


code with calls to a library that provide run-time support for optimistic parallel execution. The proposed system is able to translate sequential DES programs into parallel programs automatically.

The simulated performance data derived from our experiments show that for several applications, parallelism is available to offer significant speedups. However, program transformations may be required to exploit parallelism in certain circumstances. We observed that techniques developed for parallelizing compilers to unfold DO-loops can provide significant benefits in some situations.

The experiments also indicate that performance is sensitive to overheads, and that in some cases, speedup may decline as more processors are added. This suggests that new mechanisms to reduce the overhead of STM access and rollback are required. There are several possibilities. For example, a cache can be used to reduce the cost of STM accesses. Another possible approach is to assign the rollback overhead to the processor that is furthest ahead in simulated time, thereby allowing the slowest processor to advance more rapidly through the simulation. The extent that these mechanisms can alleviate the overhead problem is currently under investigation.

References


expensive compared to the event computation in this example. For instance, it takes the 
same amount of (assumed) execution time to log and un-log an STM access. It also takes 
the same amount of time to schedule and un-schedule an event. For an event that has 
majority of its computation in STM access and event scheduling, the cost to rollback is 
almost as expensive as its computation. The overall execution time for more processor will 
eventually exceed that of less processors when rollback overheads outweigh event computa-
tion. In addition, these experiments assume that the processor that caused the rollback is 
responsible for performing the necessary work in undoing incorrect computations. Because 
this processor is often “behind” in the simulation computation, better efficiency might be 
obtained by assigning this task to some other processor.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{speedup_vs_processor.png}
\caption{Queuing network: Speedup vs. Number of processors}
\end{figure}

8 Conclusion

In this paper, a system is proposed to adapt sequential discrete event simulation programs 
to a parallel platform based on optimistic execution. A standard framework including a set 
of simulation primitives is defined. We then show how these primitives can be parallelized 
with optimistic synchronization.

A prototype simulator was developed to implement the proposed system. The simu-
lator includes a translator that transforms SIMSCRIPT II.5 programs into equivalent C
light[i].color = GREEN;
/* reactivate the first waiting car in North/South bound lanes*/
.....
} else {
    light[i].color = RED;
    /* reactivate the first waiting car in East/West bound lanes*/
    .....;
}

![Graph showing potential speedup vs. number of processors](image)

**Figure 10:** Traffic Control (modified): Potential speedup vs. Number of processors

### 7.5 Introducing Overheads

So far, we have only presented the potential speedup that can be obtained when overhead is negligible. Additional experiments were conducted that include overhead of STM accesses and rollback. The speedup curves reveal that performance is very sensitive to overheads. For example, the observed speedups of the hypercube network benchmark is significantly degraded even if moderate overheads are assumed. In figure 11, we show speedup versus the number of processors, assuming that it takes one unit of simulated time to un-log each STM access and un-schedule each event, and one unit of time to access STM. Thus, the cost to roll back an event is proportional to the amount of computation to be undone. In these experiments, the speedup declines as the number of processors increases. This is because more rollbacks occur as more processors are used. Rollback overheads are relatively
Figure 9: Traffic Control: Potential speedup vs. Number of processors

```c
{
    int i;
    for(i = 0; i<max_number; i++) {
        if (light[i].color == RED) {
            light[i].color = GREEN;
            /* reactivate the first waiting car in North/South bound lanes*/
            ....
        } else {
            light[i].color = RED;
            /* reactivate the first waiting car in East/West bound lanes*/
            ....
        }
    }
}
```

Because there are no inter-loop dependencies, the `for` statement can immediately be converted to a `for-all` statement allowing several `light_control_events` to be executed concurrently.

```c
light_control_process(i)
int i;
{
    if (light[i].color == RED) {
```
states because each car must read their status (i.e., color).

