August 1995

UILU-ENG-95-2230
CRHC-95-19

Center for Reliable and High Performance Computing

Acceleration Techniques for Dependability Simulation

James David Barnette

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-95-2230 (CRHC-95-19)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research, Comp. Sci. Corp., National Aeronautics and Space Administration
6c. ADDRESS (City, State, and ZIP Code) 1308 W. Main St. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Arlington, VA San Diego, CA Hampton, VA
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 7a 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Acceleration Techniques for Dependability Simulation

12. PERSONAL AUTHOR(S)
James David Barnette

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	August 1995	60

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	simulation, simulation model, discrete-event simulation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

As computer systems increase in complexity, the need to project system performance from the earliest design and development stages increases. We have to employ simulation for detailed dependability studies of large systems. However, as the complexity of the simulation model increases, the time required to obtain statistically significant results also increases. This paper discusses an approach that is application independent and can be readily applied to any process-based simulation model. Topics include background on classical discrete event simulation and techniques for random variate generation and statistics gathering to support simulation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

ACCELERATION TECHNIQUES FOR DEPENDABILITY SIMULATION

BY

JAMES DAVID BARNETTE

B.S., Oklahoma State University, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to express my appreciation to my thesis advisor, Professor Ravi K. Iyer, for all of his assistance and guidance throughout this thesis work. I would also like to thank my friends at the Center for Reliable and High-Performance Computing for their help. In particular, I would like to recognize Kumar Goswami, who developed the original version of DEPEND and spent many hours discussing its future direction, and Darren Sawyer, who provided many valuable real world suggestions to improve the tool. Additionally, I would like to thank my friends at InterVarsity Christian Fellowship for much needed support and encouragement during these past two years. Finally, I would like to thank my family for their encouragement throughout my college years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

As computer systems increase in complexity, the need to project system performance from the earliest design and development stages increases. We have to employ simulation for detailed dependability studies of large systems, as analytical models do not provide an understanding of the component interactions. The component models can incorporate such details as the actual communication protocols, operating system algorithms, and on-line diagnostic and maintenance procedures used in the simulated system.

Unfortunately, as the complexity of the simulation model increases, the time required to obtain statistically significant results also increases. This is a particularly difficult problem for dependability analysis, because faults are rare events. When the system is being modeled in detail, the issue may be less with the rarity of the fault, but more with the overall time needed to model the system behavior occurring before, during, and after each fault.

Fortunately, several approaches exist for reducing this simulation time explosion. These include distributed simulation [1, 2, 3], importance sampling [4, 5] and hybrid/hierarchical simulation [6] among others. All of these approaches may be categorized as somewhat "application dependent." For example, many importance sampling techniques require that fault arrivals be exponentially distributed. Such techniques cannot be applied to a system where the effects of latent faults and on-line diagnostics are modeled. Even distributed simulation requires intelligent model partitioning and scheduling to achieve reasonable speedup [7, 8].

Our approach, on the other hand, is "application independent" and can be readily applied to any process-based simulation model. We use compiler-based techniques to translate, optimize, and parallelize a process-based model into a hybrid process-based/event-driven model. This hybrid model performs much better because we avoid context-switching and certain scheduling overheads that are inherent in process-based run-time systems. Through the use of these techniques, we have obtained up to a 60 times speedup on some models.

This acceleration approach grew out of our need to develop an extremely fast, yet practically useful simulation tool for system dependability analysis. Here the issues were twofold, to provide an environment that facilitates modeling (e.g., object-oriented paradigm, process-based specification) and yet provide the speed of simpler simulation tools. The first generation of this tool was named DEPEND and has proven quite useful in computer system dependability studies [9, 10, 11, 28].

Chapter 2 provides background on classical discrete event simulation including the three classical simulation world views: event-scheduling, activity-scanning, and process-interaction. Chapter 3 describes techniques for random variate generation and statistics gathering to support simulation. Then, Chapters 4 and 5 motivate the need for and describe the key technique presented in this thesis. Chapter 6 provides a more detailed description of the simulation environment and the compiler-based techniques that we employed. Chapter 7 provides a preliminary evaluation of this approach with a simple case study. Finally, Chapter 8 contains the conclusion and discusses possible future work.

