Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

TRANSPARENT OPTIMIZATIONS OF OVERHEADS IN OPTIMISTIC SIMULATIONS *

Rajive L. Bagrodia
Wen-Toh Liao

Computer Science Department
University of California at Los Angeles
Los Angeles, California 90024

ABSTRACT

A large fraction of the overhead in a typical optimistic simulation is due to the state-saving and recomputation needed to correct a potentially incorrect computation. Traditional mechanisms that are used to trigger recomputations may incur significantly greater overheads than are necessary to ensure a correct computation. In this paper, we describe a performance study of an optimization mechanism which uses application semantics to identify *artificial rollbacks* in an optimistic simulation. Detection of artificial rollbacks can reduce unnecessary recomputations and indirectly reduce state-saving overheads. The detection mechanism has been transparently implemented in a distributed simulation language called Maisie. The paper demonstrates that detection of artificial rollbacks can yield significant performance improvements in the simulation of stochastic systems.

1 INTRODUCTION

The potential for significant performance improvements from parallel executions of simulation programs has led to the design of many algorithms for executing simulations on parallel architectures (Jefferson 1985, Misra 1986, Chandy and Sherman 1989.) However, empirical studies have shown that both conservative and optimistic algorithms incur significant overhead that may reduce the effectiveness of the algorithms in the simulations of some physical systems.

A number of experiments have been designed to analyze the cost components of distributed simulation algorithms in order to identify the primary sources of overhead. For example, Fujimoto (1990), and Lin and Lazowska (1989) note the major impact of the frequency of state-saving overheads on the cost of optimistic simulations, and Gafni (1988) demonstrates how the recomputation overheads may be reduced by using lazy cancellation. With respect to conservative methods, Fujimoto (1990) and Nicol (1988) describe how lookahead in physical systems is useful in obtaining performance improvements. Other studies have shown that application semantics may be used to improve the performance of both conservative and optimistic simulations. For example, Baezner *et al.* (1989) and Wagner (1991) show that, when lookahead is included in the design of the simulation model, the overhead associated with the optimistic or conservative algorithm can be reduced. Sokol, Mutchler, and Weissman (1992) illustrates that the amount of computation associated with each event has a major impact on the performance of parallel simulations. These studies suggest that performance of parallel simulations may be further improved if the simulation engine can deduce semantic information from the applications and perform appropriate optimizations.

Bagrodia and Liao (1990) describes a distributed simulation language called Maisie, and an optimization mechanism to reduce recomputation overhead in optimistic simulations. In this paper, we describe a performance study of Maisie, and demonstrate that the design of Maisie enables the runtime system to use the application semantics and transparently reduce rollback overhead in optimistic simulations. We show that, by identifying *artificial rollbacks* and reducing unnecessary recomputation, the execution of Maisie programs can obtain a significant performance improvement.

The rest of the paper is organized as follows: the next section gives a brief description of Maisie. Section 3 describes the transparent rollback optimization mechanism. The experiments are illustrated in section 4, and the sequential and parallel implementations are described in 5. Section 6 discusses the performance results of the experimental study. Section 7 is the conclusion.

*This research was partially supported by NSF (ASC 9157610) and by ONR (N00014-91-J-1605)

2 MAISIE SIMULATION LANGUAGE

Maisie (Bagrodia and Liao 1990) is a C-based process-oriented simulation language. In Maisie, objects of a given class in the physical system is modeled by an entity-type. An entity-instance, henceforth referred to simply as an entity, represents a specific object in the physical system, and may be created and destroyed dynamically. Interactions among objects are modeled by exchanging of messages. Entities communicate with each other using buffered message passing, and every entity is associated with a unique message-buffer. An entity sends a message to another entity by depositing the message into the message-buffer of the destination entity at the same simulation time as it is sent.

Maisie uses typed messages. Every entity must define the types of messages that may be received by it. An entity accepts messages from its message-buffer by executing a **wait** statement. The wait statement has two components: an optional wait-time (t_c) and a required resume-block. If t_c is omitted, it is set to an arbitrarily large value. The resume-block consists of a set of resume statements, each of which has the following form:

mtype(m_i) [**st** b_i] $statement_i$;

where m_i is a message-type, b_i is an optional boolean expression referred to as a *guard*, and $statement_i$ is any C or Maisie statement. The guard is a side-effect free boolean expression that may refer to local variables or message parameters. If omitted, the guard is assumed to be *true*. The message-type and guard are together referred to as a *resume condition*. A resume condition with message-type m_i and guard b_i is said to be *enabled* if the message-buffer contains a message of type m_i, and b_i evaluates to *true*; the corresponding message is called an *enabling message*.