Two experiments with different problem sizes were performed. The first has a street map consisting of $3 \times 3$ intersections, and the second a $9 \times 9$ intersections. As we can see (figure 9), the potential speedups in both cases are very poor. This is because the simulation program was written with a centralized light control process, that sequentializes the execution by changing the status of every traffic light. Therefore, the parallelism in the original application was only slightly more than 2—derived from the independence between the car generation process, and the light control process.

A restructuring of the traffic benchmark was performed to improve the parallelism. Rather than using a single process to control the lights at all intersections, a separate process could be used for each intersection. With such transformation, not every car process has to interact with the same light control process as before. Therefore, the parallelism in the program is proportional to the number of intersections (i.e., the number of light control processes). The potential speedup is shown in figure 10.

Although the above transformation was performed manually in our experiments, it could be automated. The optimization is similar to that commonly used in parallelizing compilers for concurrent execution of DO-loops. The original traffic light code was defined as followed:

```c
light_control_process()
```
7.3 Example 2: Shark’s World

This process-oriented simulation models the activities in an ocean containing two types of creatures: fish and sharks[4]. The ocean is divided into a two-dimensional matrix of sectors with toroidal wrap around connections at the edges. Sharks swim faster than fish, and can attack fish, but not each other. The benchmark was designed to capture behaviors found in combat simulation models.

In this simulator, each creature is modeled as a process, and the location and status (alive, or dead) of each creature as shared variables. A creature can swim from one sector to any adjacent sector. When a shark enters a sector, it will attack and kill all the fish in that sector before it leaves. The speed and direction are fixed for each creature. The speeds of sharks are uniformly distributed between 50 and 100 units per second, and those of fish between 10 and 20. The direction of each creature is uniformly distributed between 0 and 360 degrees.

The size of the ocean is 3200 by 1600 units, and is divided into a 8×4 matrix. The initial ratio of fish to sharks is 6 to 1. The number of resident creatures varies in each run, ranging from 56 to 224 creatures. When the parallelized sharks world program was run on the simulator, the potential speedup for 8 processors is as high as 7.45. However, for 32 processors, the potential speedup is at most 16.93 with 224 creatures (see figure 8). This is because interactions between creatures will limit the amount of concurrent execution. For example, when more processors are used, more creature processes can be executed concurrently. Since the sharks in the same sector may attack the same fish, therefore, more interactions are observed.

7.4 Example 3: Traffic Control Simulation

The traffic control simulation consists of a number of intersections which are controlled by traffic lights. To reach its destination, each car has to drive from its starting point through a sequence of intersections. When arriving at an intersection, a car will be enqueued in the lane if other cars are present. It is dequeued when it is its turn to cross the intersection. Three types of processes are used, a car process, which models the itinerary of a car, a light control process, which changes the status of all lights, and a generation process, which continuously generates a car process. When a car is enqueued, it suspends itself until it is reactivated by other processes. In this benchmark, traffic lights are modeled as shared
executed that are committed (the efficiency) decreases from 95.6% to 64.5% as the number of processors increases from 16 to 64 when the message population is 256. The decrease in the committed event rate is more pronounced when the population is small.

Figure 6: Queuing network: Potential speedup vs. Number of processors

Figure 7: Queuing network: Efficiency vs. Number of processors
event list. The zero overhead experiments are used to determine the amount of parallelism that is available in the application.

Three SIMSCRIPT II.5 benchmarks were used in our initial experiments. The first example is an event-oriented queuing network simulation. The second is a process-oriented sharks world[4] simulation, and the third is a simplified traffic control simulation. The performance metric used is the potential speedup, defined as the ratio of execution times between $n$ and a single processor. The execution time of a single processor is defined as the time spent to execute the application code. It does not include the event list overhead. In the following, we present the results assuming zero overhead, then measurements with overheads included.