2. CLASSICAL DISCRETE EVENT SIMULATION

In this chapter, we describe discrete event simulation including the three classical simulation strategies or world views. Discrete event simulation concerns the modeling on a computer of a system in which state changes can be represented by a collection of discrete events. These events can occur at regular or varying intervals of time. If the system can be adequately described as having events all of which occur at constant intervals, then a simpler approach may be used which avoids the overhead associated with the maintenance of an event list. Here, we concern ourselves with systems which can be described by events occurring at irregular or varying time intervals. Zeigler offers a theoretical formalism for discrete event simulation in [12, 13].

In a discrete event system, change takes place as each event occurs. No state changes occur to entities during the time between event occurrences. Thus, there is no need to simulate this time in our models; it can be skipped over. As a result, all modern computer simulation environments use the *event driven* approach to time advancement. After each event has executed (changing the state of the system), time is advanced to the time of the next event, where required state changes are made again. In this way, a simulation is able to skip over the inactive time whose passage must be endured in the real world. Thus, the event list becomes the central element of any discrete event simulation environment.

In modeling a system for computer simulation, there are two kinds of intercomponent relationships: *mathematical* and *logical*. Mathematical relationships exist between variables

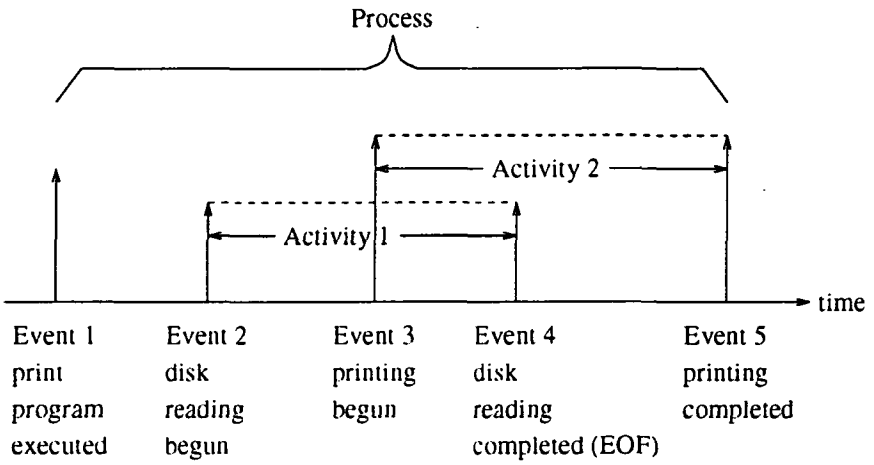

Figure 2.1: Relationship between the simulation strategies

associated with system components. For example, if D is the number of waiting requests for a disk drive in a computer system, then D is incremented when a new request arrives and decremented when the requested read/write operation is completed. Logical relationships, on the other hand, describe a condition that must hold before a particular event occurs. Consider the disk example again. When a disk request is completed, the disk becomes idle, if no requests are still pending; otherwise, the disk remains busy and begins serving the next request. The expression of mathematical and logical intercomponent relationships differs between the three classical simulation strategies, but before we describe the strategies, we need to first define some basic concepts.

The concepts of *event*, *activity*, and *process* are important when building a model of a system. As already defined, an *event* signifies a change in state of an entity. An *activity* is a collection of operations that transform the state of an entity. And, a *process* is a sequence of time-ordered events. To illustrate the relationships among these concepts, consider a computer system with a disk and a printer. A program to print a file is executed which repeatedly reads from the disk and writes to the printer until the file has been completely printed on the printer. Figure 2.1 shows the relationships between these three concepts [14, page 24].

