Maisie: A Language for the Design of Efficient Discrete-event Simulations [*]

Rajive L. Bagrodia
Wen-Toh Liao

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

Abstract

Maisie is a C-based discrete-event simulation language that was designed to cleanly separate a simulation model from the underlying algorithm (sequential or parallel) used for the execution of the model. With few modifications, a Maisie program may be executed using a sequential simulation algorithm, a parallel conservative algorithm or a parallel optimistic algorithm. The language constructs allow the runtime system to implement optimizations that reduce recomputation and state saving overheads for optimistic simulations and synchronization overheads for conservative implementations. This paper presents the Maisie simulation language, describes a set of optimizations and illustrates the use of the language in the design of efficient parallel simulations.

1 Introduction

Distributed (or parallel) simulation refers to the execution of a simulation program on parallel computers. A number of algorithms[25, 10, 11, 21, 20] have been suggested for distributed simulation and many experimental studies have been conducted to evaluate the speedups that may be obtained from these algorithms and their variants. Experience with parallel simulators suggests that reduction in the completion time of a simulation depends significantly on the application as well as the specific algorithm used to execute the model on a parallel architecture. For some models, multiple independent replications or even sequential implementations may be more suitable than parallel implementations. In the absence of a priori knowledge about the suitability of a specific simulation algorithm for a given application, it is desirable to develop languages that separate the model from the underlying algorithm. Such notations would allow the analyst to develop a model and subsequently select the most suitable algorithm for its execution.

In general, efficient implementation of a model using a particular simulation algorithm requires that the model reflect some aspect of the underlying algorithm. Thus, an efficient conservative implementation may require that interactions among sub-components of the model be regular, whereas an optimistic implementation may require that checkpointing overheads be low. However,

[*]This research was partially supported under NSF PYI award ASC 9157610, ONR grant N00014-91-J-1605, Rockwell International Award No. L911014, and ARPA/CSTO Contract No. J-FBI-93-112

it is possible to develop an initial model that abstracts these differences and instead focuses on modeling the physical system at an appropriate level of detail. Subsequent refinements to the model may be used to improve its efficiency with respect to a specific simulation algorithm.

In this paper we use interrogative simulation constructs[13] to design a simulation language that supports efficient execution of sequential and parallel simulation models. Although we present our ideas in the context of a C-based language called Maisie[6], they can be incorporated in most environments for discrete-event simulation, regardless of their choice of a base language. Maisie is the only existing language that supports the execution of a discrete-event simulation model with multiple algorithms and provides constructs to reduce the simulation overheads with both conservative and optimistic parallel algorithms. The simulation algorithms currently supported by Maisie include a sequential algorithm, parallel conservative algorithms based on null messages[25] and conditional events[10], a new conservative protocol that combines null messages with conditional events[23], and a parallel optimistic algorithm[11, 4].

The initial Maisie model is typically executed using a sequential algorithm. If the completion time of the sequential program is not acceptable, parallel implementations may be explored by the analyst. The first step is to make simple modifications to the program to explicitly distribute the simulation objects among available processors. At this stage, the analyst need not be concerned with the specifics of the synchronization protocol that is used to execute the simulation model on the parallel architecture. Subsequent refinements depend on the specifics of the particular simulation algorithm that is to be used. If an optimistic algorithm is used, these refinements can be targeted to reduce state saving and recomputation overheads for the program. In contrast, if a conservative algorithm is to be used, the optimizations may be used to improve lookahead properties of the model to reduce synchronization overheads. The goal at this stage is to exploit the specifics of the application and the simulation algorithm to generate an efficient implementation. The availability of an equivalent sequential implementation permits consistent comparisons of the relative efficiency of the parallel implementations. Some of these optimizations have also been useful in improving the efficiency of sequential implementations.

In addition to the user-specified optimizations, the Maisie runtime system also implements transparent optimizations of rollback overheads for optimistic simulations. In a subsequent section, we introduce the concept of *semantic rollbacks* and describe how these can be detected transparently by the Maisie runtime system to reduce overheads for optimistic algorithms. We also indicate how *lookahead* characteristics of some applications may be extracted transparently from the program to reduce the synchronization overheads for conservative algorithms.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 describes the primary constructs of the Maisie simulation language. Section 4 indicates how Maisie programs may be executed transparently using three different parallel simulation algorithms. Section 5 is devoted to reducing the overhead of optimistic implementations and section 6 discusses optimizations for conservative implementations. Section 7 addresses implementation issues, particularly the implementation of Maisie constructs that support interrogative simulation. Section 8 is the conclusion.

2 Related Work

A large number of sequential simulation languages have been designed including Simula, GASP, GPSS, Simscript, MAY[5], CSIM[32] and many others. In contrast, design of parallel simulation languages (PSL) is a relatively new area of research. PSLs typically adopt one of two approaches:

(a) enhance sequential simulation languages with primitives or library functions to specify parallel execution; examples include Yaddes[28], Maisie[6], SIMA[29], Modsim[35] and Sim++[3] among others, and (b) add simulation constructs to existing parallel languages as typified by Ada-based simulation environments like SCE[19].

The goal of Maisie was to design a simulation language that could be used to develop efficient sequential and parallel simulations. It is among the few languages that support both conservative and optimistic algorithms for its parallel implementations. The specific simulation constructs provided by Maisie are similar to those provided by other process-oriented simulation languages. For example, Sim++, a C++ based language that supports sequential simulations and Time-Warp simulations, has comparable constructs. However, Maisie extends these constructs to allow the enabling condition for an event to be described succinctly. Furthermore, Maisie is the first language that provides user-transparent and programmer-specified facilities to reduce the runtime overheads for parallel implementations using conservative or optimistic simulation techniques.

Other languages/systems that support multiple parallel simulation algorithms include OLPS [1], Yaddes[28], SPECTRUM[30], and SPEEDES[34]. Yaddes is a specification language for event-driven simulation that resembles Yacc and Lex. A Yaddes program is translated into a C program which is later compiled and linked with the runtime support library. Different simulation environments are provided by specifying the appropriate runtime library at link time. The system supports sequential, conservative, and optimistic simulations. Yaddes requires that the configuration of logical processes in the model and its mapping on a parallel architecture be specified completely at compile time.

OLPS provides a C++ object library to support sequential, conservative and optimistic simulation. It uses YACC grammar to specify the simulated physical system. OLPS is not algorithm-independent because the set of objects specified by the program depends on the specific algorithm used to execute the program. Also, OLPS uses heavyweight UNIX processes that are created at runtime, which may potentially increase its runtime overhead.

SPEEDES is a C++ based simulation environment which supports sequential algorithm, time-driven algorithm, Time-Warp algorithm, and the SPEEDES algorithm (a combination of the time-bucket and the Time-Warp algorithm). Reynolds[30] described nine design variables (partitioning, adaptability, aggressiveness, accuracy, risk, knowledge embedding, knowledge dissemination, knowledge acquisition, and synchrony) for designing a distributed simulation algorithm. SPECTRUM provides a library of application and support routines to evaluate algorithms that result from different combination of these design alternatives. Although the preceding systems are useful for comparative studies of parallel simulations, they do not provide language support for model optimizations.

A number of hardware techniques have been suggested to improve the performance of parallel simulations. For instance, synchronization overheads in both conservative and optimistic implementations may be reduced by using reduction networks[31]. State-saving and rollback overheads for optimistic simulations may be reduced transparently by incorporating a rollback chip[15] in the simulation engine.

3 Simulation Language

Maisie is a C-based derivative of MAY[5] and has been influenced in varying degrees by distributed programming languages like CSP and SR[2] among others. This section is a brief description of the Maisie simulation constructs. The Maisie manual[8] contains a complete description of the language.

Maisie adopts the process interaction approach to discrete-event simulation. An object (also referred to as a PP for physical process) or set of objects in the physical system is represented by a logical process or LP[25, 9]. Interactions among PPs (events) are modeled by message exchanges among the corresponding LPs. We first describe the process representation and communication primitives of Maisie and subsequently indicate how they are used to describe events.