If two or more resume conditions in a wait statement are enabled, the message with the earliest timestamp is delivered to the entity. If no resume condition in the wait statement is enabled, a special **timeout** message is scheduled for the entity t_c time units in the future. If *prior* to expiration of t_c the entity does not receive any enabling messages, the timeout message is sent to and accepted by the entity (timeout messages are always enabled.) If an enabling message is received before t_c expires, the timeout message is canceled automatically. For convenience, a **hold** statement is provided to unconditionally delay an entity for t_c time units.

As a simple example, Figure 1 describes a priority server in Maisie. In the physical system, the server expects two types of requests, respectively referred to as *high* and *low*, where the requests of the first type have a higher priority and can preempt the service of the request of type *low*. For requests of the same type, they are serviced in a FIFO order. In the Maisie program, an entity-type called *server* is used to modeled the priority server, and it declares two message types, *high* and *low*, to represent the two types of requests that may be received by it. Henceforth, a message of type *high* is referred to as a *high* message; similarly for a *low* message. Initially, a *server* is idle and executes a **wait** statement to accept the next arriving job. If a *high* message is accepted, the server simulates the service of the request by executing a hold statement and t_c is set to be the service time. Upon expiration of t_c, the server accepts the corresponding timeout message, increments the appropriate counter, and starts waiting for another message. On the other hand, if a *low* message is accepted, the server computes the service time *rtime* and executes a selective-wait statement with t_c=*rtime* to simulate the preemptible nature of the service. If a *high* message is accepted before t_c expires, the server computes the remaining service time of the *low* message, services the *high* message, and then resumes servicing the *low* message. If the service of the *low* message is not preempted by a *high* message, the server accepts the timeout message and increments the counter accordingly.

```
entity server{meanh,meanl}
 int meanh,meanl;
{ int hcnt = 0, lcnt = 0, rtime,lostart;
  message high;
  message low;
  while (true)
    wait until
    { mtype(high) {hold(exp(meanh)); hcnt++;}
      or mtype(low)
      {rtime= exp(meanl);
       do { lostart=sclock();
            wait rtime until{
              mtype(high) {
                rtime = rtime - (sclock()-lostart);
                hold(exp(meanh)); hcnt++;}
            or  mtype(timeout) {lcnt++; break;} }
       } while(true);
      }
    }
}
```

Figure 1: Maisie Model of Priority Server

3 ROLLBACK OPTIMIZATIONS

All optimistic algorithms require rollback and re-

computation whenever the runtime system detects the timestamp of a message is less than the current simulation time of the receiver. However, as described in (Bagrodia and Liao 1990), in some cases rollback may be unnecessary, and in some cases the rollback distance may be reduced. In this section, we describe briefly the rollback optimization mechanism and illustrate the idea with examples. We use the term *artificial rollback* to refer to a rollback whose rollback distance can be reduced while maintaining the correctness of the simulation. Detection of artificial rollbacks improves the execution efficiency of optimistic simulations by reducing recomputation and state-saving overheads.

Let r_1 be a subsequence of the correct sequence of messages that is delivered to some entity LP_a. Let F_1 and s_1 respectively be the final state of the entity and the sequence of output messages generated by the entity as a result of receiving the messages in r_1. The state of an entity includes its local variables and its message buffer. Let r_2 be a permutation of r_1, $r_1 \neq r_2$, and F_2 and s_2 be the final state and the sequence of output messages generated due to delivery of r_2 to LP_a. Any one of the following four relationships may hold among F_1, F_2, s_1 and s_2:

(1) $F_1 \neq F_2$ and $s_1 \neq s_2$,
(2) $F_1 \neq F_2$ and $s_1 = s_2$,
(3) $F_1 = F_2$ and $s_1 \neq s_2$, and
(4) $F_1 = F_2$ and $s_1 = s_2$.

In a typical optimistic implementation like TimeWarp (Jefferson *et al.* 1987), delivery of sequence r_2 rather than r_1 would cause recomputation of LP_a and possibly other entities with which communication has occurred, *in each of the four cases.* However, it is clear that only case (1) requires a propagating rollback. The recomputation can be considerably reduced in case (2) and (3), and completely eliminated in case (4). If upon the arrival of an out-of-order message the runtime system can determines artificial rollback as in case (2), (3), and (4), the recomputation overhead may be reduced.

To facilitate transparent detection of artificial rollbacks, the runtime system maintains the following two variables for every entity:

- $mset(t_i)$: set of enabling messages at time t_i.
- $tres(t_i)$: timestamp(s) on the enabling message(s) accepted by the entity when it resumes execution after executing wait statement at t_i.

