Cancellation Strategies in Optimistic Execution Systems

Peter L. Reiher

Jet Propulsion Laboratory

Richard Fujimoto

Georgia Institute of Technology

Steven Bellenot

Florida State University

David Jefferson

UCLA

1. Introduction

Computer systems that spread a task across several nodes of a multiprocessor or distributed system face a synchronization problem. Each part of the task may need to interact with other parts at certain points in the computation. Such interactions must be handled in the proper order, or the distributed task performs differently than it would on a single machine, usually resulting in an incorrect computation. Two methods of assuring proper synchronization are the *conservative method* and the *optimistic method*. The conservative method never permits a piece of work to be done until the system is sure that all synchronization has been performed properly. The optimistic method does not wait for assurance that the current status is correct, but performs the piece of work as soon as the node hosting the work is able to get to it.

Conservative methods will never do incorrect work, since they never permit the opportunity to arise. Optimistic methods will do incorrect work. Eventually, they will discover that such incorrect work is wrong. At this point, the incorrect results must be undone, and the work redone in the correct manner. Optimistic systems can extract more parallelism from some applications than conservative systems, at the cost of extra storage and a more complex mechanism. The characteristics of particular types of jobs determine whether conservative or optimistic methods are more appropriate to use.

One area of computation that has responded well to optimistic methods is discrete event simulation. Optimistic discrete event simulation systems are more sophisticated in their use of optimism than any others. In particular, the *Time Warp mechanism* has been used very successfully to achieve high degrees of parallelism and resulting high speedups from many discrete event simulations [Fuji89], [Weil89] [Hont89].

In an optimistic discrete event simulation system, the simulation is divided into several components, called *objects*. Objects interact with each other by sending *messages*. The arrival of a message at an object causes the object to execute an *event*. Events cause the sending of messages and changes in the internal *states* of objects. Events are ordered by *simulation times*. A proper run of the simulation should give the appearance that every event was handled in order of its simulation time.

Incorrect computation can arise for two reasons in an optimistic system. First, messages can be processed out of order. Since different objects reside on different nodes, and since those nodes do not synchronize activities among themselves, one node may be working far in the simulation time future of another node. If a node in the past sends a message to an object on a node in the future, the object receiving the message may have already moved past the message's simulation receive time. Any work done by the object at simulation times before the arrival time of this straggler message may be in error.

Out-of-order messages are the primal cause of errors in optimistic simulation systems, but they can give rise to a second type of error. If an object executes an event out of order, one possible result of the misordered computation is to send an erroneous message, a message that would not have been sent at all if events had been done in the proper order. Alternately, the message sent could contain erroneous data that will be corrected when the straggler message comes in. In this case, the error is an erroneous message in place of a correct message. Both messages will be sent to the same object at the same simulation time, but have different contents.

Erroneous messages can have cascading effects. One out-of-order arrival can cause the creation of a message that contains incorrect data, which can in turn cause the creation of a message that should not have been sent at all. In a correct optimistic system, all erroneous messages will eventually be corrected.

Two methods are used for cancelling incorrect messages in Time Warp systems. *Aggressive cancellation* cancels messages the instant that the system detects that an object has performed an incorrect or out-of-order computation. Any messages sent from that computation are immediately cancelled. *Lazy cancellation* waits until the system is certain that the messages sent as a result of the improper computation will not also be sent by the proper computation.

The time between detection of an incorrect computation and the cancellation of its messages can be quite long in lazy cancellation systems.

Aggressive cancellation systems work on the assumption that most bad input produces bad output. Lazy cancellation systems assume that much bad input produces the same output that good input would produce. Characteristics of the simulation can cause it to favor either aggressive cancellation or lazy cancellation. [Gafn88] demonstrated analytically that lazy cancellation would perform better than aggressive cancellation in many cases. [Lomo88] presented some empirical evidence that lazy cancellation performs as well or better than aggressive cancellation for one simulation on the Jade version of Time Warp.

This paper presents analytic results and performance measurements for both lazy and aggressive cancellation in two independently developed optimistic discrete event simulation systems. The paper examines the relative benefits of the two methods of cancellation, and describes how well and how poorly each method can perform. Unlike previous work on this subject, it presents performance results covering different applications and systems, and uses test applications that demonstrate the strengths of both cancellation mechanisms.

2. The Aggressive Cancellation Mechanism

An optimistic system using aggressive cancellation attempts to correct its mistakes as quickly as possible, at the possible cost of sometimes cancelling correct results. In an aggressive cancellation system, the instant that the system detects that incorrect work is done, any results of that incorrect work are immediately cancelled. This process is called a *rollback*. Consider the example shown in Figure 1.

Object A sends messages to object B. In Figure 1a, object A has received an incorrect message. The resulting event is incorrect, and has caused an incorrect message to be sent to object B. (The message and object A itself are shaded, indicating that the message is erroneous and that A has computed in error.) In Figure 1b, object A has received a cancellation of the incorrect message and a corrected version of that message. Meanwhile, object B has started working on the erroneous message sent from object A. At the moment of the rollback, object A aggressively cancels the message it sent to object B, immediately interrupting object B's incorrect execution. In Figure 1c, object A is performing its correct execution, while object B awaits further messages. Since A aggressively cancelled its message to B, B now has no messages to process, and therefore idles. In Figure 1d, object A has sent the correct message to B, and B can start to execute.

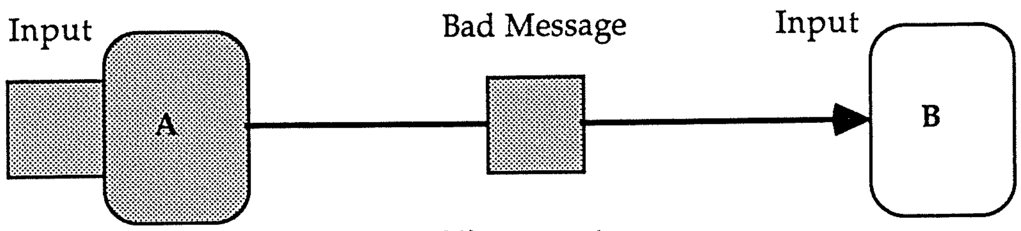


Figure 1a

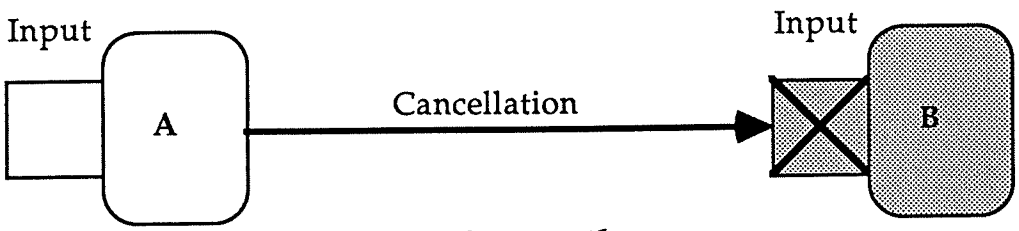


Figure 1b

Input
A
Input
B

Figure 1c

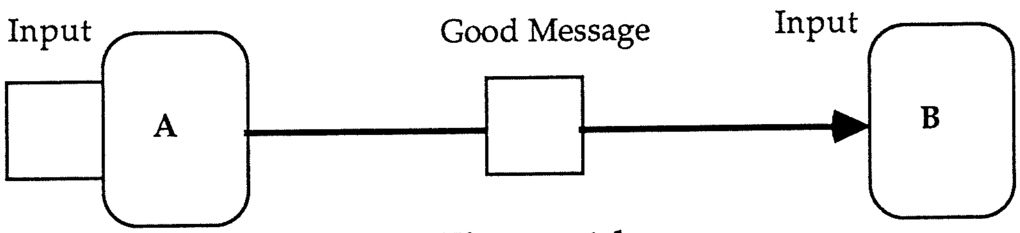


Figure 1d

Figure 1

If the incorrect message originally sent to B is different than the correct message A eventually sends to B, then aggressive cancellation is efficient. The node hosting B does not waste time on an incorrect execution based on an invalid message; perhaps that node has other objects that can be run. On the other hand, if A's second message to B is the same as the first message sent, then aggressive cancellation causes B to wait for A to complete re-execution before B can start work.