### 7.2 Example 1: 64 node hypercube queuing network

The first example is a simulation of a closed queuing network configured as a 64-node hypercube. This benchmark bears some resemblance to simulations of communication networks, and has been a widely used benchmark in PDES research. In this example, messages may be sent from one node to any adjacent node. This benchmark is programmed with only events. Two events are used to model each hop. One is for the departure, and the other for the arrival at the adjacent node. Messages at a node are processed in first-come-first-serve order. An arrival event schedules a departure event with timestamp increment corresponding to the waiting time in the queue plus a service time. The service time of each arrival is selected from a stream of random numbers with an exponential distribution with a mean of 5 time units. The simulated system is a closed network with zero-delay transmission, so a departure event schedules an arrival event immediately. There is a server at each node. Each node uses two variables: the local clock, and the number of messages at the node. Both are mapped to STM data objects in the translated program.

Several runs are performed with message populations ranging from 64 to 256. When the parallelized version runs on the simulator, the measured potential speedup is significant. With 16 processors, the average potential speedup is about 15. It is because this benchmark contains much parallelism. The ratio of potential speedup and the number of processors decreases slightly as the number of processors increases. It is because there is more interference between events at the same node, so, more rollbacks occurred. However, even with 64 processors, the potential speedup could still be as high as 46.55 when the message population is 256 (see figure 6). As shown in figure 7, the percentage of all events
SCHEDULE AN ARRIVAL IN 10 MINUTES
...
ACTIVATE AN AIRCRAFT NOW

translated to:

    event_schedule(arrival, NULL, current_time+10);
    ...
    ProcActivate(aircraft, NULL, current_time);

Figure 5: Process/Event interactions and their translation.

7 Preliminary Results and Discussion

A simulator of the parallel execution has been developed for verifying the approach and evaluating performance of the proposed parallel system. The current implementation of the simulator includes features of STM and interactions between processes.

7.1 The Simulator

The simulator uses a light-weight threads package called CTHREAD[11]. A scheduler is used to dispatch scheduled events to processors. Scheduling is based on the smallest timestamp policy, i.e., the globally smallest timestamped event would execute first. Once an event is assigned to a processor, it does not release the processor unless it finishes its execution or is rolled back. Rollbacks are performed immediately once a data dependency violation is detected. Currently, it is the responsibility of the processor executing the event that caused the rollback to undo the errors caused by the rolled back events and to return these events to the scheduling queue. An event attempting to read an STM version which is under writing by another event must wait until the write operation is completed.

The execution time of a translated program is measured by counting the number of statements that are executed. We assume that each statement of the translated C program takes one unit of simulated time to execute. The generation of the embedded code to count the number of executed statements is automated. We present results where the overhead of rollbacks and STM accesses is negligible, as well as experiments where it is not. These overheads include traversal of STM to locate a proper version of an object, logging access information, clearing premature accesses and execution (i.e., rollbacks), and maintaining the
the shared variable number_of_customer, and another for time_of_arrival. Each entry points to an object of OBJ_SIZE bytes allocated by STM. Any access to a shared variable must uses the functions defined by STM and typecast to the appropriate type (see figure 4). Similarly, event list operations and process interactions are translated into function calls to the runtime library (see figure 5).

DEFINE NUMBER_OF_CUSTOMER AS AN INTEGER NUMBER
DEFINE TIME_OF_ARRIVAL AS A REAL NUMBER

translated to:

MapTable *number_of_customer;
MapTable *time_of_arrival;
...
number_of_customer = (MapTable*)malloc(sizeof(MapTable));
number_of_customer->object = MakeObj(OBJ_SIZE);
time_of_arrival = (MapTable*)malloc(sizeof(MapTable));
time_of_arrival->object = MakeObj(OBJ_SIZE);

Figure 3: Global variables declarations and their translation.