These three concepts lead to the three classical simulation strategies or world views. The *event-scheduling* strategy emphasizes a detailed description of the steps that occur when each

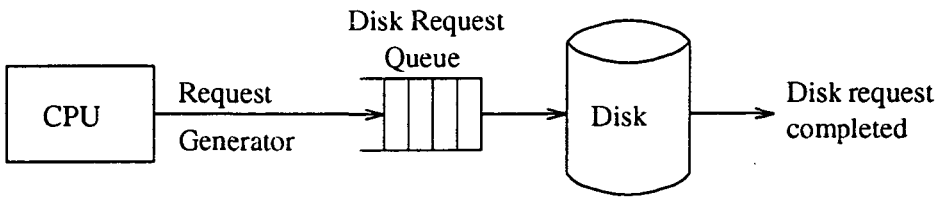

Figure 2.2: CPU/Disk system model

event occurs. In this strategy, both mathematical and logical relationships are explicitly specified. In the case of logical relationships, this results in checking the condition at every point where the dependent event may be triggered.

The *activity-scanning* strategy emphasizes the review of activities to be initiated or terminated each time an event occurs. Only those activities with logical relationships to other system components need be reviewed. In this strategy, mathematical relationships are explicitly specified as in the event-scheduling strategy.

The *process-interaction* strategy emphasizes the progress of an entity from its arrival event through its departure event. Here, both mathematical and logical relationships are handled implicitly by the simulation environment. For example, first-come/first-serve queues automatically block the invoking process until it reaches the head of the queue.

More recently Evans developed the engagement strategy as a combination of the process-interaction and activity-scanning strategies [15].

We will use a simple model of a CPU and disk subsystem to demonstrate each of the simulation strategies (see Figure 2.2). Here the CPU is modeled by a random disk access (read or write) request generator, and the disk is modeled as having a certain average seek time, rotational latency, and transfer rate. The events using the event-scheduling strategy for such a model might be the generation of a new disk request, the initiation of service of a disk request, and the completion of service for a disk request.

In order to generate the disk requests and the components of the disk access time, various random variate generators have to be employed. The random variates include uniform, exponential, and normal distributions among others. In a deterministic computer system, it is not possible to generate a truly random stream of numbers, rather, the computer uses various

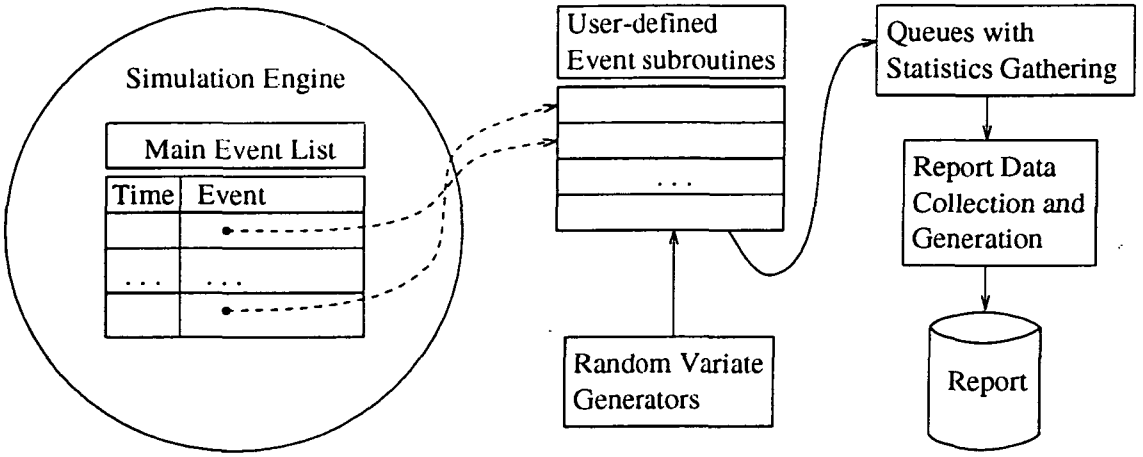

Figure 2.3: Event-scheduling simulation run-time environment

numerical techniques to generate a sequence of numbers that have good statistically random properties. These streams inevitably depend on an initial "seed" value which determines the sequence of numbers to be generated. Generally nonuniform random variates are generated by drawing one or more uniform pseudorandom numbers. These techniques will be discussed in Chapter 3.