The example in Figure 1 shows how aggressive cancellation works in the face of an actual incorrect message. It behaves in a very similar manner when messages arrive out of order. If the early execution of an event at a high simulation time has to be rolled back due to the arrival of a message at a lower simulation time, aggressive cancellation cancels the results of the misordered computation as soon as the earlier message causes the rollback. All effects of the misordered computation are then undone as quickly as the system can manage.

The aggressive cancellation mechanism can be quite simple and efficient to implement. Whenever the system rolls back an object for any reason, any messages sent by that object for the same or a later virtual time are cancelled. The actual mechanism for performing cancellation can vary, depending on implementation, but the mechanism can be invoked simply and immediately under aggressive cancellation.

In some cases, aggressive cancellation will actually send more messages than lazy cancellation. Any time that lazy cancellation would choose not to send a message, because it matches one sent during an earlier execution of the object, aggressive cancellation requires that the message be cancelled and resent. If cancelling a message requires sending a message, then aggressive cancellation will send two more messages than lazy cancellation every time that a message is resent in an unchanged form.

3. The Lazy Cancellation Mechanism

Lazy cancellation waits for a message to be proven incorrect before actually cancelling it. If correct or properly ordered events often produce the same results as incorrect or misordered events, lazy cancellation does not cause the message sent by the incorrect event to be cancelled, and the system need not send the copy produced by the correct event. The object that received the copy of the message sent by the incorrect event can gprocess that message before the correct version of the event even begins execution.

This property of lazy cancellation has an interesting consequence. Distributed Time Warp systems using lazy cancellation can, under certain circumstances, run faster than the critical path of the simulation [Berr86]. Beating the critical path depends on unlikely circumstances, and is not likely to occur in real applications. The important aspect of this property is that lazy cancellation allows some objects to run before all of the work they depend on has been correctly completed.

Figure 2 shows how lazy cancellation works. As in Figure 1, object A has received an incorrect message. As a result of A's incorrect execution, it sends another incorrect message to B, as shown in Figure 2a. In Figure 2b, object A has received the cancellation of the incorrect message and the corrected version. Despite knowing that it earlier processed a possibly incorrect message and sent a second incorrect message as a result, A does not immediately cancel the message it sent to B. Instead, A processes its correct message. Meanwhile, B continues executing with the incorrect message. In 2c, A has finished processing its event. A realizes at this point that the message it sent earlier was incorrect, so it sends B the cancellation of the incorrect message and a copy of the correct message. (The cancellation and correction may not actually be bundled into the same message.) In Figure 2d, B cancels the incorrect message and starts executing the correct message.

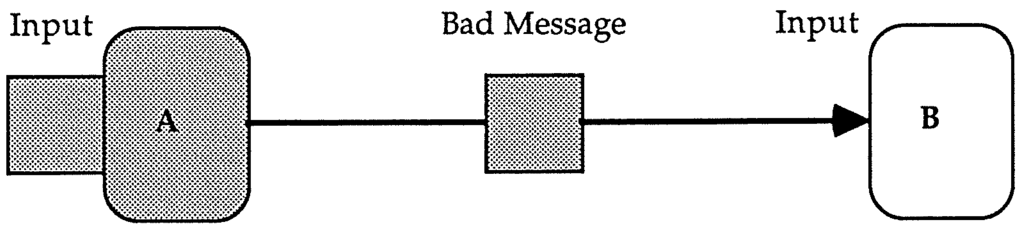


Figure 2a

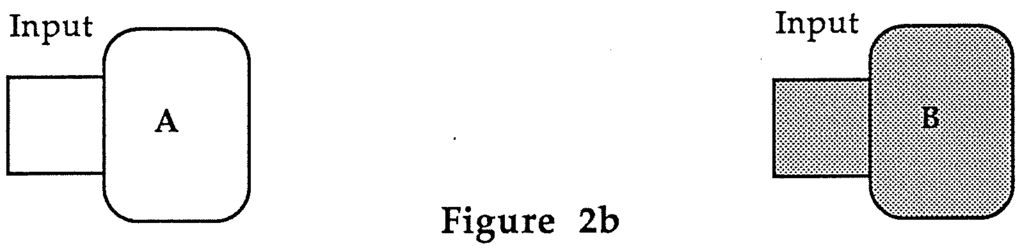


Figure 2b

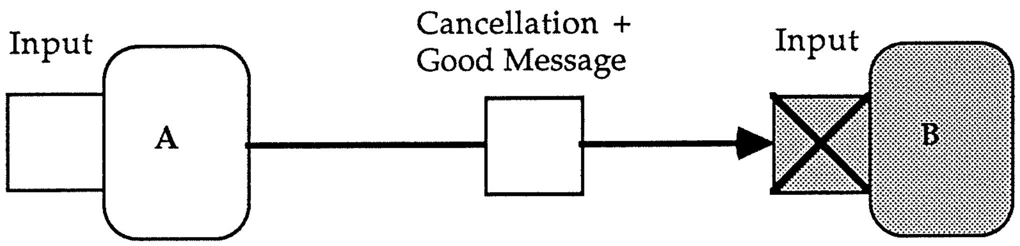


Figure 2c

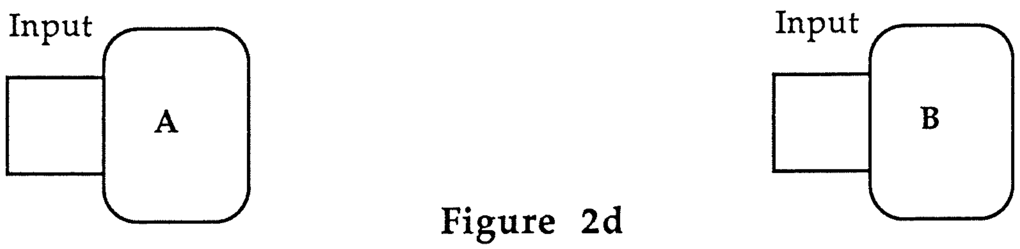


Figure 2d

Figure 2

Figure 2 covers the case in which lazy cancellation loses. Figure 3 shows the course of events when lazy cancellation wins. As in Figure 2a, Figure 3a shows A sending a message to B. While this message is the result of an incorrect execution, it is exactly the same message as would result from a correct execution. In Figure 3b, A gets the cancellation of the original incorrect message and the correct message, and starts processing the latter. As in Figure 2b, A does not yet cancel the message to object B. B goes ahead with execution of the message sent by A. In Figure 3c, object A finishes processing the correct event. The message sent as a result proves to be exactly the same as the message produced in Figure 3a. Object A takes no action to notify object B of A's rollback. B continues processing the message A already sent. Object B may even manage to complete execution of that message before object A has finished processing the correct event.