ADD 1 to NUMBER_OF_CUSTOMER
LET TIME_OF_ARRIVAL = 100.0

translated to:

int *ptr_number_of_customer;
float * ptr_time_of_arrival;
...
ptr_number_of_customer = (int*)WriteVar(number_of_customer);
*ptr_number_of_customer = *ptr_number_of_customer + 1;
ptr_time_of_arrival = (float*)WriteVar(time_of_arrival);
*ptr_time_of_arrival = 100.0;

Figure 4: Global variables accesses and their translation.

The translated program (with embedded library calls) is then compiled and linked with the runtime library. This library provides a function for each SIMSCRIPT II.5 simulation constructs discussed in previous section.
The parallelization system defines a function called **ProcInterrupt** that uses the retraction primitive to retract the event for the interrupted process, then records the remaining waiting/working time.

**RESUME**, for resuming an interrupted process. For example, the statement **RESUME customer**, puts the **customer** notice back into the event list. The process will resume executing when the notice is processed.

The parallelization system defines a function called **ProcResume** that uses the scheduling primitive to schedule an event for the resumed process. The scheduled event has the timestamp of the current time plus the saved remaining waiting/working time when it was interrupted.

**SUSPEND**, for a process to suspend itself. The only format for this construct is simply **SUSPEND**. This statement signals the end of the current event. As in the **INTERRUPT** statement, it keeps the notice intact. A parallel implementation requires the current event release the processor to another event.

**REACTIVATE**, for reactivating a process after it has been suspended. For example, the statement **REACTIVATE THIS customer NOW**, puts the **customer** process back into the event list. The parallel implementation of this construct is similar to **ACTIVATE**.

### 6 The Parallelization System

The parallelization system consists of two parts. The first is a translator that takes a SIMSCRIPT II.5 program and translates into an equivalent C program with embedded calls to Space Time Memory for accesses to global variable. The second is a library of runtime support for optimistic parallel execution described earlier.

A data structure called the **maptable** is defined to provide a mechanism of mapping simulation state variables into STM data objects. In the translated program, a maptable entry is defined for each state variable, and any access to it must go through the maptable. In other word, the maptable provides a handler for simulation programs to access state variables stored as STM objects. Each entry of the maptable has two fields: a pointer to a data object in STM, and an address offset (in bytes) of that state variable within the data object. Figure 3 shows a segment of SIMSCRIPT II.5 declarations and its translated equivalent. In this example, a maptable entry is allocated (by **malloc**, the memory allocation in C) for
variable `takeoff` is marked as destroyed in Space Time Memory (see `DESTROY` below).

`ACTIVATE`, for activating a process. For example, the statement, `ACTIVATE A customer IN 10 MINUTES`, will start a process called `customer`. The starting time for this process is 10 minutes from the current simulated time.

To implement this operation, a function called `ProcActivate` is defined. This function creates a process notice (see `CREATE` below), and schedules an event for the process (see `SCHEDULE`).

`CREATE`, for creating an event or a process. On executing this statement, a notice is created (i.e., memory is allocated), but not inserted into the event list. For example, the statement, `CREATE A takeoff`, allocates memory for a notice representing a takeoff event/process. The parallel implementation of this construct allocates memory from STM (using `MakeObj`) for the event/process notice.

`DESTROY`, for destroying an event or a process and reclaiming the memory used for its representation. It is assumed that the event/process is not currently in the event list. Because the destroy may be rolled back, this operation is implemented by marking the notice as destroyed (using `EraseObj`), rather than reclaiming the memory immediately. The memory is freed only when the operation is committed.

`WORK/WAIT`, for a process to denote the passage of simulated time. For example, the `WAIT 10 MINUTES` statement puts the process notice back into the event list, and will be executed 10 minutes from the current simulated time. This statement signals the end of the current event.

To implement this operation, a function `ProcWork/ProcWait` is defined in the parallelization system. This function uses the scheduling primitive to schedule an event for itself, then halts its current execution, and releases the processor to another event.

