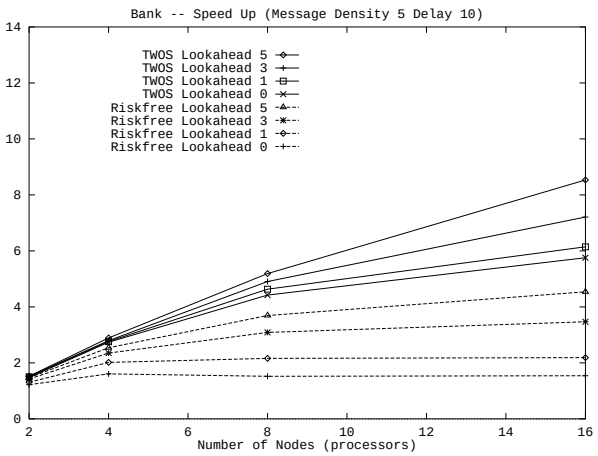

Figure 11. Bank speedups when delay is 10.

Finally we offer two figures showing the effect of message density and delay when the other parameters are at their best values. Figure 12 varies the delay parameter and the big difference between the delay 0 and the delay 10 case is easily observed. Figure 13 varies the message density parameter.

Conclusions

The performance results of the JPL benchmarks with riskfree TWOS were poor. Part of the poor results are due to the design of these simulations, both

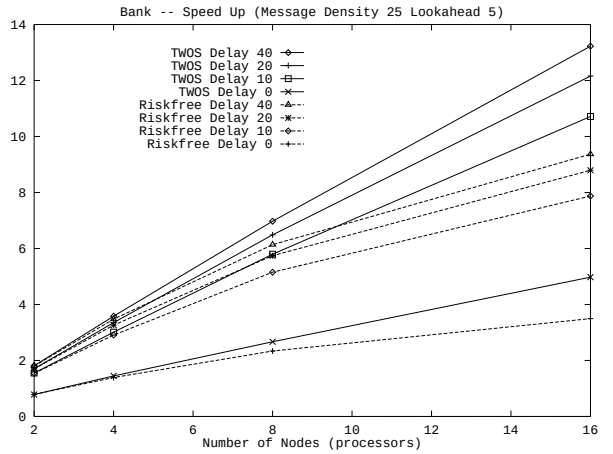

Figure 12. Bank speedups: The effect of delay.

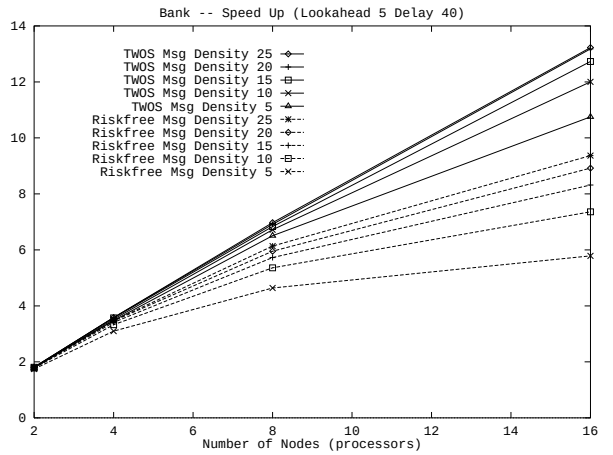

Figure 13. Bank speedups: The effect of message density.

Pucks and Warpnet having poor lookahead (and perhaps STB88 also). Part of the poor results is due to the implementation, not attempting to improve the lengthy GVT calculation times. However, the Tracker simulation shows there are real simulations like Warpnet which TWOS gets good speedup, but which a riskfree simulations will perform significantly worst. And the BeRisky simulation shows a theortical simulation where riskfree is limited to a speedup of 2, while Time Warp gets a speedup of 10. On the other hand, even with its limitations, the riskfree TWOS implementation did get good speedups with variations of the Bank benchmark.

Time Warp with lazy cancellation can obtain good speedups with simulations with poor lookahead. Time Warp can obtain the temporal parallelism available in BeRisky and Pucks. Good lookahead is needed to obtain good performance in riskfree simulation methods. The TWOS benchmark suite, at least as constructed, would likely have poor performance in any conservative simulation method.

References

[B89] S. Bellenot, *Why is Pucks lazy?,* JPL interoffice memo SFB:363-89-005, (1989).

[B90] S. Bellenot, *Global virtual time algorithms* Distributed Simulation, SCS simulation series 22 (1990), 122-127.

[B92] S. Bellenot, *State Skipping Performance with the Time Warp Operating System,* 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS simulation series 24 (1992), 53-61.

[F89] R. Fujimoto, *Performance measurements of distributed simulation strategies,* Trans. Soc. for Compul. Simul. 6,2 (1989), 89-132.

[F90] R. Fujimoto, *Parallel Discrete Event Simulation,* Communications of the ACM, 33 no 10 (1990), 30-53.

[J85] D. Jefferson, *Virtual Time,* ACM Transactions on Programming Languages and Systems, 7 (1985) 404-425.

[JBW87] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J Wedel, H. Younger, and S. Bellenot, *Distributed Simulation and the Time Warp Operating System,* Proc. 12th SIGOPS - Symposium of Operating Systems Principles, 1987, 77-93.

[HBD89] P. Hontalas, B. Beckman M. Di Loreto, L. Blume, P. Reiher, K. Sturdevant, V. Warren, J. Wedel, F. Wieland, D. Jefferson, *Performance of the colliding pucks simulation on the time warp operating systems (Part 1: Asynchronous behavior and sectoring),* Distributed Simulation, SCS Simulation Series 21 (1989), 3-7.

[PEW89] M. Presley, M. Ebling, F. Wieland and D. Jefferson, *Benchmarking the Time Warp operating system with a computer network simulation,* Distributed Simulation, SCS simulation series 21 (1989), 8-13.

[RFB90] P. Reiher, R. Fujimoto, S. Bellenot, and D, Jefferson, *Cancellation strategies in optimistic execution systems,* Distributed simulation 22,1 (1990) 112-121.

[R88] P. Reynolds, *A Spectrum of Options for Parallel Simulation Protocols,* Proc of ACM Winter Simulation Conference, (1988), 325- 332.

[S91] J. Steinman, *SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation,* Advances in Parallel and Distributed Simulation, SCS simulation series 23, (1991) 95-103.

[S92] J. Steinman, *SPEEDES: A Unified Approach to Parallel Simulation*, 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS simulation series 24 (1992), 75-84.

[WHF89] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, *Distributed combat simulation and time warp: The model and its performance,* Distributed Simulation 21 (1989) 14-20.