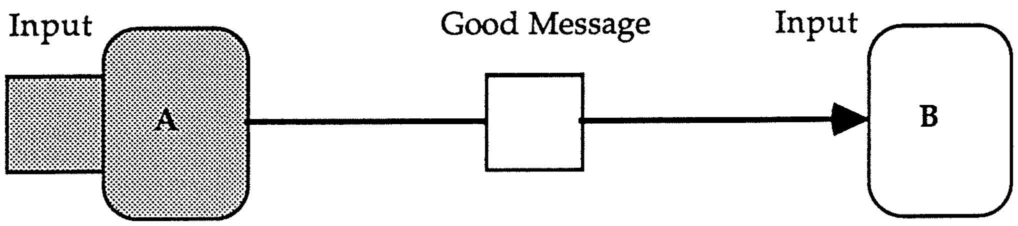


Figure 3a

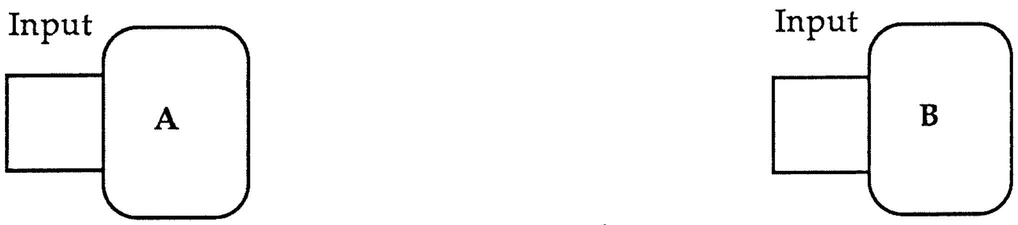


Figure 3b

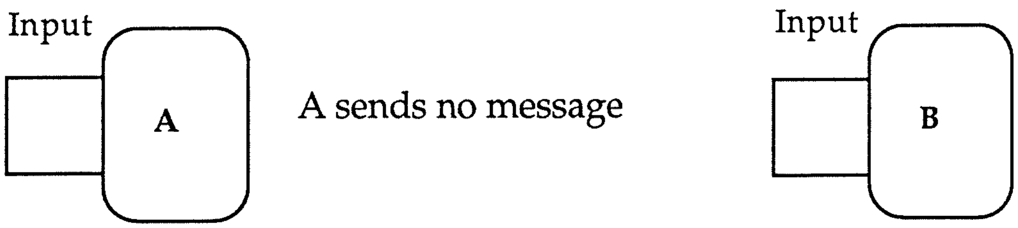


Figure 3c

Figure 3

Comparing Figure 1 to Figure 2, one can see that in the aggressive case, object B is left computing along an incorrect path for a shorter period of time. As soon as object A has reason to suspect that it has sent an incorrect message to object B, the incorrect message is cancelled. If the node hosting B had other, correct work to do, then aggressive cancellation would permit that node to move on to the correct work as soon as possible. Lazy cancellation would require the node to go on doing incorrect work until object A had finished executing its proper work.

On the other hand, comparing Figure 1 to Figure 3 shows that, in the case that object A's earlier message is the same as its later message, lazy cancellation permits the node hosting object B to finish processing the message much sooner than aggressive cancellation would have.

Lazy cancellation mechanisms are more complex than aggressive cancellation mechanisms. Whenever an object rolls back, instead of immediately cancelling the messages produced by the earlier invocation, the object must wait. Whenever the second execution needs to send a message, that message is checked against the existing messages sent by the first execution. If it matches one of the earlier messages, it is not sent again. If it doesn't match, it is sent immediately. After the second execution completes, the object must cancel any messages that were sent during the first execution, but not during the second.

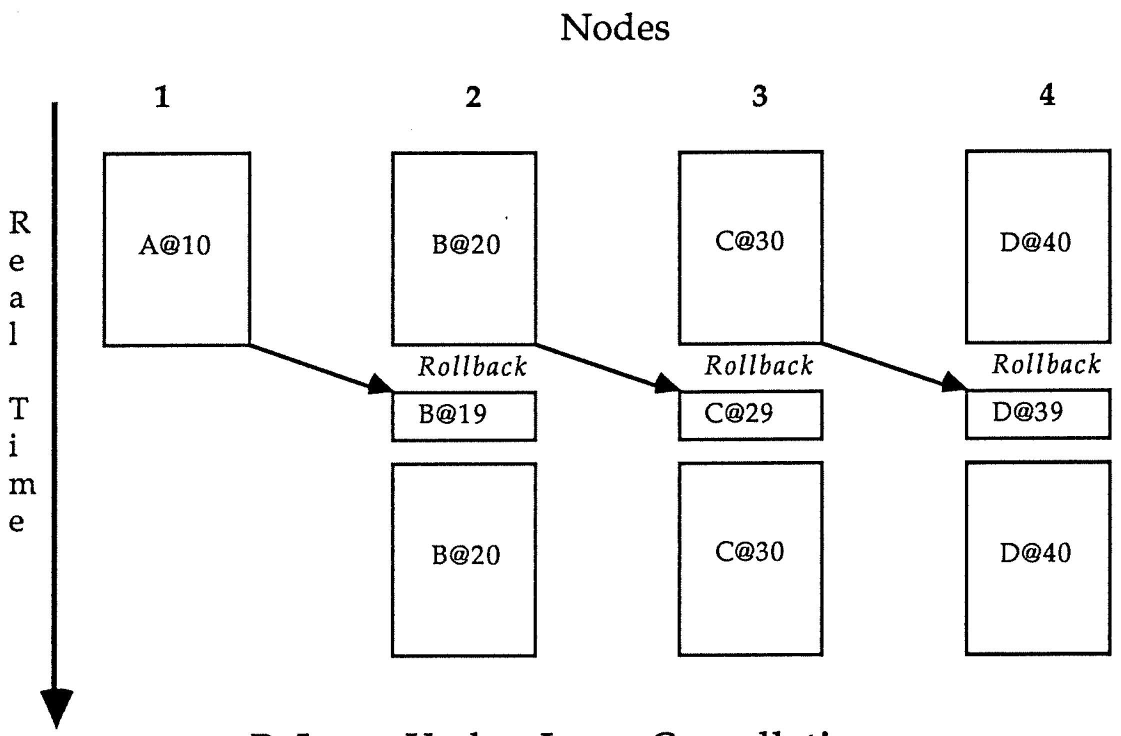
To make this mechanism work, then, the system must be able to compare old messages to new messages. The sending object must be able to easily obtain a copy of a message previously produced, so that it can be compared to a newly produced message. Either an actual copy must be stored, or the sending object must have some other mechanism for looking at the message it sent. Extra logic is also needed to decide whether new messages are to be sent. All of the cancellation logic needed for aggressive cancellation is also needed.

Lazy cancellation pays several performance penalties for its extra optimism. Messages must be compared, sending a message in the middle of the event requires comparison to all old messages, more logic is required for completing an event to ensure that only proper messages are sent, and, depending on how the sending object checks old messages, either additional memory is needed for storing extra copies of messages or the communications network will be burdened by retransmitting copies of messages for comparison purposes. In some cases, lazy cancellation will actually cause more messages to be sent than aggressive cancellation would have, as objects executing on incorrect paths may send out hordes of incorrect messages. Such runaway message senders will be bridled sooner under aggressive cancellation.

4. Extreme Cases for Aggressive and Lazy Cancellation

This section presents two artificial applications that demonstrate extreme cases for the performance of lazy and aggressive cancellation. One application does extremely well under lazy cancellation, but poorly under aggressive cancellation. The other does the opposite. These examples were constructed as "stress cases" for each strategy, and depict situations that may be unlikely in practice. Nevertheless, both examples have been implemented and tested on real systems to validate their behavior.

BeLazy is an application that performs very well under lazy cancellation, but relatively poorly under aggressive cancellation. Figure 4 shows the behavior of BeLazy. In this figure, the vertical dimension shows increasing real time, while the horizontal dimension is space, in terms of nodes of a multiprocessor system. (Only four nodes are shown, but the application generalizes to *n* nodes.) Each node in the system hosts one object. These objects each receive an initial message at the start of the simulation. Object A on node 0 receives a message at simulation time 10, object B on node 1 receives a message at simulation time 20, object C on node 2 receives a message at simulation time 30, and object D on node 3 receives a message at simulation time 40. Due to real time timing considerations, all four of these messages are delivered before any of the objects run any of the messages.