3.1 Entities

A Maisie program is a collection of entity definitions and C functions. An entity definition (or an entity type) describes a class of objects. An entity instance, henceforth referred to simply as an entity, represents a specific object in the physical system. Maisie supports dynamic and recursive entity creation. An entity is created by the execution of a **new** statement which may optionally specify the processor on which the new entity will execute. An entity terminates itself by 'falling off the end' of the entity body.

Figure 1 describes an entity type to model a resource manager. The heading in lines 1 and 2 indicates that the entity type is called *manager* and has one integer parameter called *max_printers*. This entity type will be used as a running example to illustrate various Maisie constructs.

```
1  entity manager{max_printers}
2    int max_printers;
3  { int units = max_printers;
4    message preq{ ename hisid; } ;
5    message releas;
6    for (;;)
7      wait until
8      { mtype(preq) st (units>0)
9          { units--;
10             invoke msg.preq.hisid  with done; }
11     or mtype(releas)
12         units++; }
13 }
```

Figure 1: A Resource Manager: Single Resource

3.2 Messages

Entities communicate with each other using buffered message passing. Every entity has a unique message buffer; asynchronous send and receive primitives are provided to respectively deposit and remove messages from the buffer.

Maisie uses typed messages. An entity type must define the types of messages that may be received by its instances. A message type consists of a name and a (possibly empty) parameter list. For instance, two message types are defined for the *manager* entity type(lines 4–5): *preq* which has one parameter of type **ename**, and *releas* which has no parameters. Variables of type ename are used only to store entity-identifiers. An entity sends a message by executing an **invoke** statement. Each message is transparently timestamped with the current simulation time and is deposited in the destination buffer at the same (simulation) time at which it is sent. For instance, execution of the invoke statement in line 10 of the *manager* will deposit a message of type *done* in the message buffer of the entity identified by *hisid*. Note that message types need to be declared only by the recipient entity.

An entity accepts messages from its buffer by executing a **wait** statement. This statement has the following form:

```
wait $t_c$ until
   {  declarations;
      $r_1$;
   or $r_2$;
      ⋮
   or $r_n$; }
```

where t_c is a numeric value called wait-time and each r_i is a *resume statement*. The most commonly used version of the resume statement references a single message type and has the following form:

```
   [mvar=] mtype($m_t$) [max $v_i$] [st $b_i$]
                                   statement;
```

where m_t is a message type, *mvar* is a variable of type m_t, b_i is a boolean expression called a *guard*, v_i is a message parameter called a *ranker*, and *statement* is any Maisie statement. The guard is a side-effect free expression that may reference entity variables and message parameters. If omitted, it is assumed to be the boolean constant *true*. The guard is said to be *local*, if it references only entity variables. The message type, guard, and ranker are together referred to as a resume condition. A resume condition with message type m_t and guard b_i is said to be *enabled* if the message buffer contains a message of type m_t whose timestamp is at most equal to the simulation clock, and b_i evaluates to *true* (b_i is evaluated only if the buffer contains a message of type m_t); the corresponding message is referred to as an *enabling* message. For instance, the resume statement in line 8 of entity *manager* is enabled only if the buffer contains a message of type *preq* and the local guard ($units > 0$) is satisfied.

On execution of a wait statement, the message buffer is searched for an enabling message. If the buffer contains more than one enabling message of a given type, the ranker is used to select a unique enabling message: if keyword **max** (**min**) is used, the enabling message with the largest (smallest) value for parameter v_i is selected. If the ranker is omitted, the enabling message with the smallest timestamp is selected. If two or more resume conditions are *enabled*, the timestamps on the selected enabling message of each type are compared and the message with the earliest timestamp is selected. The selected message is removed from the buffer and delivered to the entity either in the corresponding variable *mvar* or, if *mvar* is omitted, in a system-defined variable called **msg**.

If no resume condition is enabled, the entity is suspended for a *maximum* duration equal to its wait-time t_c; if omitted, t_c is set to a value that is larger than the maximum simulation time for the model. A suspended entity resumes execution *prior* to expiration of t_c, if it receives an *enabling* message; otherwise the entity is sent a special message called a **timeout** message, as discussed in the next subsection. An entity may implement a non-blocking receive by specifying t_c=0.

Once again, consider the *manager* entity type of Figure 1: the wait statement in lines 7–12 contains two resume statements. The first resume statement (line 8) specifies *preq* as the message type and was discussed earlier. The resume condition in the second statement (line 11) does not include a guard. This condition is enabled whenever a *releas* message is available in the buffer. As neither resume condition specifies a message variable, the enabling message is returned in variable **msg**[1]

A resume condition may also reference message parameters. For instance, assume that the manager in our running example receives requests for one or more printer units and that incoming requests are serviced using the *first-fit* discipline. The following fragment shows how the resume condition is modified to ensure that a *preq* message is accepted only if the requested number of units are available.

```
5       message preq{ ename hisid; int count; } ;
        ⋮
7         wait until
8         { mtype(preq) st (msg.preq.count<=units)
            ⋮
```

Maisie also provides a number of pre-defined functions that may be used by an entity to *inspect* its message buffer. For instance, the function **qsize**(m_t) returns the number of messages of type m_t in the buffer. A special form of this function called **qempty**(m_t) is defined, which returns *true* if the buffer does not contain any messages of type m_t, and returns *false* otherwise. This function may be used to impose priorities on incoming messages. Assume that the manager is to be modified such that a request message is accepted only if no *releas* messages are present in the buffer. A simple way to do this is to strengthen the guard for the *preq* message as follows:

```
7       wait until
8       { mtype(preq) st (qempty(releas) && (msg.preq.count<=units))
```

Appropriate use of guards simplify the entity definition because the code to accept and buffer messages that cannot be processed immediately need not be included in the entity definition. The guard also facilitates rollback optimizations as discussed in section 5. Efficiency issues in the implementation of guards are examined in section 7.

3.3 Events

Each event in a discrete-event simulation model simulates some activity of interest in the physical system and may involve one or more objects. For instance, in the resource manager model, events include ‘a job requesting a printer’ or ‘a job using the printer for t time units’. In a Maisie model, events are simulated by messages. For instance, the first event is modeled by a job entity sending a *preq* message to the *manager* entity; the second event is modeled by a job entity executing a wait statement with wait-time t such that a timeout message is received by the entity after t time units have elapsed.

[1]The type of variable msg is declared by the compiler to be a union of message types. This imposes the syntactic requirement that a specific message type be included explicitly in any reference to a field of this variable, such as **msg**.*preq.hisid* in Figure 1.

An event is either *definite* or *conditional*. Assume that an entity schedules a future (timeout) event for time t_e at simulation time t_s, where $t_s \leq t_e$. The event is said to be *definite* or unconditional if its occurrence is independent of any other event in the system in the interval $[t_s, t_e]$; otherwise it is said to be *conditional*. Both definite and conditional events may be scheduled by executing an appropriate wait statement.

Consider an entity that models a priority preemptible server. The entity expects two types of requests, *low* and *high*, where the arrival of a *high* message can interrupt the processing of a *low* message. Assume that each message needs 10 units of service. Service of a *high* message is simulated by the following wait statement which schedules a definite timeout message:

```
wait 10 until mtype(timeout)
    invoke jobid with done;
```

Execution of the preceding wait statement suspends the entity for 10 time units because the wait-time is 10 and its single resume statement can be enabled only by a timeout message. A wait statement that schedules a definite timeout message may also be abbreviated by a **hold** statement. The following fragment is equivalent to the preceding wait statement:

```
    hold(10);
    invoke jobid with done;
```

A wait statement may also be used to schedule conditional events. The following statement uses a *conditional* time-out message to simulate service of a *low* message; *rtime* refers to the remaining service time of the *low* message that is currently in service. The timeout message is rescheduled if a *high* message is received by the entity in the interim.

```
    wait rtime until
        {   mtype(high)
                preempt, recompute rtime, and
                serve high priority message;
        or  mtype(timeout)
                invoke jobid with done;
            ⋮
```

3.4 Compound Resume Conditions

We now consider resume conditions that include multiple message types. The general form of a resume statement is as follows:

```
mvar_a = mtype(m_a) [max v_a] [st b_a]
and mvar_b = mtype(m_b) [max v_b] [st b_b]
⋮
and mvar_n = mtype(m_n) [max v_n] [st b_n]
                                    statement;
```