Assume that an out-of-order message (t_w, m_w) is delivered to an entity when its simulation time is t_n. Let t_l be the latest time preceding t_w at which the entity's state was saved. On receipt of m_w, traditional optimistic simulators immediately rollback the entity to t_l. However, in the optimized implementation, the entity needs to be rolled back only to the earliest t_r, $t_l \leq t_r \leq t_n$, such that m_w belongs to $mset(t_r)$ and t_w *is less than* $tres(t_r)$. In many cases, t_r may be greater than t_l, and in some cases t_r may be equal to t_n, indicating that the rollback is unnecessary. We present an example.

The example uses the simplest form of the wait statement where the resume conditions do not include a guard. Consider the preemptible priority server in Figure 1. Consider the effect of delivering the message sequence (5,*high*), (9,*low*), (7,*high*), (18,*low*), (14,*high*) to the server. Assume that message (5,*high*) is accepted by the server at time 5, and the service time computed is 10. Then, $mset(5)$ for the server includes only **timeout** messages; other messages including (9,*low*) and (7,*high*) that are delivered to the server, will be stored in its message-buffer until it receives a **timeout** message at simulation time 15. So, as long as message (7,*high*) arrives at the server before or at simulation time 15, the order of the arrival of (9,*low*) and (7,*high*) does not affect the correctness of the simulation. Furthermore, if message (14,*high*) is delivered to the entity after simulation time 15, even though the message belongs to $mset(15)$, rollback may be unnecessary as $tres(15)=7$, due to the server initiating the service of message (7,*high*).

Associative Messages It is also possible to detect artificial rollbacks in situations where a message is *processed* rather than simply *delivered* in an incorrect order. We use the notion of *associative* messages to identify such sequences. Consider two sequences r_1 and r_2 defined previously. The two sequences are said to be associative if messages in either sequence may be processed without affecting the correctness of the simulation.

As an example of an associative sequence, consider the following two sequences that are input to a FIFO server: r_1=(5,10,LP_1),(18,7,LP_2),(30,8,LP_1) and r_2=(5,10,LP_1),(30,8,LP_2),(18,7,LP_1) where the message parameters respectively represent the message timestamp, desired service duration and the requesting LP. The two sequences are associative, as the final state of the server and the output message sequences to each customer are the same, regardless of which sequence of input messages is actually processed by the server. Detection of associative sequences allows messages to be processed out of order, thus reducing recomputation overheads for the implementation.

In general, it is difficult for the runtime system to transparently extract relevant information from the

simulation program to identify an associative message sequence. As described in (Bagrodia and Liao 1992) these situations must be identified explicitly by the programmer using specific constructs provided by the simulation language.

4 EXPERIMENTS

The experimental study used three types of queueing networks that have previously been used in performance studies of parallel simulation algorithms. The first example is a closed queueing network (henceforth referred to as CQNF) that consists of N fully connected switches. Each switch is a tandem queue of Q FIFO servers. A job that arrives at a queue is served sequentially by the Q servers and is thereafter routed to one of the N neighboring switches (including itself) with equal probability. The service time of a job at a server is generated from a negative exponential distribution, where all servers are assumed to have an identical mean service time. Each switch is initially assigned J jobs. The simulation terminates when the simulation time exceeds the maximum simulation time (T) specified. Figure 2 displays an instance of the network with N=3, O=2 and Q=4.

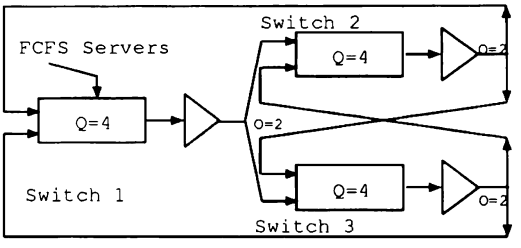

Figure 2: Closed Queueing Network (N=3,O=2)

The second benchmark (henceforth referred to as CQNP) is similar to CQNF except that a job may belong to one of two classes: high or low, where jobs in the first class have a higher priority than those in the second class. Each FIFO server is replaced by a priority preemptible server.

The third benchmark (henceforth referred to as CQNF-A) is similar to CQNF except that the service time of a job is assigned at initialization, whose value is uniformly distributed in the interval [SERVICE_MIN,SERVICE_MAX]. Furthermore, on exiting from a switch, a job is routed back to the current switch with probability P, and is routed to one of its $N-1$ neighbors with probability $(1-P)/(N-1)$. This benchmark is used to demonstrate the utility of the associative message mechanism described in the previous section.