BeLazy Under Lazy Cancellation

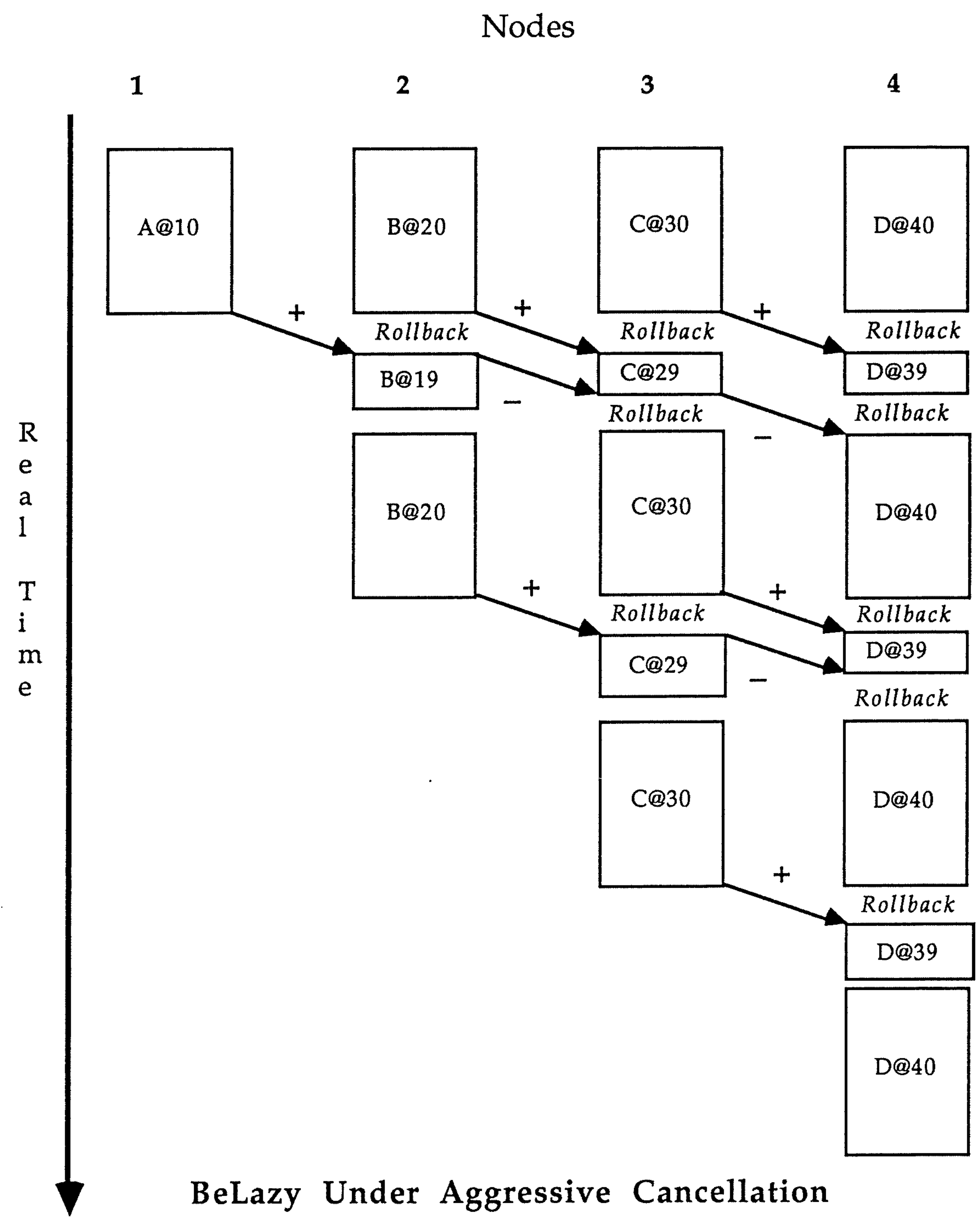
Figure 4

In a correct sequential run of BeLazy, object A would execute at simulation time 10, sending a message to object B at simulation time 19. Object B would execute at simulation time 19, sending no messages, then execute again at simulation time 20, sending a message to object C. That message would arrive at simulation time 29, causing object C to run an event at that time. Then object C would run again at simulation time 30. This event would send a message to object D at simulation time 39, causing object D to run at that time. The run would complete with object D running again at simulation time 40.

Figure 4 shows the parallel run of BeLazy under lazy cancellation. All of the objects start working on their messages. Each event takes the same amount of time. As a result of each event, an object sends a message to the next object in line, A to B, B to C, C to D, with times 19, 29, and 39, respectively. When these messages arrive, objects B, C, and D all roll back. These three objects then execute the messages at time 19, 29, and 39. Each of these events takes the same amount of time. Then objects B, C, and D execute their events at times 20, 30, and 40. Due to the characteristics of BeLazy, the message sent by object B to object C is the same as the message sent in the earlier, out-of-order run of the event at time 20. Similarly, the message sent from C to D is the same. Due to lazy cancellation, these messages are not resent. Since no more messages are sent, the run completes.

Figure 5 shows what would happen if this application were run with aggressive cancellation. As before, objects A, B, C, and D all start running their events. They all send their messages to the next object in line. When object B receives object A's message, though, object B immediately cancels its message to object C, causing object C to roll back and cancel its message to object D. (Cancellation messages are shown with a negative sign under them.) Object's C and D were in the middle of processing their events for times 29 and 39, but were interrupted when their messages for those times were cancelled. They had to roll back whatever they had done and re-execute at times 30 and 40.

B, C, and D run again, B at 19 and 20, C at 30, and D at 40. When object C completes, it again sends its message to D at time 39. When B completes, it sends a message to object C at simulation time 29. Object C rolls back and cancels its message to object D. Object C now runs at simulation time 29 while object D runs again at simulation time 40. Eventually, object C sends its message to object D. Object D rolls back again, executes at simulation time 39, and finally executes its event for simulation time 40 a fourth time. The simulation then ends.

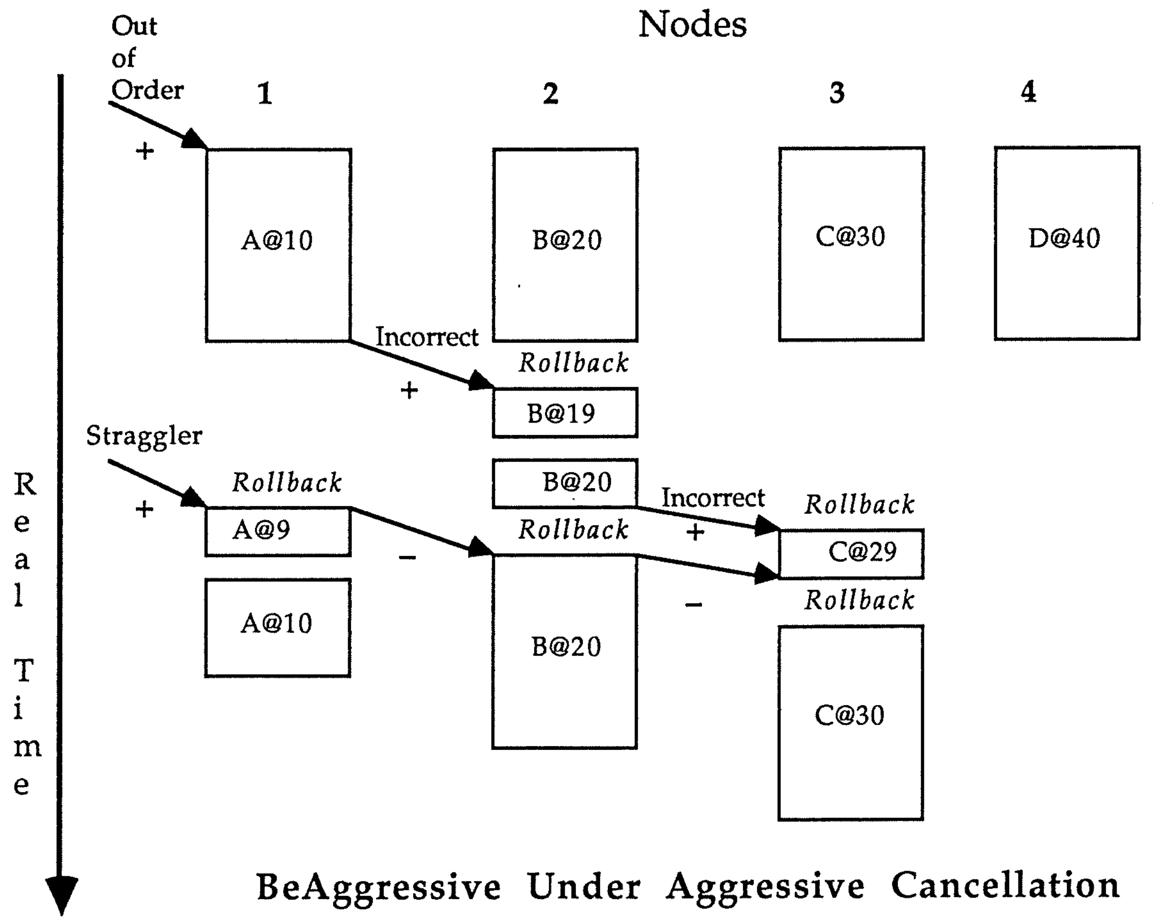


BeLazy Under Aggressive Cancellation

Figure 5

BeLazy will run very quickly under lazy cancellation. If the events at simulation times 10, 20, 30, and 40 take *t* seconds, and the events at simulation times 19, 29, and 39 take negligible amounts of time, BeLazy will require 2*t* seconds (plus overhead) under lazy cancellation. It will take 4*t* seconds under aggressive cancellation. Moreover, if the number of nodes and objects were increased to *n*, BeLazy would always take 2*t* seconds under lazy cancellation, regardless of number of nodes, while it would take *nt* seconds under aggressive cancellation.