`INTERRUPT`, for interrupting a process. For example, the statement, `INTERRUPT customer`, temporarily removes the process notice denoting `customer` from the event list. The removed process notice must be kept intact for resumption. A process can be interrupted only if it is executing a `WORK/WAIT` statement. When interrupted, the amount of remaining waiting/working time is saved.
event list are sorted by their simulated time. One notice is removed from the global list at a time, and the corresponding event/process routine is executed.

5.1 SIMSCRIPT II.5 Constructs And Their Parallelization

SIMSCRIPT II.5 provides a number of constructs to model the interaction between events/processes, and each of them can be mapped to the primitives defined in the framework. A SIMSCRIPT II.5 process is built on top of events, i.e., a process may be viewed as a sequence of events. The difference between an event and a process is when an event finishes its execution, its event notice is discarded. When a process finishes its current event, its notice is saved. When the process is rescheduled, the saved notice is inserted into the event list. Thus, SIMSCRIPT II.5 applications may be considered as event oriented.

Specific SIMSCRIPT II.5 constructs and their implementation in the parallelization system are described as below. Each construct starts with the description of its usage and operation in the sequential execution, followed by a brief discussion of how it is implemented in the parallelization system.

**SCHEDULE**, for scheduling an event. For example, the statement, `SCHEDULE A takeoff NOW`, schedules an event called `takeoff` at the current simulated time. An event notice is inserted into the event list. A global variable `takeoff` is used as an event handler.

There is a critical issue that must be resolved for the parallel implementation of this construct. That is, SIMSCRIPT II.5 allows one to schedule events with zero timestamp increment, while Time Warp does not. A straightforward solution is to append a tag to a timestamp to make it unique. Thus, the timestamp in the parallelization system consists of the simulated time specified by SIMSCRIPT II.5 and a tag. The tags of the same timestamped events must be assigned values that would preserve their sequential execution order. This can be determined from the timestamps of the events that schedules them. This construct can be directly mapped to the event scheduling primitive. The global variable `takeoff` is stored in Space Time Memory.

**CANCEL**, for canceling an event. For example, the statement, `CANCEL takeoff`, cancels the `takeoff` event. The event notice representing the `takeoff` event (pointed by the event handler `takeoff`) is removed from the event list, and discarded.

It can be directly mapped to the event retraction primitive. As described in section 4.2, the event is marked as cancelled, but still remains in the event list. Similarly, the global
To allow rollback, the retraction operation (i.e., DE primitive) must not discard the retracted event from the event list. Instead, it leaves the event in the event list and marks it as retracted. Rolling back a retraction operation need only remove the mark of the previously retracted event[10]. To undo the effect of processing an event, each event carries the following information concerning the operations it performed: (1) events it scheduled, (2) events it retracted, (3) variables it created, (4) versions it created, (5) versions it read, and (6) variables it disposed. Discarding a version also implies that all of the events that read or wrote any newer version of the state variable must be rolled back.

In many simulation languages, an event may examine the event list (i.e., EE primitive). Because the operations may depend on the outcome of this examination, it is necessary to detect changes in the outcome caused by changes in the event list. One approach is to store a representation of the event list in the Space Time Memory. When an event examines the event list, the information is recorded. Subsequent changes to the event list at earlier points in simulated time will necessitate a computation to determine if a rollback is necessary. This approach is currently under investigation, and will not be discussed further in this paper.

5 Case Study: SIMSCRIPT II.5

We choose SIMSCRIPT II.5 as the language for this case study for two reasons. First, SIMSCRIPT II.5 is a widely used language, so automatic parallelization of SIMSCRIPT II.5 programs has some practical significance. Secondly, SIMSCRIPT II.5 is a relatively old programming language developed in the 1960s, and was not designed with any consideration for parallel processing. This makes it a challenging language for automatic parallelization of existing sequential DES languages. Details of the language are described in [1, 15].