5 IMPLEMENTATION

Sequential implementation: Every event in a Maisie simulation is implemented by the communication of a message, where the timestamp on the message refers to the simulation time at which the event is assumed to occur. Other than timeout messages, all messages generated in the simulation are timestamped with the current value of the simulation clock. As such, these messages can be directly appended to the message buffer of the destination entity, *as soon as they are generated*. This implies that message delivery requires making a single copy of the message parameters together with an append operation; in particular, these messages are never inserted in a global event queue and no search is required. Furthermore, the Maisie runtime system maintains its own free-list, thus avoiding the overhead of making external calls to the operating system routines like *malloc*() and *free*(). With respect to timeout messages, each Maisie entity is associated with exactly one timeout message, which contains the future time at which the message is to be delivered to the corresponding entity. The timeout messages are the only entries that are placed in the event queue. This queue is implemented using a splay tree data structure in order to optimize the *insert* operation.

Parallel implementation: In principle, a Maisie program may be executed using either conservative or optimistic algorithms. This paper studies the performance of Maisie programs when executed using the space-time simulation algorithm. A detailed description of the implementation can be found in (Bagrodia, Chandy, and Liao 1991.)

For the parallel execution, each entity is created on a specific processor and may not move to another processor. Entities mapped to the same processor are executed sequentially. The space-time algorithm is used to synchronize entities on different processors. To implement the space-time algorithm, the runtime system must perform the following major tasks:

- checkpointing and recomputation of entities,
- synchronization of entities, and
- convergence detection to determine the time upto which the simulation has been computed correctly.

Different strategies may be used to implement the preceding tasks. For the results reported in this paper, the runtime system checkpoints an entity after each event, and uses synchronous algorithm for convergence detection.

Even though entities mapped to the same processor are executed sequentially, the runtime system on each processor is different from the sequential runtime system in many ways. Unlike the sequential implementation, the parallel runtime system uses a linked-list for the representation of the global event queue as well as the local message-buffer of each entity. Furthermore, because the sender and receiver entities may have different simulation times, the parallel runtime system cannot use an append operation to deliver a message; rather each message must first be inserted into the global event queue which is sorted on the message timestamps. Messages are subsequently removed from the event queue in increasing order of the timestamps.

Similar to the sequential implementation, the parallel runtime system manages its own free list for message transmission. However, because remote communications use the communication primitives provided by the operating system, additional copy operations must be performed. Overall, each remote communication requires two sets of calls to the system *malloc*(), *memcpy*(), and *free*() routines. This is a significant overhead when compared with a single call to the *memcpy*() function required for a local communication. Furthermore, as the receive operation for remote messages is implemented using polling, rather than interrupts, checking for arrival of remote messages is slightly more expensive than for local messages.

6 RESULTS

Measurements for the experiments reported in this paper were taken on a Symult S2010 hypercube where each node uses a Motorola 68020 cpu and has 4MB of main memory. All programs were coded in Maisie. The Maisie programs used for the parallel implementations were identical to the sequential programs, except for the explicit assignment of Maisie entities to specific nodes of the multicomputer.

6.1 CQNF Experiments

For the Maisie model of the CQNF network, a merge process and its associated tandem queue were modeled by a single entity called *queue* and a fork process was modeled by another entity called *switch*. Thus each node of the physical network is modeled by one *switch* and one *queue* entity; for parallel implementations of the model, the *queue* and *switch* entities corresponding to a node were both assigned to a single processor. The *queue* entity is programmed as follows: for each arriving job, the entity simulates service of the job at each of its Q servers, and subsequently sends a message to the associated *switch* entity. The *switch* entity simply routes each incoming job to one of its N neighbors.

To demonstrate the effectiveness of the rollback optimizations discussed in this paper, three different versions of the Maisie model were executed for each configuration of the CQNF network:

1. CQNF-no-opt: The Maisie model *emulates* the execution of the simulation in a conventional environment that does not support a selective receive primitive. In such simulations, a job message is accepted immediately on its arrival. The corresponding departure time of the job is computed and stored in a local buffer as a part of the state of the entity (Jefferson *et al.* 1987.) The arrival of *any* out-of-order message will trigger a rollback in such implementations.

2. CQNF-opt1: The *queue* entity in the program is modified such that on arrival of a job, the entity computes the departure time for the job and executes a hold statement to simulate the service of the job. The next job in the message-buffer is accepted only after the current job has completed its service. The runtime system does not explicitly detect artificial rollbacks.

3. CQNF-opt2: The Maisie model remains the same as in the previous case. However, the runtime system uses the *tres* and *mset* data structures described in the previous section to detect artificial rollbacks and to perform rollback optimizations.

The first graph (Figure 3) measures the speedup obtained in the parallel simulation as a function of N. The other parameters were fixed as O=N, Q=20, and J=64. As seen from the figure, the speedup increases linearly with N in all three cases. The CQNF-opt2 yields the best result with 70-90% performance improvement over CQNF-opt1 (for $N \geq 2$). While CQNF-no-opt and CQNF-opt1 have similar performance, CQNF-no-opt is about 4-7% better than CQNF-opt1. (We subsequently elaborate on this non-intuitive result.)