BeAggressive is an application that performs well under aggressive cancellation, and poorly under lazy cancellation. Figure 6 shows how BeAggressive works under aggressive cancellation. Like BeLazy, BeAggressive is shown running on 4 nodes, though it can also be generalized to more nodes. Again, the vertical dimension is real execution time, while the horizontal dimension shows nodes used, with one object per node.



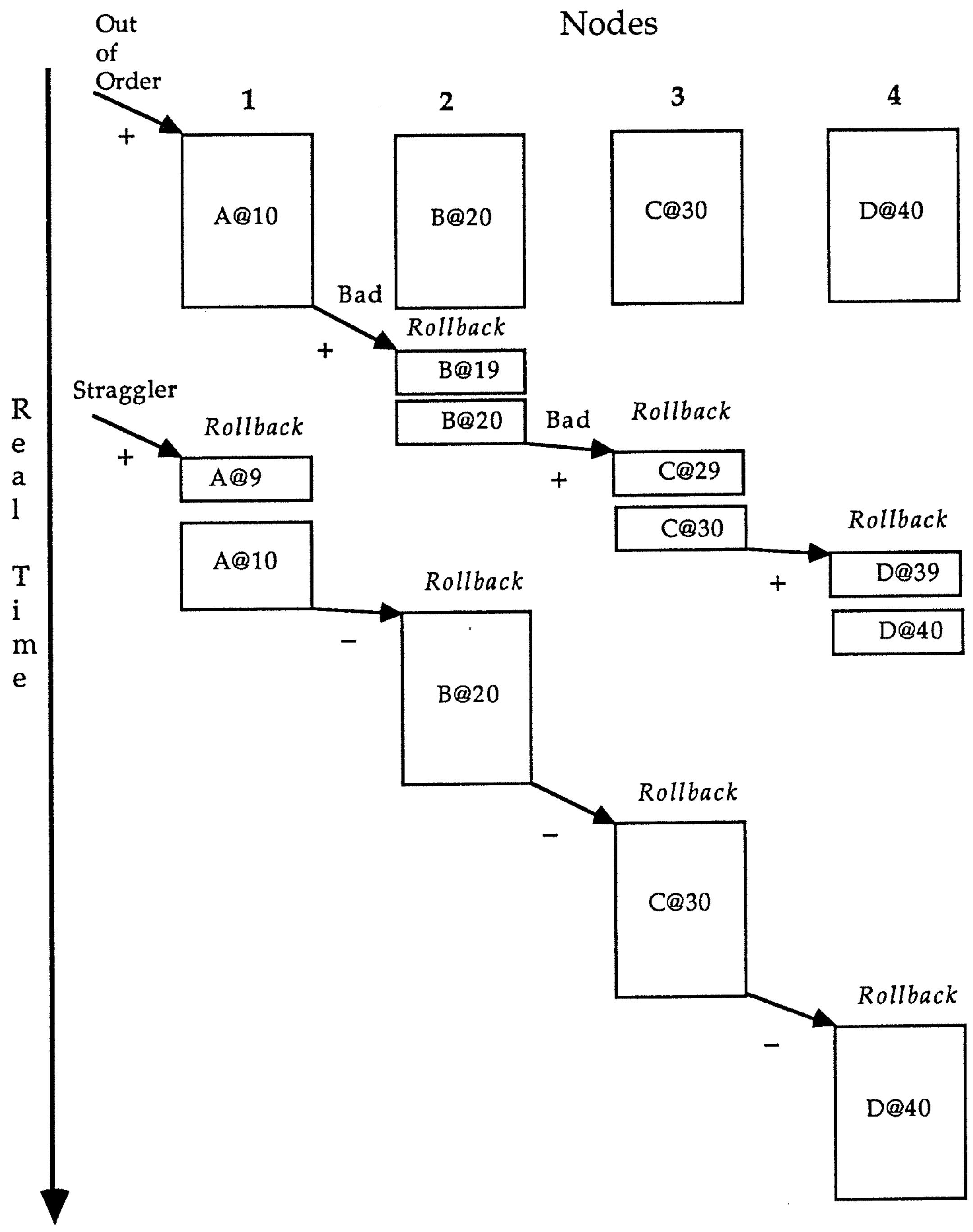
BeAggressive Under Aggressive Cancellation

Figure 6

Each object receives an initial message, A at simulation time 10, B at simulation time 20, C at simulation time 30, and D at simulation time 40. But the message to object A, while a correct message, is out of order. A should first handle a message at simulation time 9. That message's delivery has been delayed, for some reason. When any of the events at simulation times 10, 20, 30, or 40 run correctly, they produce no messages. When A's event at simulation time 10 runs without first handling its message at simulation time 9, A sends an erroneous message to object B for simulation time 19. This erroneous message causes B's event at 20 to run incorrectly and produce a message for object C for time 29. C, in turn, erroneously produces a message to D for time 39. Note that the erroneous runs of the objects take much less time than the correct runs. Once the straggler message to A finally arrives, all of the erroneous messages will be rolled back and the correct version of the simulation can be run.

As Figure 6 shows, aggressive cancellation corrects this situation fairly rapidly. A string of erroneous messages arising from the out-of-order processing at simulation time 10 begins to propagate. But as soon as the straggler message arrives at A, A immediately cancels all future output messages, including the one it sent to B at simulation time 19. As soon as B gets that cancellation, B cancels its message to C. Once all the cancellations have reached the runaway objects, the simulation runs the correct events and completes.

Figure 7 shows how BeAggressive behaves under lazy cancellation. As in Figure 6, A gets a message out of order, spawning an incorrect chain of computations. However, when the straggler arrives at object A, the chain is not immediately chased and destroyed. Instead, the straggler is executed, then A's message for simulation time 10 is re-executed. Only when node 1 is certain that A will not reproduce its erroneous message to B does that message get cancelled. Similarly, once B receives that cancellation, B must re-execute at simulation time 20. Only when B completes that event can it cancel its message to object C at simulation time 30. C also runs a long event before determining that it must cancel its message to D. Once D has re-executed at simulation time 40, the simulation can finally complete.



BeAggressive Under Lazy Cancellation

Figure 7

For BeAggressive, aggressive cancellation executes corrected computations in parallel, while lazy cancellation executes them serially. Note that, in Figure 7, the time between the completion of the incorrect events at an object and the start of the correct event at that object diverges as the distance from the original fault increases. Generally, this divergence would continue for as long as the chain of messages to objects continued, or until the chain returned to a node that had already been traversed in the chain. (In the latter case, optimistic scheduling would tend to bias the system towards the correct computation catching up to the incorrect computation.) In a theoretical system with an infinite number of objects spread across infinite numbers of nodes, the incorrect computation would never be caught by the correction. The correct computation is finite in duration, but it might never complete under lazy cancellation.

In more practical finite systems, the effect is not complete divergence, but can still be arbitrarily bad performance. Assume that the straggler message's event takes negligible time, that all other correct events take t seconds, that objects send messages strictly into the future (messages to be received at the current virtual time are disallowed), and that the computation is running on n nodes. Counting from the time at which the straggler message arrived at object A, the computation takes t seconds plus the time to propagate the cancellation messages. Under lazy cancellation, with the same assumptions, the computation takes nt seconds.