The structure of a SIMSCRIPT II.5 program consists of a declarative part and an executable part. The declarative part, called the PREAMBLE, defines all of the global variables, modeling elements such as events and processes, and metrics of the system to be measured. The executable part contains a mandatory MAIN routine, a routine for each type of event and process declared in PREAMBLE, and miscellaneous supporting routines for function and procedure calls.

SIMSCRIPT II.5 was originally designed to use a sequential execution mechanism. When an event/process is scheduled, a data structure (called event/process notice) representing the event/process is inserted into a global event list. The notices in the global
WriteObj(var_id, value) modifies a data object. It creates a new version of a data object
(var_id) whose new contents will be value. Events that have accessed versions of this
object at higher points in simulated time will be rolled back and re-executed. It is an
implementation of the MS primitive.

ReadObj(var_id): version reads the contents of a data object. It searches through the
versions of a data object(var_id), and returns a pointer to the most recent (in simu-
lated time) version. It is an implementation of the RS primitive.

EraseObj(var_id) reclaim the memory allocated to a data object. It deallocates the
memory which is allocated to a data object(var_id) by the operation MakeObj.
To allow rollback, the implementation of this function must ensure that the actually
deallocation of memory is not done until the deallocating event is committed. That is,
when a data object is to be erased, it is indicated with a mark. If the erase operation
is rolled back, the mark is cleared off, hence un-erased. It is an implementation of the
DS primitive.

Rolling back an event entails undoing any actions it might have done, e.g., writes to
other objects and may result in a cascade of rollbacks, as in Time Warp. Details of this
mechanism are described elsewhere[5].

During a simulation, STM maintains the following information for each data object:
(1) the simulated time at which it was created, (2) the simulated time at which the object
was deleted, and (3) a list of versions. Each version also records: (1) the creation time,
i.e., the timestamp of the event that executed a write to create this version, (2) the event
that created the version, and (3) a list of events that have read this version. With this
information, STM is able to determine the set of events that need to be rolled back on a
write operation.

4.2 Event Set Operations

Optimistic parallel execution uses rollback to correct erroneous computations. Thus the
system must be able to undo the operations on the event list. The event list can be
either centralized or distributed. Rolling back the schedule operation (i.e., SE primitive)
is accomplished by using an anti-message like mechanism (similar to Time Warp[7]) to
annihilate the previous scheduled event(s) (see [5] for details).
reads and writes to the variable with \( T \) as the temporal coordinate. The state variables used by the simulator are mapped to a set of data objects, so the spatial coordinate is an object number. In general, each data object includes many state variables. A row of memory locations in STM can be viewed as a record of the history of object values across simulated time. When an event modifies a shared variable for the first time, STM creates a new copy, called a version, for that variable, rather than overwrite the previous contents of the variable. Any subsequent modifications to the variable by this event are made by overwriting the newly created version, i.e., new versions are not created. By saving multiple versions of state variables, STM can provide multiple views of system state corresponding to different points in simulated time. A snapshot of the system state at any point of simulated time will consist of the value of each state variable at that simulated time.

![Diagram of memory address: conventional memory vs. Space Time Memory](image)

**Figure 2:** Memory address: conventional memory vs. Space Time Memory

STM provides the following operations which correspond to the four primitives (i.e., **NS, RS, MS, and DS**) described in the framework earlier:

**MakeObj(size):** var.id allocates memory for data objects(s). It allocates a new block of memory containing size bytes, and returns a handle (var.id) for referencing this data objects. It is an implementation of the **NS** primitive.
4 Concurrent Execution of Simulation Primitives

Here, we assume the parallelized program executes on a shared memory multiprocessor. The execution mechanism is a simple generalization of the event list mechanism used by sequential simulators. Specifically, each processor repeatedly removes the smallest timestamped event from the event list, and then executes simulation code to process that event. The simulation code is functionally similar to that of a sequential simulator. The runtime system ensures that the parallel execution achieves the same results as the sequential execution.