Given a network of N switches, the amount of computation is determined by the number of jobs assigned to each switch (J) as also by the number of servers in each tandem queue (Q). Figure 4 shows the speedup of CQNF-opt2 obtained for a network of N=32 switches as a function of the number of jobs in the system for Q=10 and 20 respectively. Note that initially the speedup increases with J and then levels off to reach a peak speedup of about 22 and 25 for

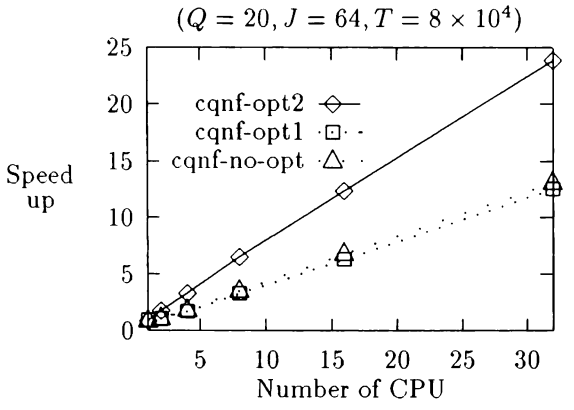

Figure 3: CQNF Speedup: Effect of optimization

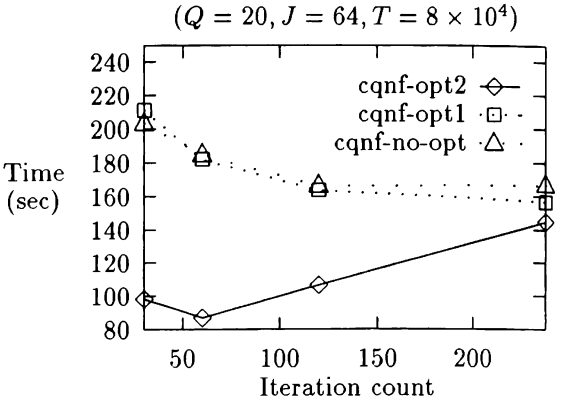

Figure 5: CQNF: Execution time

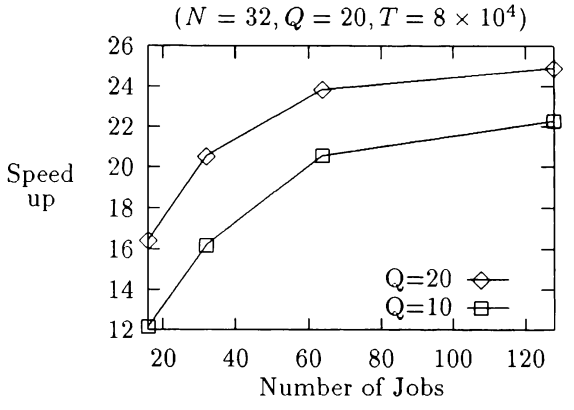

Figure 4: CQNF Speedup: No. of jobs

J=128 depending on the value of Q. This behavior is expected. The initial increase in J offsets the message communication time in the network and once every node is fully utilized, further increases in J have no effect on the speedup. Other things remaining the same, the performance is better for a larger Q, as the network is more fully utilized.

The next set of experiments were designed to study the overhead due to the simulation algorithm and the rollback optimization. For these experiments, the iteration count (K) is defined to be the number of events processed before each node communicates with other nodes. Other things remaining the same, the value of K affects the amount of network message traffic, and the frequency with which the system convergence time (or GVT) is computed, which in turn affects the frequency of garbage collection. Figure 5, 6, 7, and 8 show the total execution time, total number of rollback events, state-saving and garbage-collection overhead, and communication overhead, respectively, as a function of K (for K=30, 60, 120, 240.)

As seen in Figure 5, the execution time of both CQNF-opt1 and CQNF-no-opt decrease monotonically as K increases, but the execution time of CQNF-opt2 increases as K increases after first dropping to the minimum value of 87(sec) at K=64. The intuitive explanation for this behavior is that, as K increases, each node communicates less frequently and thus the node is more likely to be in an incorrect state, resulting in more rollbacks. Figure 6 demonstrates this effect. The total number of rollback events for CQNF-opt2, as expected, increases monotonically as K increases. However, the value of K seems to have little effect on CQNF-opt1 and CQNF-no-opt. The number of rollback events has a direct impact on state-saving and garbage-collection overhead. Figure 7 shows that only CQNF-opt2 demonstrates increases of state-saving overhead as K increases. Note, CQNF-no-opt has much higher overhead due to the fact that messages have to be saved internally in the local state of the entity increasing the size of the state that must be saved.