At first glance, the bad cases for both of these simulations seem equally bad. Examining the critical path lengths and sequential execution times of the two simulations reveals an important difference. The critical path length of BeLazy for n objects on n nodes is nt. The sequential execution time is also nt, since all events are on the critical path. Using aggressive cancellation, it takes on the order of nt seconds, while using lazy cancellation takes on the order of t seconds. Aggressive cancellation runs at critical path speed, while lazy cancellation beats the critical path and sequential execution time by a factor equal to the number of nodes used. BeAggressive has a critical path length of t, and a sequential execution time of nt. Aggressive cancellation performs at the critical path for this application, but lazy cancellation takes sequential time.

These examples demonstrate that lazy cancellation can sometimes beat the sequential execution time by a factor of the number of nodes available, in the zero overhead case. At worst, with zero overhead and messages sent strictly into the future, lazy cancellation can take time equal to the sequential time. Aggressive cancellation can never beat the critical path. At worst, with zero overhead and sufficient parallelism, it will never perform worse than the critical path time.

5. Relative Performance of Aggressive and Lazy Cancellation

This section presents performance results for aggressive and lazy cancellation. Results were obtained from two different Time Warp systems. One system is TWOS, the Time Warp Operating System, developed at JPL [Jeff87]. The other is PST, the Parallel Simulator Testbed, developed at the University of Utah [Fuji89]. TWOS was originally developed to use lazy cancellation. PST was first built with aggressive cancellation. Both systems were tested using both aggressive and lazy cancellation, on a variety of benchmarks. All test results shown here were obtained on a Butterfly Plus, running the Chrysalis operating system underneath the Time Warp system.

TWOS is designed to be a portable Time Warp system usable on a variety of parallel and distributed systems, and contains many special features, such as object migration, dynamic memory allocation, and dynamic creation of objects. PST is meant to be a very fast implementation of the Time Warp mechanism for use on shared memory machines. PST tends to run applications as fast or faster than TWOS, but is not as portable and does not contain many of TWOS's features.

Several applications were tested. Implementations of BeLazy and BeAggressive were run on TWOS. Also, two typical applications were run on TWOS to test the relative performance of lazy and aggressive cancellation on more realistic benchmarks. The first is STB88, a military theater level simulation [Wiel89]. The second is Warpnet, a simulation of a computer message passing network [Pres89]. Also, two queueing network simulations were run on PST. The first simulates messages moving among elements in a queueing network, being handled with a first-come, first-serve protocol. The second simulation shows the same situation, but with a small proportion of high priority messages that can preempt handling of lower priority messages.

Figure 8 shows the performance of BeLazy for both lazy and aggressive cancellation under TWOS. Both versions of the system were used, for numbers of nodes varying between 4 and 20. One object was put on each node for each run, leading to different numbers of objects and amounts of work for each number of nodes. As Figure 8 shows, the lazy cancellation version of the system takes approximately the same time, no matter how many nodes were applied. The aggressive cancellation version of TWOS shows a linear increase in its run time as the number of nodes is increased.

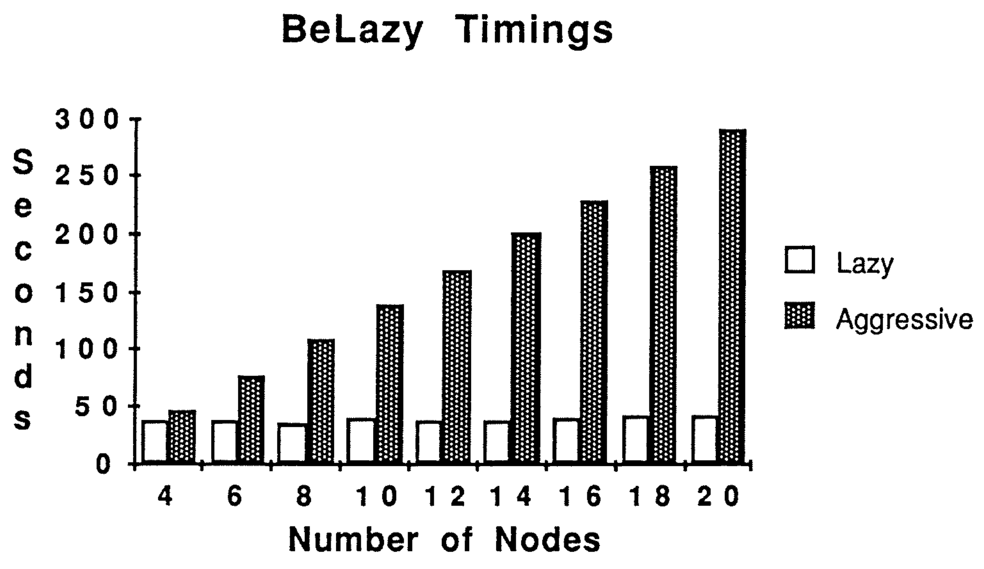


Figure 8

Figure 9 shows the performance of BeAggressive for lazy and aggressive cancellation under TWOS. Again, multiple numbers of nodes were tested and more objects were added along with the nodes. For this application, the aggressive version of TWOS produced an almost constant run time, while the run times of the lazy version increased linearly with the number of nodes.

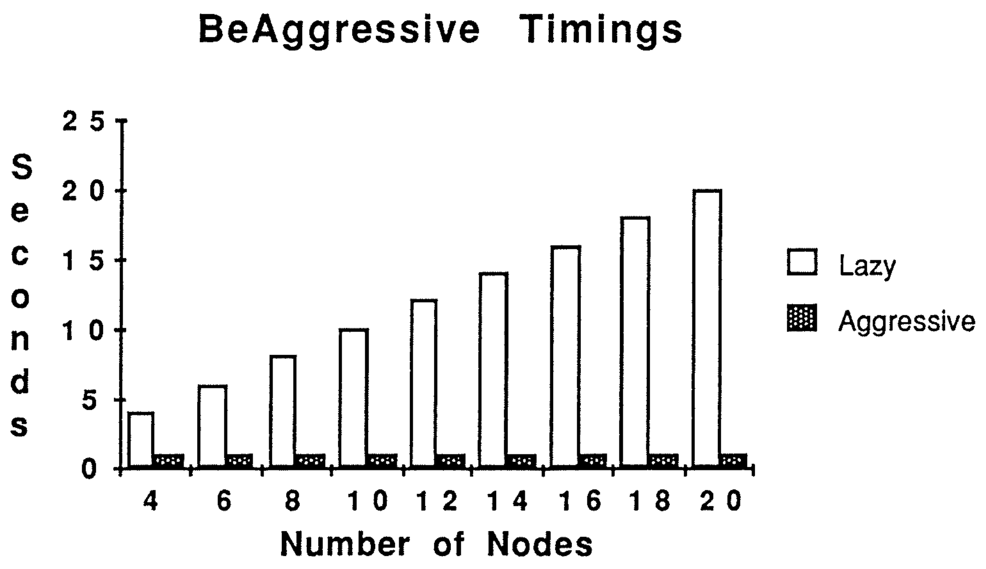


Figure 9

Neither BeLazy nor BeAggressive are claimed to be realistic simulations. Both were written to exercise extreme behavior in cancellation strategies. While they give an idea of how well or how poorly the two strategies could perform in extreme cases, they do not show how the strategies perform in typical simulations. The data on some practical applications gives a better idea of the real performance of lazy and aggressive cancellation. The TWOS data was gathered with a variant version of TWOS 2.0, running on top of the Chrysalis Operating System on a Butterfly Plus. The PST datawas gathered with an experimental version of PST running on top of BBN's implementation of the Mach Operating System, also on a Butterfly Plus.

The performance of STB88 and Warpnet did not change much based on the cancellation mechanism. Lazy cancellation performed slightly better than aggressive cancellation, but no more than 10% better, and usually 1-2% better. In one case, for 50 nodes, aggressive cancellation performed slightly better than lazy cancellation. Figure 10 shows the speed of STB88 with lazy and aggressive cancellation on varying numbers of nodes.