Two critical issues must be addressed before the above approach to parallelization can provide the same results as the sequential execution. First, two concurrently executing events, each containing a different timestamp, may access a common state variable. Consider two events, one with timestamp 10 and the other 20, that both access a state variable $X$. Should $X$ hold the value that existed at simulated time 10, or the value at time 20? To address this question, we use an abstraction called Space Time Memory (STM)\cite{2, 5, 18} and a computation model based on optimistic computation, i.e., rollbacks are used to synchronize computation.

The second problem concerns the EE primitive that access the event list. At any point in time, the event list in the parallel execution will be substantially different from that of the sequential execution. This is a more difficult problem than the first. We suggest a possible solution for this problem, but the central focus of this paper is on using space-time memory to support shared state variables.

4.1 Space Time Memory

The optimistic execution mechanism processes events containing different timestamps concurrently. Since events with different timestamps may have different views of the system state, it is imperative for each event to get the view that existed at its timestamp. Because our computation model uses a shared memory computing platform, STM is designed to address the shared state problem by maintaining multiple versions of each state variable.

Conventional memory can be viewed as a one-dimensional array of values, which can be addressed by specifying a spatial address. STM, on the other hand, is organized as a two-dimensional array (see figure 2), in which each memory location is addressed by a row (spatial) as well as a column (temporal) coordinate. An event with timestamp T will perform
of a variable (i.e., \( v_0 = RS(S_{v/v_0}, v) \), \( S_{v/v_0} \) means that \( S \) includes a variable \( v \) whose value is \( v_0 \)), (3) modify the contents of a variable (i.e., \( S'_{v/v_0} = MS(S_{v/v_0}, v) \)), and (4) delete a variable (i.e., \( S' = DS(S, v) = S - \{v\} \)). Similarly, three operations may be performed on the event set: (1) add a new event (i.e., \( E' = SE(E, e) = E \cup \{e\} \) where \( e \) is an event to be scheduled, represented by a tuple containing the function to be executed, its arguments, and the simulated time at which event is to be executed), (2) delete or unschedule a previously scheduled event (i.e., \( E' = DE(E, e) = E - \{e\} \) where \( e \) is an event to be unscheduled), and (3) examine the contents of the event set and return some value based on the event set, for example, the number of the pending events (i.e., \( V = EE(E) \) where \( V \) could be any value, e.g., boolean, integer, or an event set). The execution of a sequential DES program can be viewed as a sequence of transitions from one \( S, E \) pair to another.

These seven primitives (i.e., \( \text{NS}, \text{RS}, \text{MS}, \text{DS}, \text{SE}, \text{DE} \) and \( \text{EE} \)) appear to be sufficient to model a variety of existing simulators written in simulation languages, such as GASP, GPSS, and SIMSCRIPT II.5. Since higher level constructs (e.g., processes) can be realized as abstractions that are built on top of these primitives, these primitives are able to encompass applications with different paradigms of modeling, or \textit{world views}[8, 9, 14, 16, 20].
other hand, optimistic approaches allow dependence violations to occur, and use a rollback mechanism to recover. Because optimistic methods can exploit parallelism in situations where dependence violations might occur, but actually do not, they offer the advantage of exploiting more parallelism than conservative approaches. Currently, simulation programs using these approaches must be written explicitly for parallel execution.

Other approaches have been proposed for concurrent execution of sequential simulation programs. The replicated trials approach executes independent copies of the simulation on separate processors. This is a suitable approach when collecting data over a wide range of parameter settings, or when long simulation runs are needed to reduce the variance of output statistics. The memory requirements for this technique are large, however, because \(N\) times the memory of a sequential execution are required when using \(N\) processors.

Like the replicated trials approach, the functional specialization method does not require modifications to the sequential simulation program. Dedicated processors are used to execute frequently used functions such as random number generation and event list manipulation. However, this approach only offers a limited amount of speedup[3, 12].