Figure 8 shows that as K increases, the communication overhead decreases in all three cases. As explain previously, when K increases, there is less communication and thus less overhead. The decrease of the communication overhead in CQNF-opt1 and CQNF-no-opt is also the major contributor to the decrease of the total execution time in both cases.

Note, despite having more rollback events and more state-saving overhead, CQNF-no-opt still has simi-

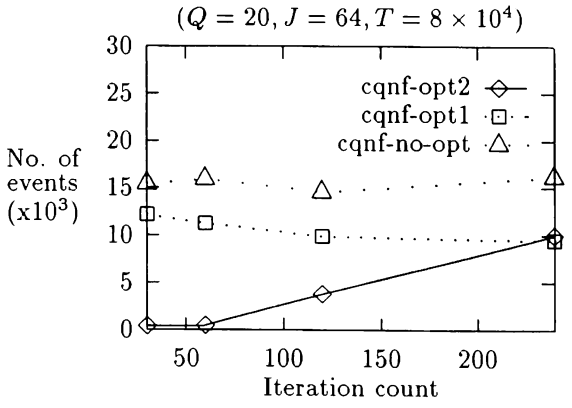

Figure 6: CQNF: Rollback events

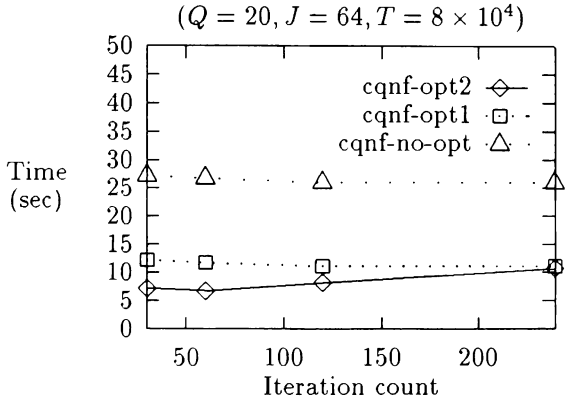

Figure 7: CQNF: State-saving, garbage-collection

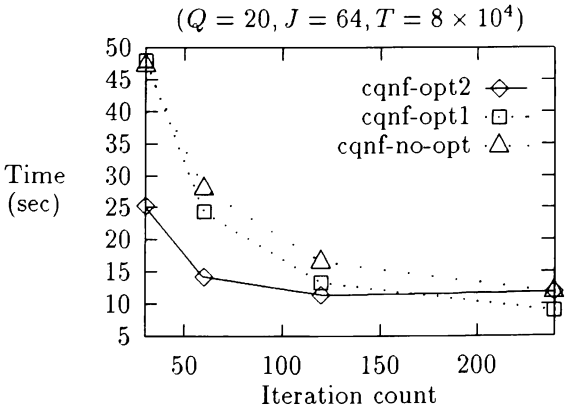

Figure 8: CQNF: Communication overhead

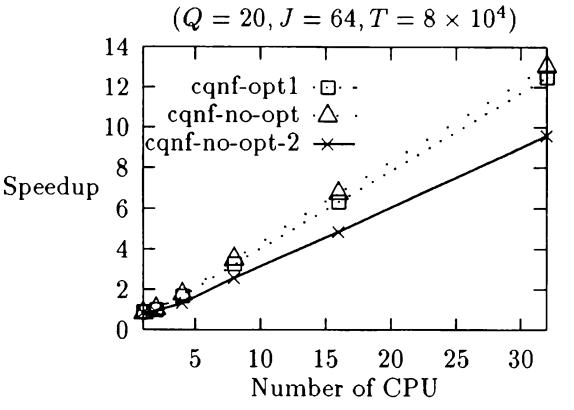

Figure 9: CQNF Speedup

lar execution time as CQNF-opt1, which is counter-intuitive. The explanation is as follows: for CQNF-no-opt model, there are two types of events: (a) accepting a message and computing the departure time; (b) sending a departing job to the associated *switch* entity. Between the two, event granularity (the amount of computation assiciated with an event) of type (a) is larger than that of type (b). Upon rollback, if most of the rollback events are of type (b), CQNF-no-opt requires relatively little cpu time to complete recomputation. Whereas, in CQNF-opt1 model, almost all events are of the first type with a larger event granularity. Recomputation in CQNF-opt1 thus require more cpu time. To confirm this, we re-programmed the *queue* entity to compute the departure time of a job only when it is ready to service the job. Figure 9 shows the result. As expected, the new benchmark, CQNF-no-opt-2, performs much worse than CQNF-no-opt. Examining the data shows that, comparing to CQNF-no-opt, CQNF-no-opt-2 requires more calls to the function computing job departure time. This confirms that recomputation of CQNF-no-opt-2 takes more cpu time to finish than that of CQNF-no-opt.