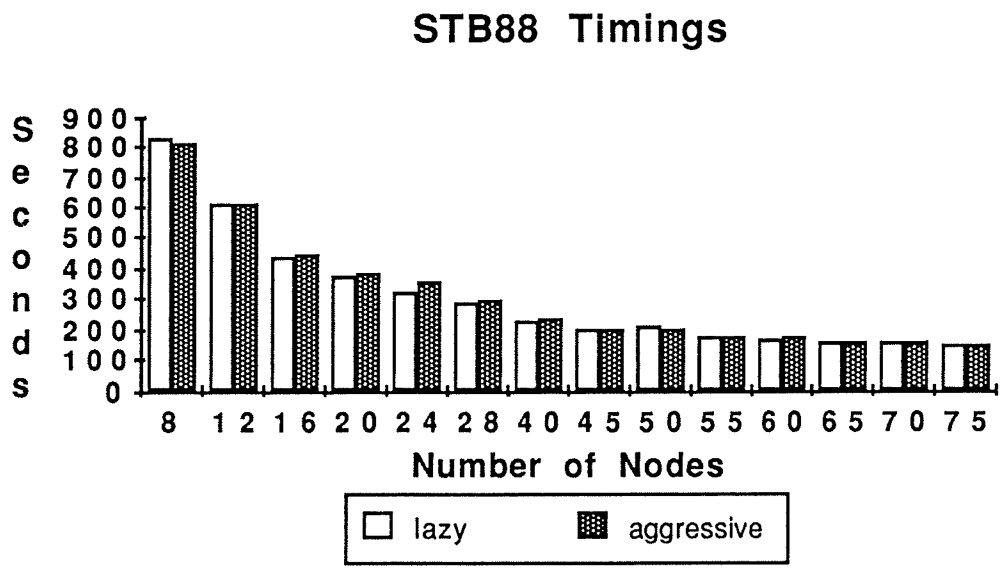


Figure 10

Figure 11 shows the speed of Warpnet. Aggressive cancellation tended to do slightly worse on Warpnet than lazy cancellation. Again, the differences were 10%, at most. The greatest percentage differences tended to show up on fairly short runs, so the size of the percentage is a bit suspect for these cases.

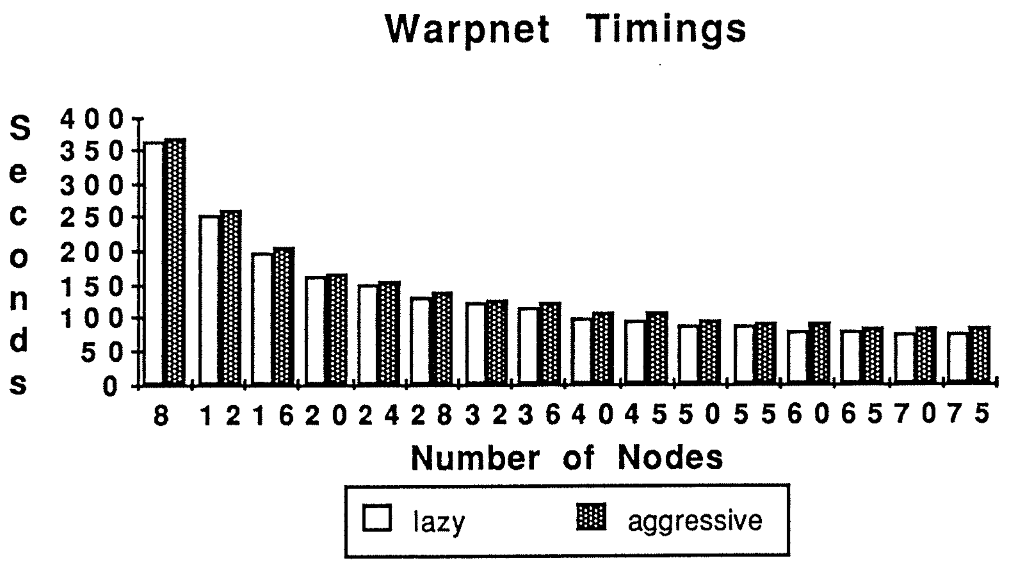


Figure 11

Figure 12 shows the speedup obtained by the PST queueing simulation for FIFO queues, with 256 messages being passed through the network. Neither cancellation strategy has a clear advantage in this case, though aggressive cancellation is a little bit better in most cases.

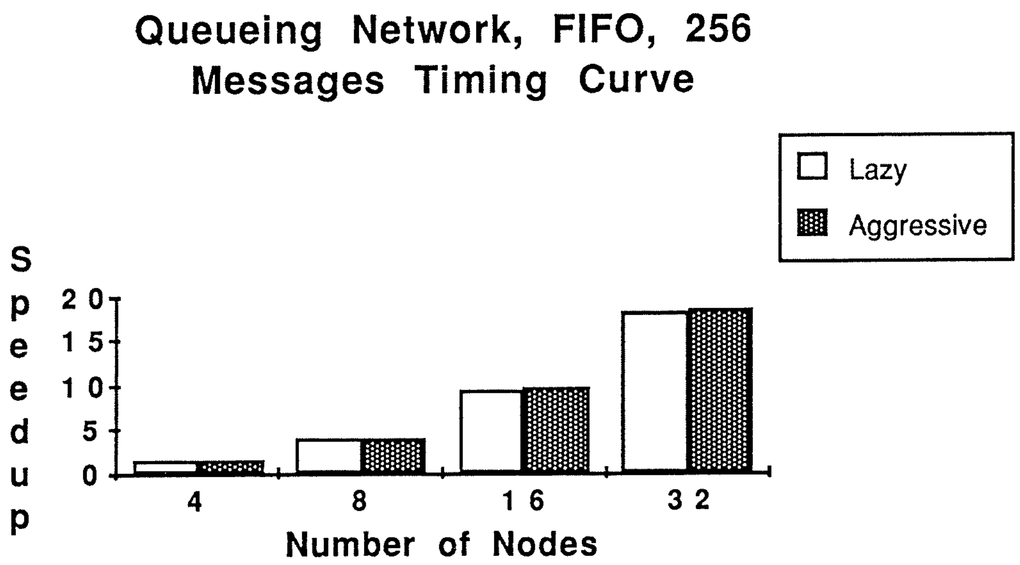


Figure 12

Figure 13 shows the same simulation with 4096 messages in the system. Again, neither strategy shows a significant advantage, though aggressive cancellation has a slight edge.

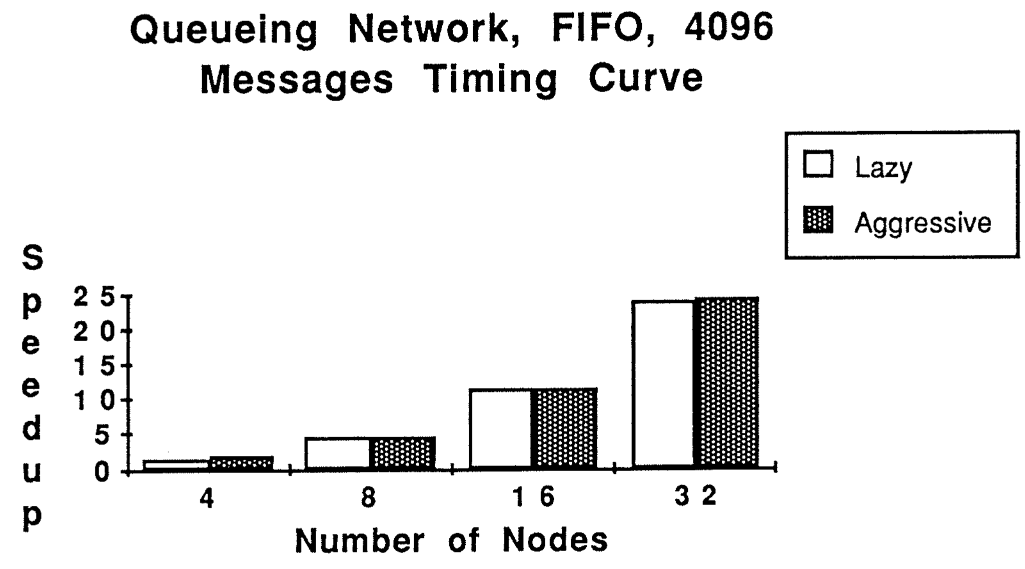


Figure 13

Figure 14 shows the preemptive case for 256 messages in the system. In this case, lazy cancellation does better than aggressive cancellation on higher numbers of nodes, by almost 10%. The reason is that any low priority messages handled out of order will eventually be handled in the same way, with the same results. Therefore, lazy cancellation can gain some speedup by not cancelling the work it has done immediately. With few nodes, the situation that allows lazy cancellation to win is less likely to arise, as the chances of out-of-order messages is decreased.