The preceding statement is enabled if the message buffer contains a different enabling message for each conjunct in the resume condition. If the statement is enabled, the corresponding set of enabling messages is removed from the buffer and delivered to the entity in the specified message variables.

```
1   entity manager{max_printers}
2   int max_printers;
3   {
4     int i, units = max_printers, cfree[MAXC];
5     message chnls{ename hisid; int cno;} csend;
6     message chnlr{ename hisid; int cno;} crecv;
      ⋮
14      wait until {
      ⋮
20      or  csend= mtype(chnls) st (cfree[msg.chnls.cno])
21        and crecv= mtype(chnlr) st (msg.chnlr.cno==csend.cno)
22          { cfree[csend.cno]=0;
23            invoke csend.hisid with alloc{cno};
24            invoke crecv.hisid with alloc{cno}; }
              ⋮
27
```

Figure 2: Resource Manager: Multiple Resources

The **and** operator in the compound resume condition is a short circuit operator; a conjunct is evaluated only if the message buffer contains an enabling message for each preceding conjunct. In a compound condition of the form $mvar_a$=*r1* **and** $mvar_b$=*r2*, the guard in resume condition *r2* may reference message variable $mvar_a$. However, $mvar_a$ is modified only if the corresponding enabling sequence is delivered to the entity.

We illustrate the use of compound resume statements by modifying the *manager* entity type to include channel resources. A channel is identified by a unique integer id and requests for a channel are satisfied only in pairs that match a transmitter process with a receiver. The transmitter process requests a channel using a *chnls* message and the receiver process uses a *chnlr* message. A *chnls* request matches a *chnlr* request only if both messages contain the same channel id. A specific channel is allocated by the manager if it receives matching requests and the requested channel is available. Similarly a channel becomes available only when it is released by both the sender and receiver processes. A fragment of the modified entity type is in Figure 2, where the message types *chnls* and *chnlr* are declared in lines 5–6, and the resume condition to allocate channels is in lines 20–21.

Complex resume conditions typically lead to concise programs and also allow the enabling condition for an event to be expressed directly. In the resource manager example, if unmatched messages were buffered internally in the entity, the analyst would have to design the data structures to store the requests: if channel numbers belonged to a small range and each channel was requested by at most one pair of processes, an array implementation would be the most efficient. In contrast, if the range was large (any positive integer) and multiple pairs could simultaneously request the same channel number, a hash table may be more appropriate. The compound resume condition allows these considerations to be postponed until the analyst has a chance to collect more information. If channel requests are relatively conflict-free and both processes tend to request their access at approximately the same time, then the overhead for the compound condition is small and it may not be necessary to refine the code any further. However, in general, the efficiency of a model may be improved by refining resume conditions with complex guards to a simpler form that uses a single message type and a local guard[2].

We present one refinement of this model: assume that although there is heavy conflict for the channels, most requests arrive in matched pairs. In this case, it may be desirable to modify the resume condition such that requests are removed from the buffer in matched pairs and are buffered internally if the channel is unavailable. The resume condition in lines 20-21 may be modified as shown below, where the actions of the entity are specified using pseudo-code.

```
20    or csend= mtype(chnls)
21      and crecv= mtype(chnlr) st (crecv.cno==csend.cno)
          { if (cfree[csend.cno])
              inform requesting processes;
          else buffer matched request for channel cno;
          ⋮
```

The modified version is more efficient because a programmer is free to organize the internal queue such that whenever a channel is returned, the queue can be searched efficiently to find a buffered request pair. As the data structures used to implement the external message buffer are not under programmer control, in general it is less efficient to find an enabling message(s).

3.5 Example

We develop a complete Maisie model for a simple closed queueing network benchmark[10], henceforth referred to as CQNF. The network consists of N fully connected switches. Each switch contains Q FIFO servers connected in tandem. 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. Each switch is initially assigned J jobs.

The Maisie model of this network consists of two primary entity types: a *server* entity that models each server in the tandem queue and a *router* entity that routes a job after it has completed service at a queue. Each job in the network is modeled by a sequence of messages. The complete Maisie program for this example is in Figure 3. Every Maisie program must include an entity type called **driver** that serves as the 'main' program. In this model, the driver entity creates the *router* and *server* entities(lines 12–16). As the *server* and *router* entities communicate with each other, each must have the entity-identifier for the other. The appropriate id is passed to the entity as either an entity parameter (as when creating the *router* entities in line 16) or in a separate message (as for the *server* entities in line 19). The driver entity also creates an instance of entity *basic_stats* (line 11), which is used to compute the average system time spent by a job in a queue. A Maisie program terminates when the simulation clock exceeds the value specified by function **maxclock**; this model is simulated for 10000 time units (line 10).

The *server* entity (lines 37-49) simulates fifo service of an incoming job simply by executing an appropriate hold statement (line 46). After servicing a job, the entity sends the job to the next server in the queue or, if it is the last server, to its *router* entity (line 47). The jobs initially allocated to each switch of the physical network are allocated to the corresponding *router*. On

[2]As discussed in section 7, a simple monitoring facility can be transparently attached to a Maisie model to track the 'cost' of executing specific wait statements in an entity.

```
#include "mayc.h"
#define N               16
#define J               32
#define Q               10
extern entity basic_stats{};

entity driver{}
{  ename q[Q+1][N], stat1;
   int i,j;
   maxclock("10000");
   stat1 = new basic_stats{"Average System Time"};
   for(i=0;i<N;i++)
      for(j=0;j<Q;j++)
        q[j][i]=new server{10};
   for(i=0;i<N;i++)
      q[Q][i]= new router{i,J,stat1,q[0]};
   for(i=0;i<N;i++)
      for(j=0;j<Q;j++)
         invoke q[j][i] with idmsg{q[j+1][i]};
 }

 entity router{myid,njobs,statid,qids}
   int myid, njobs;
   e_name statid, qids[N];
 { int i;
   message job{int dep;} j1;
   for(i=0;i<njobs;i++)
      invoke qids[myid] with job{sclock()};
   for(;;)
      wait until mtype(job)
      { j1=msg.job;
        invoke statid with value{sclock()-j1.dep};
        invoke qids[urand(0,N)] with job{sclock()};
      }
 }

 entity server{mean}
   int mean;
 { message job{int dep;} j1;
   message idmsg{ename id;};
   ename nextid;
   wait until mtype(idmsg) nextid= msg.idmsg.id;
   for (;;)
      wait until mtype(job)
      { j1=msg.job;
        hold(expon(mean));
        invoke nextid with job=j1;
      }
 }
```

Figure 3: Maisie model of CQNF

```
50 entity queue{mean,nsrvr}
51  int mean,nsrvr;
52 { int i,t1,lastj[Q];
53   ename nextid;
54   message job{int dep;} j1;
55   message idmsg{ename id;};
56
57   wait until mtype(idmsg) nextid=msg.idmsg.id;
58   for(i=0;i<nsrvr;i++)
59      lastj[i]=0;
60   for(;;)
61      wait until mtype(job)
62      { j1=msg.job;
63        t1=j1.dep;
64        for(i=0;i<nsrvr;i++)
65        { lastj[i]=MAX(t1,lastj[i]) + expon(mean);
66          t1=lastj[i];
67        }
68        hold(t1-sclock());
69        invoke nextid with job{j1.dep};
70      }
71 }
```

Figure 4: Refined CQNF Model

being created, a *router* entity routes all these jobs to its queue (line 27–28). Subsequently, for each incoming job, it forwards the job to one of the N switches with equal probability (line 33). Also, the total time spent by the job in the queue is sent to the statistics collection entity (line 32).

If the number of servers at a switch is large, it is typically more efficient to use a single entity to simulate the multiple servers at a switch (Figure 4). The *queue* entity maintains an array *lastj* to track the time at which the last job serviced at the queue departs from each server. The departure time of a job at each server is computed in line 63–67 and the service of the job is simulated with the hold statement in line 68. Note that the *router* entity remains unchanged in the refined model while the driver entity requires minor modifications in creating new entities. Also, due to different random number sequences, these two models may not behave identically. However, maintaining a separate seed for each server solves the problem. Figure 5 plots the execution time for sequential implementations of both models for a configuration with 16 switches and 32 jobs at each switch. As seen from the figure, the refined model performs significantly better as the number of servers is increased.