6.2 CQNP Experiments

Unlike in CQNF, for CQNP experiments, the tandem queue is modeled by Q *queue* entities each of which simulates a priority preemptible server. Each *queue* entity is programmed in a way that when servicing a *high* job, it executes a hold statement; when servicing a *low* job, a selective-receive construct is used to simulate the effect of preemption (see Figure 1.) As in CQNF experiments, three different experiments were conducted for a given configuration:

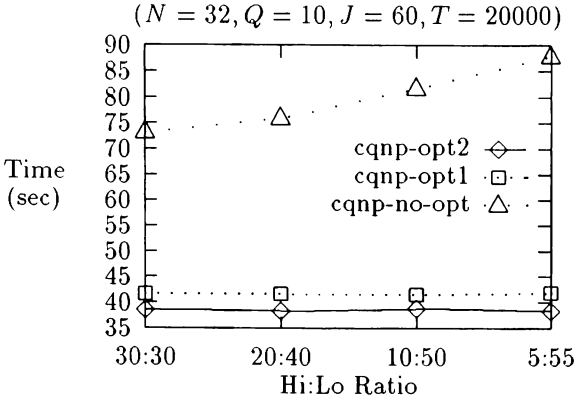

Figure 10: CQNP: Parallel time

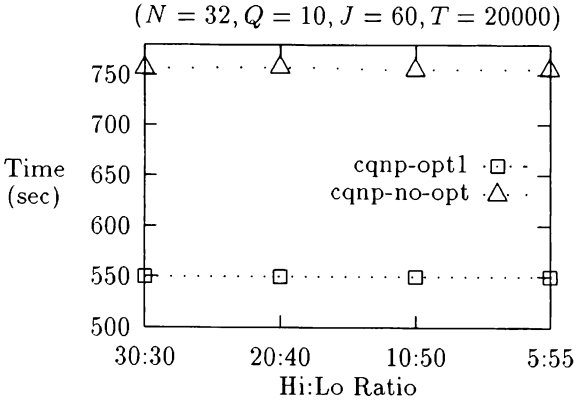

Figure 11: CQNP: Sequential time

CQNP-no-opt, CQNP-opt1, and CQNP-opt2 represent the non-optimized, the minimum optimized, and the fully optimized version, respectively.

The first graph (Figure 10) plots the parallel execution time of the three experiments as a function of the ratio of high to low priority jobs. The configuration is fixed as N=32, T=20000, and J=60. It demonstrates that, though not as effective as in CQNF experiments, rollback optimization (CQNP-opt2) still achieve about 10% improvement over CQNP-opt1. This is as expected. First, because of the preemption functionality of the server, it is less likely to detect artificial rollback in CQNP. Second, since each *queue* entity represents only a single server (compared to Q servers in CQNF), the event granularity in CQNP is smaller, and thus, the recomputation overhead reduced is less significant. The graph also shows that as the number of high job decreases, the performance of CQNP-no-opt is worse. The performance degradation is mainly due to the increase in the number of rollback events. The next graph (Figure 11) plots the sequential execution time of CQNP-no-opt and CQNP-opt1 (or CQNP-opt2). The sequential CQNP-no-opt has worse performance than CQNP-opt1 due to more context-switches (in fact, more than double), event-insertion, and internal message-saving. This again illustrates that Maisie selective-receive construct can be used to transparently minimize context-switching overhead in both sequential and parallel simulations, and state-saving overhead in parallel execution.

6.3 CQNF-A experiments

For CQNF-A experiments, the network is modeled similarly to CQNP: each FIFO server is modeled by a *queue* entity. Each job message carries, as a part of its parameter, the time required for servicing the job in the physical system. The optimization mechanism is implemented using information explicitly passed from the application program: on the arrival of an out-of-order job, the runtime system checks if the server was continuously idle for the duration corresponding to the requested job service time. If so, the rollback optimization for executing associative sequence can be applied.

Different configurations of the CQNF-A model were executed to measure the effectiveness of the rollback optimization. For a given configuration, let D refer to the total rollback distance without implementing the rollback optimization, and D' refer to the rollback distance in the optimized implementation. The reduction in the rollback distance is expressed as a ratio $R = (D - D')/D$. Figure 12 plots R (expressed as percentage) as a function of the number of jobs in the system. The other parameters were kept constant as N=32, Q=10, P=0.5, T=40000; the job service time was sampled from a random number with uniform-distribution in the interval [20, 40]. As seen from the figure, the optimization is useful in a relatively under utilized system where the idle periods of a server are more likely to overlap with the service time of an out-of-order job. As J increases, the idle period is less frequent, and hence this optimization is less effective.