We will now illustrate each of the three world views using a simple CPU/disk model as a demonstration vehicle.

2.1 Event-Scheduling Strategy

The event-scheduling strategy represents a straightforward implementation of event-driven simulation. As in all of the simulation strategies, the event list is at the heart of the simulation providing a time ordering of events as the simulation progresses. A simplified diagram of the simulation run-time environment for the event-scheduling strategy is given in Figure 2.3.

A model such as this CPU/disk system could be used to determine the distribution of service times (time from request to completion) or the distribution of waiting times in the queue. Various request generation patterns could be tried to determine their effect on the above. Or, the disk parameters could be modified to determine the sensitivity of the model to small changes in disk system performance. The pseudocode for this model appears in Figure 2.4.

GenerateNewRequest:

1. Draw number of sectors to access from an exponential distribution rounding to the nearest integer
2. Enqueue the request onto the disk request queue
3. If the disk is not busy, then schedule a BeginService event to execute at the present simulation time
4. Draw an interarrival time from an exponential distribution
5. Schedule a GenerateNewRequest event for the current time + interarrival time

BeginService:

1. Take next request from the head of the disk request queue
2. Compute service time as the sum of the seek, rotational latency, and transfer times
3. Schedule a CompleteService event for the current time + service time

CompleteService:

1. Mark the service as completed and log waiting time and service time
2. If the queue is not empty, schedule a BeginService event to execute at the present simulation time

Figure 2.4: Event-scheduling pseudocode for the CPU/disk model

After this model has been executing for awhile and just after a BeginService event was executed, the event list could contain the following.

Event list	
Time	Event
43	GenerateNewRequest
47	CompleteService

The event list enforces causality by executing events in time order. In this case, the simulation would proceed by setting the current time to 43 and executing the GenerateNewRequest event which would add a new request to the disk request queue and schedule a new GenerateNewRequest event at some future time > 43. If we assume 50, then the event list after the execution of the event at 43 is as follows:

Event list	
Time	Event
47	CompleteService
50	GenerateNewRequest

The simulation would proceed by setting the current time to 47 and executing the CompleteService event. Since the disk request queue was not empty, it would immediately schedule a BeginService event at time 47. Thus, the event list would appear as follows after the execution of this CompleteService event:

Event list	
Time	Event
47	BeginService
50	GenerateNewRequest

Simulation would proceed by setting the current simulation time to 47 and executing the BeginService event. This event would schedule a CompleteService event for some future time (could be before or after 50). If the event is scheduled to complete after time 50, then the sequence of events is the same as it was at time 43 and the simulation will proceed in a similar pattern until completion.

As can be seen in this example, the event list and event dispatcher are central to the simulation. As models increase in complexity, the number of events simultaneously scheduled

can increase dramatically, although some complex models are structured in such a way that only a few events are scheduled on the event list at a given time. For these models, a simple linked-list is adequate to maintain time ordering; however, where many events are on the event list at a time, a more complex data structure such as a heap or some indexing scheme should be employed. A good simulation environment should permit the simulation user to select the appropriate data structure. Of course, the event dispatcher must also be efficient since it too is in the critical path between each event.

2.2 Activity-Scanning Strategy

Conceptually, the activity-scanning strategy classifies events according to two categories, B- and C-activities. B-activities are those which are bound to occur at some future point in time; they correspond to mathematical relationships in the system being modeled. As such, these are placed on the event list as in the event-scheduling approach. C-activities are those which occur as a consequence of another activity and correspond to logical relationships in the system being modeled. Unlike the event-scheduling approach, in the activity-scanning strategy, these events are placed on a special conditional event list which is scanned after each B-activity is executed. A given C-activity will only be executed when its associated condition evaluates to true (e.g., if condition, then execute the C-activity).