4 Parallel Simulations

Sequential execution of a Maisie program is straightforward: future messages are stored in increasing order of their timestamps in a global event-list. At every step, the entry with the earliest timestamp is removed from the list and the corresponding message is delivered to the destination

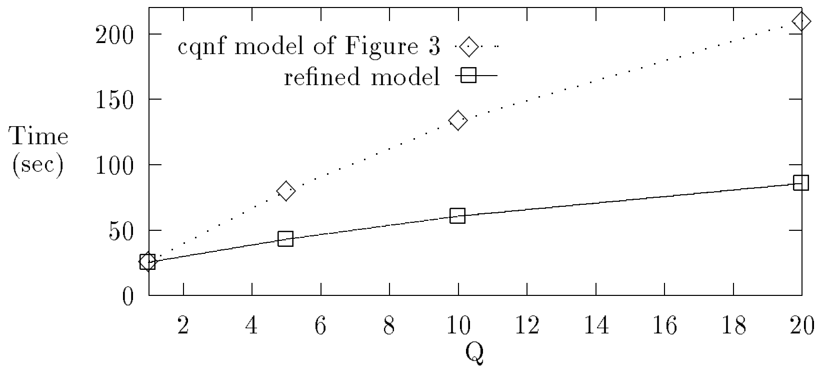

Figure 5: Sequential execution of CQNF ($N = 16, J = 32, T = 10^4$)

entity[3]. For parallel execution of the model, the event-list is physically distributed across the parallel architecture. While each node of the parallel architecture maintains its local simulation clock, messages must still be processed in the global order of their timestamps. This is guaranteed by the underlying distributed simulation algorithm.

Distributed discrete-event simulation algorithms are broadly classified into *conservative* and *optimistic* based on their tolerance of *causality errors*. Conservative algorithms do not permit any causality error: a simulation object (also called an LP) cannot process a message until the system can guarantee that it will not subsequently receive a message with an earlier timestamp. This constraint may introduce deadlocks, which are typically handled by incorporating deadlock detection[25] or deadlock avoidance[25, 9, 10] mechanism into the simulation algorithm. Optimistic algorithms[21, 11] allow an LP to process messages out of order; causality errors are corrected by using rollbacks and recomputations. Implementations of optimistic algorithms are usually more difficult because they require complex mechanisms for detection and handling of causality errors, termination detection, exception handling, and memory management. A comprehensive discussion of parallel discrete-event simulations may be found in [16, 25].

A sequential Maisie implementation may be refined to a parallel implementation simply by allocating the entities among available processors, and executing the program in the parallel environment. The runtime system for the parallel environment has two major responsibilities: providing interprocess communication (IPC) facilities and implementing the distributed simulation algorithm. The Maisie IPC facilities have been designed to operate in conjunction with existing IPC packages like UNIX Sockets, PVM [18], and the Cosmic Environment [33]. They can be easily modified to work on top of other distributed operating system kernels. The distributed simulation algorithm is implemented via a set of routines that are essentially transparent to the Maisie programmer. Entities mapped to a common processor are simulated sequentially and entities on different processors may be synchronized using a variety of different protocols.

The rest of the section discusses how a Maisie program can be executed transparently using three different protocols. In the interest of brevity, we do not describe the respective protocols, but

[3]For simplicity, we assume that all timestamps are unique; if not, alternative criteria must be used as suggested in [25] to select the next message.

simply indicate how the information required by each algorithm may be extracted from the Maisie program. Maisie implementations using conservative algorithms are described in [23]; optimistic implementations are described in [4].

4.1 Null-Message Algorithm

A Maisie program can be executed using either *lazy* or *demand-driven* variations for the *null*-message algorithm[25]. In order to implement any *null*-message scheme, each LP must be aware of the set of its source and/or destination LPs. (In the absence of this information, *null* messages may have to be broadcast, making the implementation inefficient for simulation of a sparsely connected physical system). For each LP in the simulation, the runtime system implicitly maintains two variables, the *source-set* and the *dest-set* which respectively refer to its set of source and destination entities. We briefly indicate how the two sets are maintained for each entity (or LP) by the runtime system.

In order for a Maisie entity LP_s to send a message to LP_d, LP_s must have the identifier for LP_d. As an entity identifier may only be stored in a variable of type ename, the *dest-set* of an entity is assumed to comprise of all ename variables and is maintained transparently by the runtime system. To determine the *source-set* of an LP, it is sufficient to determine the set of entities that have access to its name. An entity, say LP_s, can gain access to the identifier for another entity, say LP_d in one of two ways: if LP_s creates LP_d, or if LP_s receives the information in a message from another LP (including LP_d). In either case, the problem is to add LP_s to the *source-set* of LP_d. In the first case, this is done transparently by implicitly including LP_s in the initialization information used to create LP_d. In the second case, an entity that sends the identifier LP_d to say entity LP_s must first execute a system call which updates the *source-set* of LP_d to include LP_s[23]. The runtime system can detect violations of the above rule and take appropriate action including abnormal termination of the simulation. On termination of an entity, the system automatically removes the name of the terminated entity from all *source-set*s and *dest-set*s.

The overhead associated with maintaining the *source-set* and *dest-set* information is negligible if the communication topology in the application is static, as is the case for many simulations. Other than the system calls required to maintain the *source-set* information for an entity, the simulation algorithm is completely transparent to the Maisie programmer.

4.2 Conditional Event Simulation

Chandy and Sherman[10] have described a conservative simulation algorithm that does not rely on *null* messages to guarantee progress. Instead the algorithm distinguishes between definite and conditional events in a simulation. Generation of a message by an LP_c is a definite event, if it depends only on its current state (and the sequence of messages that have been received by the LP) and is not affected by any subsequent messages that may be received by it. An event that is not definite is conditional. If at some point in the computation, the next event of every entity is a conditional event, the simulation may deadlock. Rather than use *null* messages to avoid deadlocks, the algorithm suggests that cond_x, the timestamp on the earliest conditional event for LP_x be recorded consistently for all LPs. The minimum cond_x represents the earliest conditional event in the system, which may then be transformed into a definite event.

In order to execute a Maisie program using the conditional event algorithm, it must be possible to distinguish between definite and conditional events, as also to determine the cond_x for each entity. The resume conditions specified in the wait statement may be used to transparently distinguish definite events from conditional events. If the resume condition includes only the timeout message, the event is definite, otherwise it is treated as being conditional. In the former case, the simulation time of the entity is immediately incremented by the wait-time specified in the wait statement and the action associated with the receipt of the timeout message are executed to generate the appropriate messages as definite events. Conversely, if the resume condition indicates a conditional event, cond_x for the entity may be determined from the wait-time specified in the corresponding wait statement and the earliest message in the input buffer. Once the definite events and the cond_x have been determined, the program may easily be executed using the conditional event algorithm.

4.3 Space-Time Simulation

In order to transparently execute a Maisie program using an optimistic algorithm, the runtime system must perform three primary tasks : checkpointing, recomputation, and determining the duration over which the simulation has converged. Functionally, checkpointing is transparent to the programmer; an entity changes its state on receipt of a message, and the old state is saved in a timestamped queue. Rollback is implemented automatically by tracking the timestamps on the messages delivered to each entity and the algorithm for detecting simulation convergence[11] can easily be made transparent to the Maisie programmer. A detailed description of the optimistic implementation has been provided in [4].

Optimistic executions may also require runtime support to deal with abnormal termination of a model due to errors caused by optimistic execution of an entity. The current version of Maisie does not provide specific constructs to deal with this problem.