Much research has been completed concerning compilation techniques to identify data dependence relationships between operations, typically iterations of DO-loops, in order to parallelize arbitrary sequential programs[13, 17, 19]. The work in parallelizing compilers is applicable to PDES as it can provide some useful dependence information, however, the general problem of identifying all data dependence constraints in DES is very difficult. This is because many of the data dependence relationships in DES cannot be determined at compile time. For instance, many dependencies depend on timestamp values, which are not known until after execution begins.

3 A Framework for DES Programs

In our parallelization system, a common framework is first defined to which sequential discrete event simulation programs are mapped (see figure 1). In our framework a DES program maintains two sets – a set of state variables \(S\), and a set of unprocessed events, \(E\). The former is used to describe the system state of the model, and the latter to specify state changes that will occur in the simulation future. During the execution of an event, one or more of the following four operations may be performed on the state variable set: (1) create a new variable (i.e., \(S' = NS(S, v) = S \cup \{v\}\), where \(v\) is a variable.), (2) read the contents
1 Introduction

Discrete event simulation (DES) has long been a widely used technique for analyzing the behavior of complex dynamical systems such as communication networks, computer systems, and combat scenarios, to name a few. These applications often require excessively long execution times on sequential machines to obtain useful results. As multiprocessor computers become more widespread, parallel execution of discrete event simulation programs (PDES) offers great potential for solving this problem.

However, developing parallel simulation software is not only time-consuming but also requires much more training and experience than developing sequential software. Further, in many cases, it is impractical, if not impossible, for users to abandon large sequential simulation programs that they have been developing and using. It is clear that there would be enormous benefit if these sequential programs could be readily translated into parallel programs.

In this paper, we will present a method to automatically translate a sequential DES program into an equivalent parallel version. The remainder of the paper is organized as follows: in section 2, we will review related work. A standard framework is described in section 3 that defines a common set of simulation primitives that are used by many existing sequential simulation languages. In section 4, we describe how these primitives can be parallelized. In section 5, we describe a case study examination of SIMSCRIPT II.5, a widely used sequential simulation language, and show how it is mapped to the framework. A prototype implementation of a parallelizing SIMSCRIPT II.5 compiler is described in section 6. Section 7 presents preliminary performance data from three SIMSCRIPT II.5 benchmark programs, and section 8 presents conclusions.

2 Related Work

Much of the work in PDES that has been reported thus far is concerned with synchronizing concurrently executing event computations. A number of synchronization protocols have been proposed (see [6] for a survey of these techniques.) These protocols may be categorized as conservative or optimistic. Conservative approaches strictly avoid any possibility of violating dependence relationships arising from concurrent execution of events. These approaches use a mechanism to determine whether it is safe to process an event. On the
Automatic Parallelization of
Discrete Event Simulation Programs

Jya-Jang Tsai
Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
GIT-CC-93/19
June 15, 1993

Abstract

Contemporary simulation applications usually need to process many millions of
events to obtain sufficiently accurate performance predictions for large, complex sys-
tems. While parallel computation offers a solution to this problem, developing parallel
simulation code is very time-consuming and currently requires a high level of expertise.
In addition, traditional approaches to automatic parallelization, as used in many
parallelizing compilers, are not well-suited for discrete event simulations because these
computations are very irregular and exhibit highly data dependent behavior.

In this paper, we present a framework to which many existing sequential simulation
languages can be mapped. Using a Time Warp like parallel execution mechanism, we
show that it is possible to automatically translate DES applications written in these
sequential simulation languages into equivalent parallel programs. A case study of the
SIMSCRIPT II.5 language is described to illustrate the proposed parallelization method.
Based on this approach, a parallelizing compiler for the SIMSCRIPT II.5 language was
developed. Preliminary performance data using this compiler is presented based on
simulations of parallelized discrete event simulation programs.

1This work was supported by NSF grant CCR-8902362.