The event scheduler maintains a list of B-activities and the time at which they occur. Time advances to the first scheduled B-activity, which is executed by invoking the associated event-handling subroutine. This subroutine may change state such that one or more C-activities are activated. Upon return, the simulator scans all C-activities and executes those that were activated. When all activated C-activities have been executed, the system then advances time to the next B-activity and repeats the above procedure. Simulation proceeds until there are no more B-activities to execute. A simplified diagram of the activity-scanning run-time environment is given in Figure 2.5.

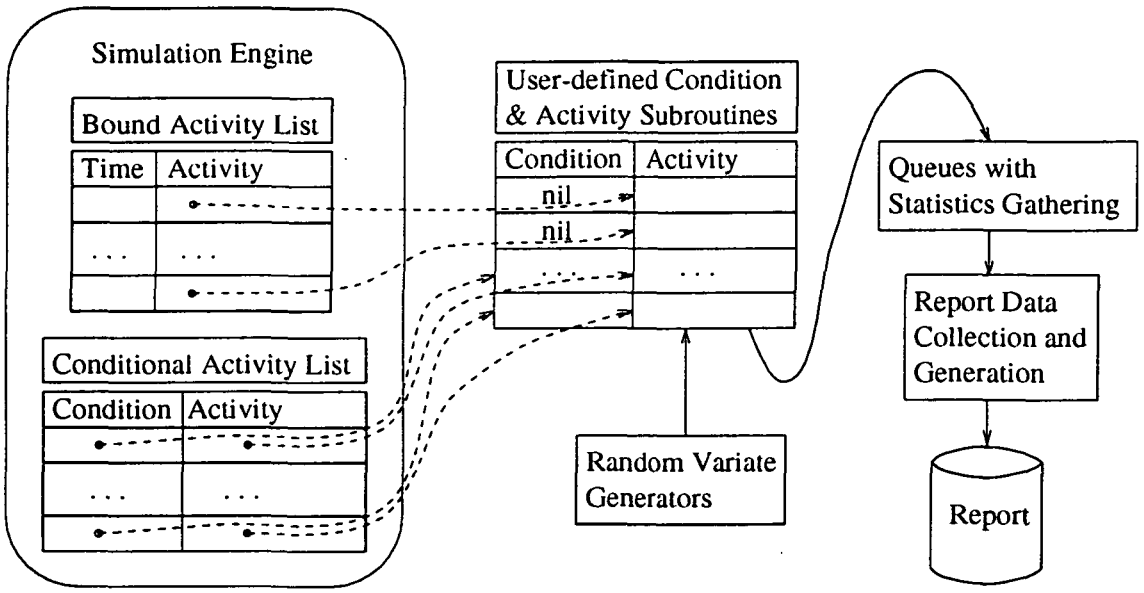

Figure 2.5: Activity-scanning simulation run-time environment

In our CPU/disk model, the B-activities are request generation and end service and the only C-activity is begin service. The pseudocode for the CPU/disk model using the activity-scanning strategy appears in Figure 2.6.

For this model, the conditional event list would contain a single entry for the BeginServiceActivity event. After this model has been executing for awhile and just after a BeginServiceActivity event was executed (i.e., the disk is busy), the bound event list could contain the following.

Event list	
Time	Event
43	GenerateRequest
47	EndServiceActivity

The event list enforces causality by executing bound events in time order. In this case, the simulation would proceed by setting the current time to 43 and executing the GenerateRequest event which would add a new request to the disk request queue and schedule a new GenerateRequest event at some future time, say 50. After executing the GenerateRequest event, the simulator engine would scan each conditional activity and check if the associated condition evaluated to true, and, if so, execute that conditional event. Since the disk is currently busy, the

GenerateRequest: *Bound activity*

1. Draw number of sectors to access from an exponential distribution rounding to the nearest integer
2. Enqueue the request onto the disk request queue
3. Draw an interarrival time from an exponential distribution
4. Schedule a GenerateRequest event for the current time + interarrival time