4.4 Example

We modify the Maisie model described in section 3.5 for parallel execution. The only modification that is needed is to change the driver entity to specify remote creation of the *queue* and *router* entities. The modified driver entity is shown in Figure 6 where the **at** clause has been added to the **new** statement (lines 13 and 15) to indicate the processor number on which the corresponding entity is to be created and executed. Figure 7 shows the speedup obtained with the parallel implementations

```
7  entity driver{}
8  {  ename rtr_id, q[N], stat1;
9     int i,j;
10    maxclock("10000");
11    stat1 = new basic_stats{"Average System Time"};
12    for(i=0;i<N;i++)
13      q[i]=new queue{10,Q} at i;
14    for(i=0;i<N;i++)
15      rtr_id = new router{i,J,stat1,q} at i;
16    for(i=0;i<N;i++)
17      invoke q[i] with idmsg{rtr_id};
18 }
```

Figure 6: Parallel CQNF: Driver Entity

of the CQNF model using the Space-Time algorithm. The model contains N=16 switches, where each switch contains Q=20 servers and J=32 jobs. These and other parallel experiments reported in this paper were conducted on a Symult Series 2010 hypercube. Each node of the multicomputer uses a Motorola 68020 cpu and has 4MB memory. The sequential version used for the comparison was executed on a single node of the same machine using a sequential simulation algorithm.

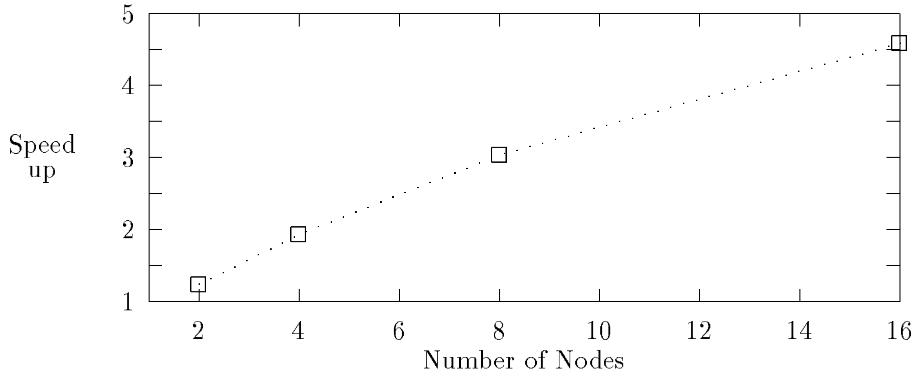

Figure 7: Speedup for refined CQNF ($N = 16, Q = 20, J = 32, T = 10^4$)

5 Optimizations for Optimistic Algorithms

An optimistic simulation is rolled back if the runtime system detects that a message sequence delivered to an LP in the simulation is different from the message sequence delivered to the corresponding PP in the physical system (or its model). This may be either because the former contains a message that is not present in the latter (or vice-versa), and/or because the message sequence in the simulation is a permutation of the sequence in the physical system. In the first case, recomputations are typically unavoidable. However, rollbacks may be reduced in the second case as explained subsequently. In the remainder of the paper, we restrict attention only to the second type of rollbacks.

The term *rollback distance* refers to the total number of events that must be recomputed when a rollback is initiated. Reducing the rollback distance increases the efficiency of the simulation by reducing recomputation and state saving overheads. This section describes a variety of ways to reduce the rollback distance.

Let r_1 be a subsequence of the correct sequence of messages that must be 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 represent some permutation of sequence r_1 such that $r_1 \neq r_2$. Let 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.

- $F_1 \neq F_2$ and $s_1 \neq s_2$
- $F_1 \neq F_2$ and $s_1 = s_2$

- $F_1=F_2$ and $s_1 \neq s_2$
- $F_1=F_2$ and $s_1=s_2$

The term *straggler* message is used to refer to a message that is delivered to an LP after a message with a larger timestamp has been delivered. Let (m_w, t_w) represent the earliest straggler message in r_2. In a typical optimistic implementation like TWOS[22], delivery of sequence r_2 rather than r_1 would cause recomputation of LP_a from a state with a timestamp smaller than t_w, in each of the four cases. However, as we show subsequently, the rollback distance can be considerably reduced in the second and third case, and completely eliminated in the last case. The term *semantic rollback* refers to a rollback whose rollback distance can be made smaller than the distance determined by the timestamp of the straggler message that initiates it.

The semantic rollback optimizations differ from those implied by lazy message cancellation[17]. Lazy cancellation prevents cascading rollbacks by not canceling messages that are regenerated after a rollback. A semantic rollback directly reduces the rollback distance for the LP that receives a straggler message. As the recomputation following a rollback also incurs state saving overheads, reductions in the rollback distance help to reduce the overall state saving overheads.

5.1 Transparent Optimizations

This section describes conditions under which a rollback may be identified transparently as a semantic rollback by the runtime system. Assume that entity LP_a executes a wait statement at simulation time t. The following two variables are defined for every entity:

- $mset_a(t)$: set of enabling messages for LP_a at time t.
- $tres_a(t)$: timestamp(s) of the enabling message(s) accepted by the entity when it resumes execution after executing wait statement at t.

Variables *mset* and *tres* are automatically maintained for every entity. Henceforth, we will drop the subscript on *mset*, when the corresponding entity is uniquely indicated by the context.

Assume that a straggler message (m_w,t_w) is received by an entity when its simulation time is t_n; by definition $t_w < t_n$. Let t_l be the latest time preceding t_w at which the object's state was saved. As traditional optimistic simulators set the simulation time of an object equal to the timestamp on the last message delivered to the object, receipt of m_w would immediately initiate a rollback to t_l. In contrast, the simulation time of a Maisie entity is advanced only when the entity removes an enabling message from its buffer. Depositing a message in the message buffer of an entity does not affect its simulation time. It follows that arrival of the straggler message in the optimized Maisie implementation would cause a rollback 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 a few examples.

Consider the preemptible priority server of section 3.3 that receives messages of type *high* or *low* to represent requests of different priority, where arrival of a *high* message may preempt service of a *low* message. 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. Then, *mset*(5) for the server includes only timeout messages; other messages including (9,*low*) and (7,*high*) that are received by the server, will be stored in its message buffer until it receives a timeout message. The delivery order for these two messages is immaterial to the correctness of the simulation, as long as message (7,*high*) arrives at the server before simulation time 15 (note that *mset*(15) includes messages of type *high*). Furthermore, if message (14,*high*) is delivered to the entity after simulation time 15, even though the message belongs to *mset*(15), rollback is unnecessary as *tres*(15)=7, due to the server initiating the service of message (7,*high*).

If the resume conditions of a wait statement only contain local guards, the overhead of maintaining the *mset* of an entity is low. As most entities contain a small number of message types (almost never exceeding 32), the *mset* can be typically saved in a single word by using a unique bit-mask for each message type defined by the entity. An additional word is required to store *tres* for every recorded state. The processing overhead is also small: for each recorded *mset* one logical *and* operation and a comparison is required to determine if a straggler message (m_w,t_w) belongs to the corresponding *mset* and one comparison is required to determine if t_w is greater than the recorded *tres*.

If a resume condition includes message parameters, computing the *mset* of the entity and determining if a straggler message belongs to a recorded *mset* are both more expensive than if the guard is local. In this case, the guard is used to create a parameterized function, where the parameters correspond to the entity variables and message parameters referenced in the guard. Assume t_l, t_n, t_w and t_r as defined previously. In order to determine if a straggler message belongs to the entity's *mset*, the function must be executed for each recorded *mset* in the interval [t_l,t_r].

Similarly, a ranker may be used in a resume condition to specify the order in which messages of a given type are to be serviced. This would permit the runtime system to initiate a rollback only if the rank of the straggler message is higher (or lower) than that of the enabling message accepted by the entity. The overhead due to recording and scanning the recorded *mset* is of similar magnitude to the previous case, as the ranking parameter and rank of the enabling message can be recorded as a boolean expression. Additional overhead is incurred in maintaining the message buffer as an ordered queue. (Note that different resume conditions in an entity may specify different rankers for messages of a given type.) To minimize unnecessary overheads, syntactic tags are used to ensure that this condition is known at compile time. This allows the system to maintain an ordered queue *only when the queue would otherwise need to be maintained by the programmer*. Thus the queue maintenance does not really contribute to additional overhead.

Example We present experimental measurements of the refined CQNF model (Figure 4) to illustrate the effectiveness of some of the transparent optimizations described in this subsection. The number of nodes used in the parallel execution of this model is equal to N, the number of switches in the system. The sequential version used for comparison was executed on a single node of the same machine using a sequential simulation algorithm.