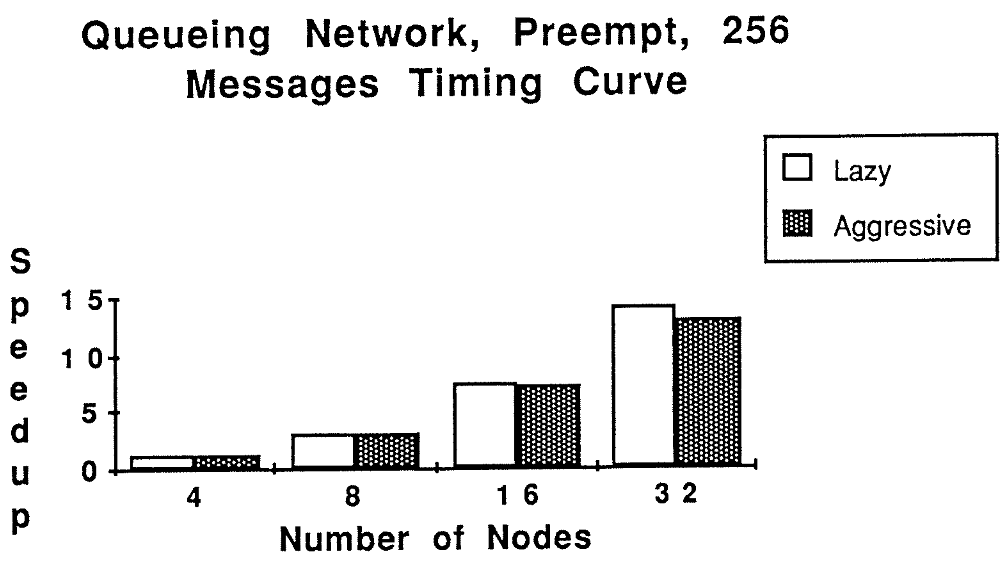


Figure 14

Figure 15 demonstrates the same effect with 4096 messages in the system. Lazy cancellation does nearly as well here as in Figure 14.

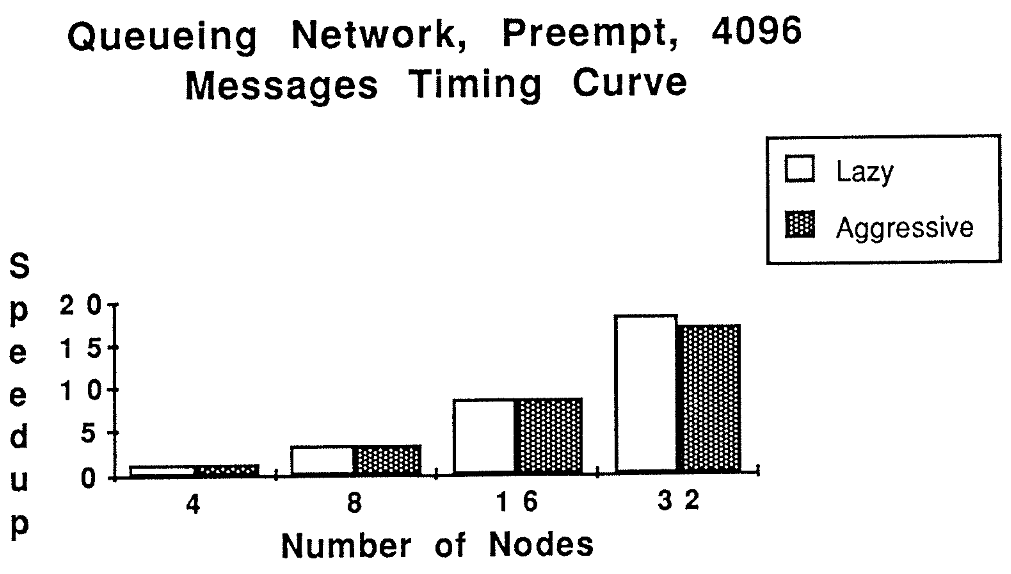


Figure 15

These curves suggest that, for most applications, the difference between using lazy cancellation and aggressive cancellation is slight. In a few cases, lazy cancellation will do better, but not by all that much. At most, a 10% improvement was obtained.

One interesting discovery from making the TWOS measurements is that the implementation of aggressive cancellation is much more sensitive to low-level, internal tuning parameters than lazy cancellation, especially on large numbers of nodes. The lazy cancellation runs would give the same results with a wide range of reasonable values for these parameters. The run times using aggressive cancellation were very sensitive to the values of certain low-level message handling parameters.

6. Conclusions

Depending on the particular application, either lazy or aggressive cancellation can produce better performance. Aggressive cancellation will perform at some level between the critical path performance and sequential performance, for the zero overhead case. Lazy cancellation will perform at a level between the critical path performance divided by the number of nodes used and the sequential performance.

In the practical applications studied, lazy and aggressive cancellation performed within about 10% of each other, and usually much closer. Typically, lazy cancellation performed a bit better than aggressive cancellation. These numbers demonstrate that, for some realistic simulations, lazy and aggressive cancellation both give acceptable performance. There may well exist realistic simulations that cause one or the other strategy to perform much worse than its counterpart.

Giving users the option of using lazy or aggressive cancellation is probably a good choice for designers of optimistic execution systems. However, designers should not expect users to be able to analyze whether their simulation, or a part of their simulation, will perform better with one strategy or the other. Experience shows that even producing the extreme cases was quite difficult, and rarely followed intuition. Expecting users to understand difficult performance issues is unrealistic. Rather, the optimistic execution system should handle the choice between lazy and aggressive cancellation in a manner similar to optimization switches on a compiler. If the performance of an application is not good, changing the cancellation switch setting might make it better, or it might not.

Given that typical TWOS applications show slightly better performance with lazy cancellation than with aggressive cancellation, lazy cancellation will continue to be used as the default cancellation mechanism in that system. TWOS will contain a switch allowing the user to try aggressive cancellation, if he is unsatisfied with the performance of TWOS with lazy cancellation.

Bibliography

[Berr86] Berry, O. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism* , Ph.D. dissertation, University of Southern California, 1986.

[Fuji89] Fujimoto, R. 1989.

[Gafn88] Gafni, A. "Rollback Mechanisms for Optimistic Distributed Simulation", *Proceedings of Society for Computer Simulation Multiconference on Distributed Simulation,* 1988.

[Hont89] Hontalas, P., Beckman, B., Di Loreto, M., Blume, L., Reiher, P., Sturdevant, K., Warren, L. V., Wedel, J., Wieland, F., Jefferson, D. "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part 1: Asynchronous Behavior and Sectoring)", *Proceedings of the Society for Computer Simulation's 1989 Eastern Multiconference,* 1989.

[Jeff87] Jefferson, D., Beckman, B., Wieland, F., Blume, L., Di Loreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., Bellenot, S. "Distributed Simulation and the Time Warp Operating System", *ACM Operating System Review,* Vol. 20, No. 4.

[Lomo88] Lomow, G., Cleary, J., Unger, B., West, D. "A Performance Study of Time Warp", *Proceedings of Society for Computer Simulation Multiconference on Distributed Simulation,* 1988.

[Wiel89] Wieland, F., Hawley, L., Feinberg, A., Di Loreto, M., Blume, L., Ruffles, J., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., Jefferson, D. "Distributed Combat Simulation and Time Warp: The Model and its Performance", *Proceedings of the Society for Computer Simulation's 1989 Eastern Multiconference.*