BeginServiceActivity: *Activate if disk request queue not empty and disk not busy*

1. Take next request from the head of the disk request queue
2. Compute service time as the sum of the seek, rotational latency, and transfer times
3. Schedule an EndServiceActivity event for the current time + service time
4. Mark the disk as busy

EndServiceActivity: *Bound activity*

1. Mark the service as completed and log waiting and service times
2. Mark the disk as idle

Figure 2.6: Activity-scanning pseudocode for the CPU/disk model

condition associated with the BeginServiceActivity is not met. Thus, the conditional activity is not executed at this time. The event list after the execution of the event at time 43 is as follows:

Event list	
Time	Event
47	EndServiceActivity
50	GenerateRequest

The simulation would proceed by setting the current time to 47 and executing the EndServiceActivity event. This would remove the request and mark the disk as idle. Then the simulator would scan the conditional activity list and determine that the BeginServiceActivity was activated. Thus, it would now execute the associated event subroutine. This would then schedule an EndServiceActivity for some time after time 50, say 59. At this point the event list would be as follows:

Event list	
Time	Event
50	GenerateRequest
59	EndServiceActivity

We are back to the case we started with.

As can be seen in this example, the bound event list, event dispatcher, and conditional event list are central to the simulation. As models increase in complexity, the number of conditional events will also increase. In addition, as in the event-scheduling approach, the bound list can grow quite large. Clearly the data structure used to implement these lists significantly affects the run-time performance of the simulation.

The advantage of this approach over the event-scheduling approach is that the condition triggering each conditional activity is associated with that activity rather than being distributed into many other events. However, there is a cost. The run-time system must now scan all conditional activities after each bound activity is executed to see if they are triggered. Thus activity-scanning provides for an increase in modularity, but at a run-time cost that, in complex models, could grow quite large. The activity-scanning strategy is sometimes extended by having multiple conditional event lists, one associated with each bound event in the system. This approach reduces the time wasted in needlessly scanning events that will not be activated.

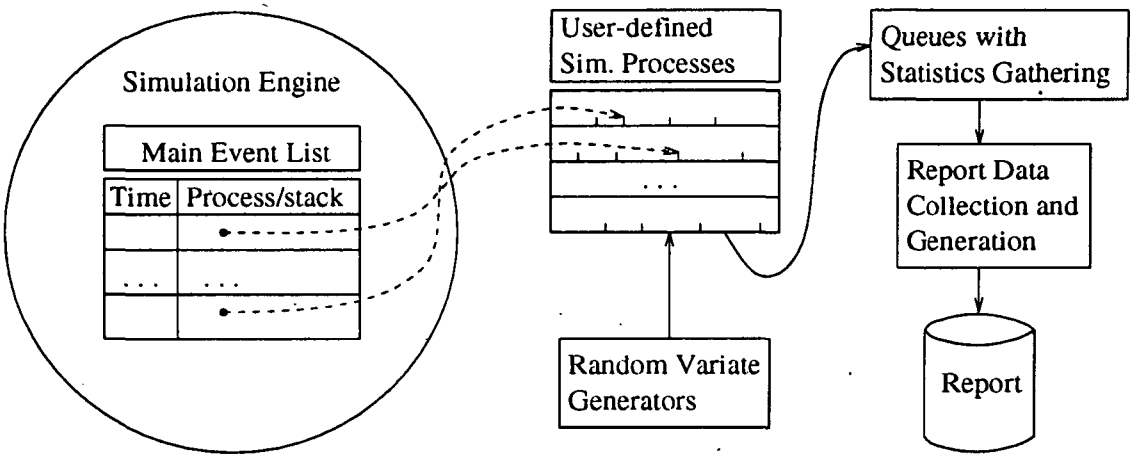

Figure 2.7: Process-interaction simulation run-time environment

2.3 Process-Interaction Strategy

In the process-interaction strategy, a system is modeled as a collection of cooperating simulation processes. These simulation processes, like threads, execute in a single memory space. However, only the simulation process containing the current event is ready to execute at any given time. Such restricted threads are sometimes called semi-coroutines [16]. Depending on the facilities provided by the operating system, simulation processes may be implemented using threads, coroutines, or a custom machine-dependent package to provide the semantics required for simulation processes. We will refer to the processes in a process-interaction model as simulation processes.