Figure 8 presents the measurements for a CQNF model with 16 switches. The figure plots the speedup as a function of the number of servers in each queue. The speedup is plotted for both optimized and non-optimized parallel implementations of the Space-Time algorithm described in section 4.3. The event-granularity (i.e. the amount of computation associated with an event) increases as Q increases. The performance of the optimized version improves as the event granularity increases primarily because a larger number of semantic rollbacks are identified. As seen from the figure, the transparent optimizations have a significant impact on improving the performance of the simulation, particularly for models with a large event granularity. A detailed discussion of the implementation of the optimizations together with experimental measurements of their utility in reducing the completion time of optimistic simulations of stochastic benchmarks may be found in [7].

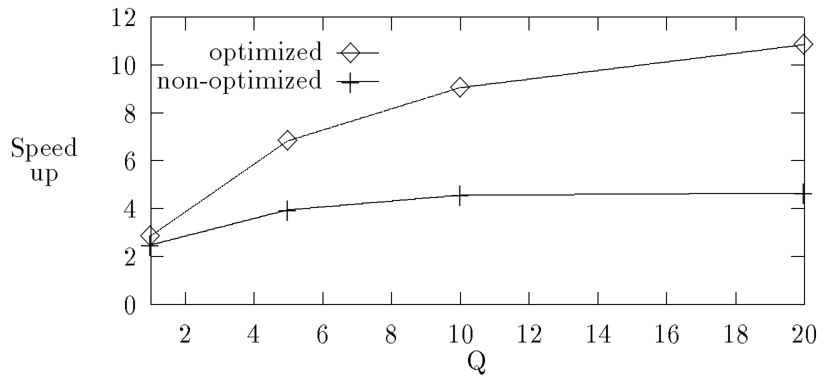

Figure 8: Effectiveness of Optimization

5.2 User-specified Optimizations

The optimizations described in the previous subsections are useful in identifying semantic rollbacks when a message is *deposited* in the message buffer of an entity in an incorrect order. In this section, we extend the optimizations to include situations where the timestamp on a straggler message is less than the simulation time of a Maisie entity.

Probe Messages A *probe* message refers to a message whose processing does not alter the state of the recipient entity. A probe message is typically used to obtain state information about the recipient entity, such as whether it is *active* or *idle*. Processing a *probe* message in an incorrect order would typically result in situations where $F_1=F_2$ but $s_1 \neq s_2$. Although it is sometimes possible for the runtime system to transparently detect *probe* message, it is more efficient to use syntactic tags for this purpose. A message type is declared to be a probe by preceding its declaration with the keyword **probe**. The following fragment illustrates the declaration and use of a probe message-type called *status*:

probe message *status*{**ename** *jobid*;};

⋮

wait until mtype(*status*)
    invoke msg.*status.jobid* **with** *reply*{*idle*};

Variable *idle* denotes the current status of the entity. If a straggler message (m_w,t_w) is identified as a probe, the message is processed in the state that is saved at or immediately prior to t_w. The subsequent events that have already been processed by the entity do not need to be canceled. Once again, if the state of the entity is saved after every event, implementing this optimization adds negligible overhead but may reduce recomputation and state saving overheads.

Commutative Messages It is sometimes possible to process straggler messages that modify the state of the recipient process without initiating a rollback; such a message is referred to as a *commutative* message. As an example, 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_1),(18,7,LP_2) where the message parameters respectively represent the message timestamp, desired service duration and the requesting LP. 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; the message (18,7,LP_2) is said to be commutative, and the permuted sequence r_2 is said to be *compatible* with the correct sequence.

As another example of commutative messages, consider a bounded buffer that receives data from a producer process via *put* messages and requests for the data from a consumer process via *get* messages. Let (p1,p2,c1,p3,c2) be the 'correct' message sequence, where p1, p2, p3 represent *put* messages and c1, c2 *get* messages. Sequences (p1,c1,p2,p3,c2) and (p1,p2,p3,c1,c2) are both compatible with the correct sequence.

Our aim in this section is to suggest language primitives that allow a programmer to identify straggler messages that are commutative. For this purpose, we define a separate, optional section of an entity called the *warp* section. This section consists of a set of warp statements, each of which is syntactically similar to a resume statement. Each warp statement defines a warp condition and warp actions, where the former is a temporal predicate and the latter is a C or Maisie statement. A warp statement has the following form:

mtype(m_t) **st** b_i [**in** (t_i,t_j)]
                    statement ;

A warp condition includes a message type, a guard and an optional temporal component that defines a time interval. If omitted, the interval is assumed to be the single time instant corresponding to the timestamp of the straggler message. Note that b_i, t_i, and t_j may include message parameters. A straggler message of type m_t is commutative, if the guard in its warp condition is continuously true at every instant in the corresponding time interval. Assuming that t_n, t_l, t_r and t_w are defined as in the previous section, a compatible straggler message is processed in the state of the recipient entity saved at time t_r. In addition, to ensure that the effect of the straggler message is included in the final state of the entity, the specified warp actions must be executed in the state of the entity at t_n. If an entity includes a *warp* section, the runtime system is required to save the state of the entity after every event so that the warp condition may be evaluated over the specified interval.

We illustrate these ideas in the context of a FIFO server. Figure 9 presents the entity definition for a FIFO server. On receiving a *request* message, the entity simulates its service by executing an appropriate hold statement and sends a *done* message to the requesting process. The warp section includes a warp condition for message type *request* which indicates that the entity may process a straggler *request* message, if it was idle over the duration that corresponds to the service-time of the straggler[4]. The warp actions ensure that the count of messages serviced by the entity is updated correctly.

Dead States The state of an entity is typically saved after each event to minimize rollback distance. However, some states in an entity may be such that no recomputation ever begins from that state; such a state is referred to as a *dead* state. Consider a timeout message that is scheduled as a definite event. From the definition of a definite event in section 3.3, it follows that if the sequence of messages received by the entity preceding some timeout message is correct, the timeout

[4]Every Maisie message includes a field called *tstamp* that refers to the simulation time at which the message was sent.

```
#define R msg.request
entity server {}
{ int nojobs=0, idle=true;
  ename jobid;
  message request {int stim; ename jobid; } ;
  for(;;) {
    idle=true;
    wait until mtype(request) {
        idle=false; jobid=msg.request.jobid;
        hold(msg.request.stim);
        invoke jobid with done;
        nojobs=nojobs+1;}
    }

    warp  {
      mtype(request) st (idle) in (R.tstamp, R.tstamp+R.stim)
        nojobs=nojobs+1;
    }
}
```

Figure 9: A FIFO Server with *Warp* Section

message must also be correct; the timeout message can never be the first incorrect message. In other words, the state immediately preceding the receipt of the timeout message is a dead state that will never be used to initiate a recomputation, and hence need not be saved. For entities with large states, this may be a significant improvement. For a specific application, it may be possible for an analyst to identify other states as dead states; typically these states relate to the scheduling of definite events. The programmer may explicitly flag some resume statement r_i, to indicate that if the entity resumes its execution by executing r_i, the preceding state need not be saved. Such a resume statement is indicated simply by replacing keyword **mtype** in the resume condition by keyword **ctype**. Note that, in the worst case, incorrectly labeling a state as a dead state may degrade the completion time by increasing the rollback distance, but will not affect its correctness. Of course, if the entity also includes a warp section, the dead states must nevertheless be saved to allow the warp condition to be tested exhaustively.

6 Optimizations for Conservative Algorithms

The performance of conservative algorithms is influenced significantly by the lookahead properties of the LPs in the model[14]. Lookahead is defined as follows: assume that the simulation interval for the model is $[0,H]$. A process is said to have lookahead ϵ, if given the state and inputs to the process at time t, $0 \leq t \leq H$, the outputs of the process can be predicted in the interval $[t, t+\epsilon)$. In order to have good lookahead, it is important that a process have information about the state of each of its predecessor process. For instance, consider a fifo server that has only one predecessor process. Such a server has excellent lookahead: whenever it receives a job it can immediately predict the time at which it will depart. However, if the server has two predecessors, say P and Q, the server can predict the departure time of an arriving job only after it has received a message from both P and Q. If the predecessors feed the server at different rates, the server must explicitly synchronize with its predecessors to determine the departure time of an incoming job. This section describes optimizations which allow the lookahead for some type of objects to be extracted transparently by the runtime system. In addition specific primitives are provided to allow programmers to encode lookahead in the definition of an entity.