7 CONCLUSION

This paper described a performance study of the transparent optimization mechanism implemented in

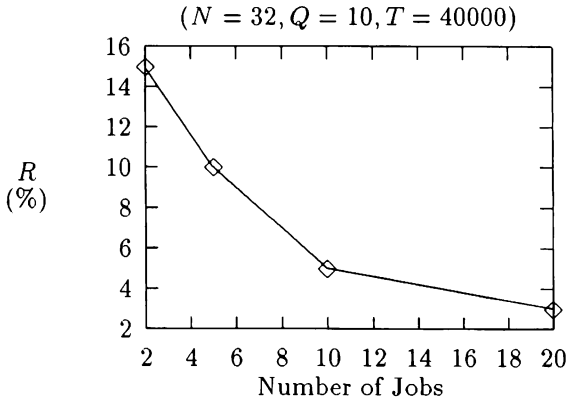

Figure 12: CQNF-A: Rollback distance reduced

the Maisie simulation language. The experiments demonstrated that a number of optimizations can be performed transparently by the runtime system to reduce the rollback and state-saving overheads for optimistic executions of Maisie programs. First, the selective receive primitive of Maisie can be used to ensure that an entity accepts a message only when it is ready to process the message, thus reducing both context-switching and state-saving overheads. Second, by not storing the message internally in an entity, less memory space is required in a sequential execution, and thus less state-saving overhead in a parallel execution. Finally, by identifying *artificial rollbacks*, the parallel runtime system can eliminate unnecessary recomputation and obtain significant performance improvement.

ACKNOWLEDGMENTS

The experiments reported in this paper were executed on a Symult S2010 at Caltech. We are grateful to Professor Chuck Seitz for providing access to the Symult S2010.

REFERENCES

Baezner, D., J. Cleary, G. Lomow, and B. Unger. 1989. Algorithmic optimizations of simulations on Time Warp. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, pages 73–78, Tampa, Florida, January 1989. SCS.

Bagrodia, R.L., K.M. Chandy, and W.T. Liao. 1991. A unifying framework for distributed simulations. *ACM Transactions on Modeling and Computer Simulation*, October 1991.

Bagrodia, R.L., K.M. Chandy, and W.T. Liao. 1992 An experimental study on the performance of the Space-Time simulation algorithm. In *the Proceedings of 6th Workshop on Parallel and Distributed Simulation*, pages 159–168, January 1992.

Bagrodia, R.L., and W.T. Liao. 1990. Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, 205–210, San Diego, California.

Bagrodia, R.L. and W.T. Liao. 1992. A language for iterative design of efficient simulations. Technical report no. UCLA-CSD-920005, Computer Science Department, UCLA, CA 90024, March 1992.

Chandy, K.M. and R. Sherman. 1989. Space-Time and simulation. In *Proceedings of Distributed Simulation Conference*, pages 53–57, March 1989.

Fujimoto, R.. 1990. Parallel discrete event simulation. *CACM*, pages 30–53, October 1990.

Gafni, Anat. 1989. Rollback mechanisms for optimistic distributed simulation systems. In *Proceedings of 1988 SCS Multiconference on Distributed Simulation*, pages 61–67, February 1988.

Jefferson, D. 1985. Virtual Time. *ACM Transaction on Programming Languages and Systems.*, pages 404–425, July 1885.

Jefferson,D., B. Beckman, and F. Wieland et al. 1987. Distributed simulation and the time warp operating system. In *Symposium on Operating Systems Principles*, Austin, Texas, October 1987.

Lin, Yi-Bing, and Edward.D. Lazowska. 1989. The optimal checkpoint interval in Time Warp parallel simulation. Technical Report TR 89-09-04, Department of Computer Science and Engineering, University of Washington, September 1989.

Misra, J. 1986. Distributed discrete-event simulation. *ACM Computing Surveys*, pages 39–65, March 1986.

Nicol, D.M.. 1988. Parallel discrete event simulation of FCFS stochastic queueing networks. *ACM SIGPLAN*, pages 124–137, July 1988.

Sokol, L.M., P.A. Mutchler, and J.B. Weissman. 1992. The role of event granularity in parallel simulation design. In *Proceedings of 6th Workshop on Parallel and Distributed Simulation(PADS92)*, Marc Abrams and Paul Reynolds, editors, volume 24:3, pages 178–185. SCS, January 1992.

Wagner, David. 1991. Algorithmic optimizations of conservative parallel simulations. In *Proceedings of Advanced in Parallel and Distributed Simulation*, V Madisetti, D. Nicol, and R. Fujimoto, editors, volume 23:1, pages 25–32. SCS, January 1991.