Assume that the *source-set* and *dest-set* data structures are maintained for each process as described in section 4.1. A basic conservative algorithm may be implemented transparently as follows: whenever an entity sends a (non-null) message, say (m_i,t_i) it also sends a *null* message timestamped t_i to every other entity in its *dest-set*. A message say (m_i,t_i) is delivered to an entity only if the entity has received some message timestamped t_i or greater from every entity in its *source-set*. As long as every cycle of entities in the model has at least one lookahead process, progress is guaranteed[25]. The basic scheme outlined above may perform poorly for many applications. The performance can be improved if lookahead for the various entities is exploited aggressively. In a Maisie program, lookahead for an entity may often be extracted transparently: if the *mset* of a suspended entity only contains timeout messages, the wait-time specified in the most recent wait statement represents its lookahead and may be used to advance its simulation clock even in the absence of a message from all members of its *source-set*.

For some entities, it may be possible to extract the lookahead only using application-specific information. For instance, presampling of random numbers may be used to generate lookahead for a server whose service time is sampled from random distributions[26, 24]. Every Maisie entity includes a compiler-defined local variable called **lookahead**. When an entity schedules a definite future event, the runtime system automatically updates this variable to reflect the lookahead time for the entity. In addition, an entity may explicitly compute its lookahead and store it in this variable before executing a wait statement. The control graph model described in [12] uses a similar feature to permit automatic extraction of lookahead. Figure 10 illustrates lookahead computation for a priority server. When the server is idle, its lookahead is the minimum of the presampled service time for the next request (represented by variable *htime* and *ltime* for *high* and *low* messages respectively). When serving a *low* message, its lookahead is the minimum of the remaining service time for the request (*rtime*) and the presampled service time (*htime*) for any *high* message that may interrupt it. By setting *ltime=MAXINT* when servicing a *low* message and *rtime=MAXINT* when the server is idle, its lookahead in the preceding two cases is simply the minimum of *htime*, *ltime*, and *rtime* as shown in the figure (line 12). The server uses a hold statement to service a *high* message, where its lookahead can be computed automatically by the runtime system (line 20). A detailed description of the optimizations with conservative implementations may be found in [23].

7 Implementation Issues

Maisie has been implemented on both sequential and parallel architectures. For sequential simulation, the splay-tree data structure is used to implement the global event-list. For conservative simulations, the algorithms described in sections 4.1 and 4.2 as well as the optimizations discussed in section 6 have been implemented[23]. For the optimistic implementation using the space-time algorithm, the current implementation supports the transparent optimization discussed in section 5.1; the implementation of user-specified optimizations is in progress.

This section discusses the implementation of wait statements whose efficiency has a significant impact on the efficiency of sequential and parallel Maisie programs. The Maisie wait statement uses the concept of *interrogative* simulation, where an entity autonomously determines the order

```
entity server { hmean, lmean }
    int hmean, lmean;
{
    message high { ename hisid; } ;
    message low { ename hisid; } ;
    ename hjobid, ljobid;
    int rtime, htime, ltime, otime, busy=0;
    htime=expon(hmean);
    ltime=expon(lmean);
    rtime=MAXINT;
    for(;;)
    {   lookahead=MIN(htime, ltime, rtime);
        if(busy)
            otime=rtime + sclock();
        wait rtime until
        {   mtype(high)
            { if(busy)
                  rtime=otime − sclock();
              hjobid=msg.high.hisid;
              hold(htime);
              htime=expon(hmean);
              invoke hjobid with done; }
            or mtype(low) st(!busy)
            {  busy=1; ljobid=msg.low.hisid;
              rtime=ltime;
              ltime=MAXINT }
            or mtype(timeout)
            {  busy=0; rtime=MAXINT;
              invoke ljobid with done;
              ltime=expon(lmean); }
        }
    }
}
```

Figure 10: A Priority Server with Lookahead

in which messages are removed from its buffer. In contrast, an *imperative* algorithm typically delivers a message to the data-space of the destination entity at the simulation time specified by the message timestamp; if the entity is not ready to process the message, it must be buffered internally. As discussed in the previous sections, the Maisie wait statement facilitates the design of concise programs and is useful in reducing simulation overheads in parallel execution of Maisie models. However, interrogative algorithms may be less efficient because the cost of selecting an enabling message (which is not necessarily the message with the earliest timestamp) may be higher than for an imperative algorithm[27]. We examine the factors that contribute to this cost and discuss techniques that reduce it.

Let *lbuffer* refer to the number of messages that must be inspected from the entity's buffer before an enabling message is identified. For an imperative algorithm, the *lbuffer* is 1: messages are stored in the buffer in order of their timestamps, and the enabling message is simply the message at the head of the buffer. A naive implementation of the interrogative algorithm may yield an upper bound on *lbuffer* that is proportional to the total number of messages in an entity's buffer; however, in many cases, a constant upper bound can be derived.

The message buffer for a Maisie entity is implemented as a number of separate lists, one for each message type defined by the entity. All messages in a given list are ordered by their timestamps (or alternately by the ranker), and messages in different lists are also linked in the order of their timestamps. Consider a wait statement, such that every resume condition in the statement has a local guard and references a single message type. In this case, the *mset*, that is the set of enabling messages for the corresponding entity, is completely specified by a few message types: a message of type m_t belongs to the *mset* only if the wait statement executed by the entity included a resume condition that referenced m_t and if the value of the corresponding guard was *true*. This implies that the *lbuffer* has a tight upper bound given by the number of message types defined for the entity. If the wait statements executed by an entity do not restrict the messages that may be accepted by the entity, its *lbuffer* is exactly 1.

For wait statements with local guards, the time to identify an enabling message may be further reduced by implementing and searching the *mset* efficiently. As most entities define a small number of message types (almost never exceeding 32), the *mset* can be implemented by a single-word bit-mask. Further, the *mset* is stored outside the data-space of the entity allowing the runtime system to determine if the buffer for an entity contains an enabling message without having to go through a context-switch.

If a resume condition includes message parameters, the *mset* of an entity can no longer be specified by message type alone. For instance, if the guard for message type m_t references a message parameter, the guard must be evaluated for successive m_t messages from the buffer until some enabling message (not necessarily of type m_t) is identified or it is determined that no resume condition is enabled. In addition to the number of message-types, the *lbuffer* for such a wait statement will also depend on the number of messages of type m_t in the buffer. Similarly, the *lbuffer* for a wait statement with a compound resume condition may also depend on the number of messages of a given type that are present in the buffer.

Maisie provides a transparent facility to monitor the *lbuffer* for each wait statement; collected statistics include the average, maximum, and median values of the *lbuffer* for each wait statement. Complex resume conditions with a large *lbuffer* may be simplified such that the messages are buffered internally and can be searched efficiently by the programmer. The built-in monitoring facility can be used to determine if the elaboration is likely to yield any benefit. As the code for internal buffering and its efficient searching can be complex, it is desirable to postpone this refinement until appropriate information about its possible impact is available.

Example We present the results of an experimental study to illustrate the relationship between the execution time of a sequential simulation model and its average *lbuffer*. Consider the resource manager model of Section 3, where the manager is initialized with 10 units of the resource, and each *preq* message requests n units, where n is uniformly distributed in the interval [1,10]. We use three different models of the manager entity: Model 1 is as described in section 3.2, where the resume condition references a message parameter and finding an enabling message requires inspection of individual *preq* messages in the buffer. In the second model, the resume condition is simplified by removing the guard; if an incoming request cannot be satisfied by the manager it is stored in an internal buffer that is implemented as a linked-list using the dynamic memory allocation routines provided by C. The third model is similar to the second except that the internal queue is implemented using arrays (which requires a priori knowledge of the upper bound on the number of buffered requests).

Number of job entities	resume condition with guard		resume condition without guard	
	Model 1		Model 2	Model 3
	Average *lbuffer*	Execution time (sec)	Execution time	Execution time
1	1	0.77	0.77	0.77
10	6.383	9.03	9.48	8.43
20	13.29	19.97	18.72	17.64
30	20.20	36.12	30.67	27.65

Figure 11: Effect of *lbuffer* on execution time

Figure 11 shows the execution time for each of the three models as the number of job entities is increased from 1 to 30, where each job entity generates 2000 requests. For these experiments, the completion time were measured on a SUN Sparc/IPC workstation. As expected, the models with a local guard do not have a significant performance gain for configurations with a small value of *lbuffer*, but can be upto 25% faster as the average *lbuffer* increases to 30. This supports our contention that resume conditions with local guards do not incur a performance penalty and that refinements of resume conditions with non-local guards are desirable only if the average *lbuffer* is expected to be large. Note that for models with a small average value of *lbuffer*, the performance of the linked-list implementation (Model 2) is *worse* than that of Model 1 due to the overheads of calls to the C *malloc*() and *free*() routines; however as the average value of *lbuffer* increases, the search on the internal queue can be implemented more efficiently resulting in overall performance improvements.

8 Conclusion

Simulations are typically large and complex programs and the design and validation of parallel simulations is particularly hard. This paper described a language called Maisie to support the design of parallel simulations by iterative refinements of a model, where the refinements are used primarily to improve the execution efficiency of the model. Innovative features of Maisie include the ability of an entity to inspect specific messages from its message buffer and the use of compound resume conditions. These constructs allow an entity to remove a message from its buffer only when the entity is ready to process the message. Appropriate use of the wait statement leads to succinct programs and reduces program development time.

Monitoring facilities may be transparently attached to an entity to track the cost of evaluating each resume condition in a wait statement. This is another innovative feature of the language that allows a programmer to selectively refine certain parts of the model to improve its efficiency. The initial program is executed using a sequential simulation algorithm and may be tested on a workstation. If the completion time of the sequential simulation is not acceptable, it may be refined for parallel execution.

The initial transformation of a Maisie model to a parallel implementation simply allocates Maisie processes among available processors. At this stage, the simulationist need not be concerned with the specific simulation algorithm that is used to execute the program on the parallel architecture. Maisie is among the few languages that allow a model to be executed using either conservative or optimistic algorithms. After identifying the most suitable simulation algorithm, the final refinements to the model are to use application and algorithm specific information to reduce the completion time for the simulation program. An optimistic implementation attempts to reduce recomputation and state saving overheads by identifying semantic rollbacks and dead states. A conservative implementation reduces synchronization overheads by distinguishing between definite and conditional events, and by aggressively exploiting the lookahead in an application. To the best of our knowledge, Maisie is the only language that supports optimizations to reduce the overhead of both conservative and optimistic execution of parallel discrete-event simulation models. The paper also presented a brief summary of the measurements on the effectiveness of some of the optimizations in reducing the completion time for the simulation of a simple queueing network.

Acknowledgments

We are grateful to Mary Vernon and the referees for their many useful comments. We are also grateful to Mani Chandy for many helpful discussions and to Chuck Seitz for access to the Symult S2010 multicomputer at Caltech.

References

[1] Marc Abrams. The object library for parallel simulation (OLPS). In *Proceedings of the 1988 Winter Simulation Conference*, pages 210–219, San Diego, California, December 1988.

[2] G.R. Andrews. Synchronizing resources. *ACM TOPLAS*, 3(4):405–430, October 1981.

[3] Dirk Baezner, Greg Lomow, and Brian W. Unger. Sim++: The transition to distributed simulation. In *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, pages 211–218, San Diego, California, January 1990.

[4] R.L. Bagrodia, K.M. Chandy, and W. Liao. A unifying framework for distributed simulations. *ACM Transactions on Modeling and Computer Simulation*, pages 348–385, October 1991.

[5] R.L. Bagrodia, K.M. Chandy, and J. Misra. A message-based approach to discrete-event simulation. *IEEE Transactions on Software Engineering*, SE-13(6):654–665, June 1987.

[6] R.L. Bagrodia and Wen-Toh Liao. Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, pages 205–210, San Diego, California, January 1990.

[7] R.L. Bagrodia and Wen-Toh Liao. Transparent optimizations of overheads in optimistic simulations. In *Proceedings of the 1992 Winter Simulation Conference*, pages 637–645, Arlington, Virginia, December 1992.

[8] R.L. Bagrodia and Wen-Toh Liao. Maisie user manual. Technical report, Computer Science Department, UCLA, Los Angeles, CA 90024, June 1993.

[9] Randal E. Bryant. Simulation of packet communication architecture computer systems. Technical Report MIT-LCS-TR-188, MIT, 1977.

[10] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, pages 93–99, Tampa, Florida, March 1989.

[11] K.M. Chandy and R. Sherman. Space-Time and Simulation. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation Conference*, pages 53–57, Miami, March 1989.

[12] B.A. Cota and R.G. Sargent. Automatic lookahead computation for conservative distributed simulation. Technical Report CASE Center No. 8916, Simulation Research Group and CASE Center, Syracuse University, New York, December 1989.

[13] Ole-Johan Dahl. Discrete event simulation languages. In F. Genuys, editor, *Programming Languages*, pages 348–395. Academic Press, 1968.

[14] R. Fujimoto. Lookahead in parallel discrete event simulation. In *International Conference on Parallel Processing*, August 1988.

[15] R. Fujimoto, G. Gopalakrishnan, and J.J. Tsai. The roll back chip: Hardware support for distributed simulation using Time Warp. In *Proceedings of the 1988 SCS Simulation Multiconference on Distributed Simulation*, San Diego, California, February 1988.

[16] Richard Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[17] Anat Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, pages 61–67, San Diego, California, February 1988.

[18] Al Geist, Adam Beguelin, and Jack Dongarra et al. PVM 3.0 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, February 1993.

[19] D.H. Gill, F.X. Maginnis, S.R. Rainier, and T.P. Reagan. An interface for programming parallel simulations. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, pages 151–154, Tampa, Florida, March 1989.

[20] I. Greenberg, A.G. Mitrani and B. Lubachevsky. Unbounded parallel simulations via recurrence relations. In *1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1990.

[21] D. Jefferson. Virtual Time. *ACM TOPLAS*, 7(3):404–425, July 1985.

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

[23] Vikas Jha and R.L. Bagrodia. Transparent implementation of conservative algorithms in parallel simulation languages. In *Proceedings of the 1993 Winter Simulation Conference*, December 1993.

[24] Yi-Bing Lin and Edward D. Lazowska. Exploiting lookahead in parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):457–469, October 1990.

[25] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[26] D.M. Nicol. Parallel discrete event simulation of FCFS stochastic queueing networks. In *Parallel Programming: Experience with Applications, Languages and Systems*, pages 124–137. ACM SIGPLAN, July 1988.

[27] Kristen Nygaard and Ole-Johan Dahl. The development of the Simula language. In Richard Wexelblat, editor, *History of Programming Languages*, pages 439–493. Academic Press, 1981.

[28] Bruno R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, pages 139–144, Tampa, Florida, March 1989.

[29] Hassan Rajaei. SIMA: An environment for parallel discrete-event simulation. In *The 25th Annual Simulation Symposium*, pages 147–149, April 1992.

[30] Paul Reynolds. A spectrum of options for parallel simulation. In *Proceedings of the 1988 Winter Simulation Conference*, pages 325–332, San Diego, California, December 1988.

[31] Paul F. Reynolds. An efficient framework for parallel simulations. In V. Madisetti, D. Nicol, and R. Fujimoto, editors, *Proceedings of the 1991 SCS Multiconference on Advances in Parallel and Distributed Simulation*, volume 23:1, pages 167–174. SCS, January 1991.

[32] H. Schwetman. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, Washington, DC, December 1986.

[33] C.L. Seitz. The cosmic cube. *CACM*, 28(1):22–33, January 1985.

[34] Jeff Steinman. SPEEDES: Synchrnous parallel environment for emulation and discrete event simulation. In *Proceedings of the 1991 SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–103, Anaheim, California, January 1991.

[35] Joel West and Alasdar Mullarney. ModSim: A language for distributed simulation. In *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, pages 155–159, San Diego, California, February 1988.