Unlike the event-scheduling and activity-scanning strategies, the flow of control through a process-interaction model is captured implicitly by the simulation process. Processes interact with each other through resource queues and other synchronization mechanisms such as barrier synchronization and mailboxes. Note that in the case of operating system and network simulations, the process-interaction strategy matches very closely to the way the system is actually defined. In fact, real operating system processes can be integrated with a process-based simulation with little or no changes to the code. This permits more realistic modeling of fault handling algorithms in dependability simulation. A simplified diagram of the process-interaction run-time environment is given in Figure 2.7.

The primary difficulty with the process-interaction strategy is that executing an event now corresponds to reactivating a simulation process. The reactivation of simulation processes is controlled by the list of future events and generally involves saving all user-visible CPU registers on the stack associated with the previous simulation process, switching to the stack associated with the next simulation process, restoring CPU registers from the stack, and returning control to the next simulation process through a subroutine return statement. It is important to note that the context switches between simulation processes never preempts a simulation process; the simulation process voluntarily returns control to the simulation engine whenever it waits for a future time or event to occur (the operating system is free to preempt the simulation program to run other programs).

In the case of the process-interaction strategy, our CPU/disk model is decomposed into two simulation processes, the request generator and disk server. Note that now the begin and end disk service events are combined into a single simulation process. This process waits for a request to arrive and services the request by waiting for the computed service time to elapse. This process is repeated until the simulation is terminated either by a limit on the number of disk requests, the simulation time or the run-time. The pseudocode for the CPU/disk model using the process-interaction strategy appears in Figure 2.8.

After the model has been initialized, but before the simulation has actually begun, the event list would appear as follows:

Event list		
Time	Process context	Stack contents
0	Beginning of RequestGenerator	Empty
0	Beginning of DiskServer	Empty

The simulation would proceed by setting the current time to 0 and activating the Request-Generator process at the beginning using an initially empty stack. The process would execute up through line 1(b) which would schedule the RequestGenerator process to reactivate at the computed interarrival time with the reactivation point being after line 1(b). If we assume that the computed interarrival time was 8, then the event list after the initial execution of the RequestGenerator process is as follows:

RequestGenerator:

1. while (!done)
   (a) Draw an interarrival time from an exponential distribution
   (b) Wait for the computed interarrival time to pass
   (c) Draw number of sectors to access from an exponential distribution rounding to the nearest integer
   (d) Enqueue the request onto the disk request queue

DiskServer:

1. while (!done)
   (a) Dequeue a request from the disk request queue (implicitly waiting, if none available)
   (b) Compute service time as the sum of the seek, rotational latency, and transfer times
   (c) Wait for the computed service time to pass
   (d) Mark the service as completed and log waiting and service times

Figure 2.8: Process-interaction pseudocode for the CPU/disk model

Event list		
Time	Process context	Stack contents
0	Beginning of DiskServer	Empty
8	After line 1(b) of RequestGenerator	Local vars.

Next, the simulation engine would activate the DiskServer process at the beginning using an initially empty stack. The process would then execute up to line 1(a) where it attempts to dequeue a disk request. Since there are no requests on the disk request queue at this time, the DiskServer process is removed from the event list and placed on the waiting list that is built-in to the queue. Thus, the event list would appear as follows after the initial execution of the DiskServer process:

Event list		
Time	Process context	Stack contents
8	After line 1(b) of RequestGenerator	Local vars.

Next, the simulation engine would set the current time to 8 and reactivate the RequestGenerator process after line 1(b) restoring any local variables by restoring the stack. When this

process enqueues a disk request in line 1(d), the disk request queue will automatically place the DiskServer process in the event list scheduled for the current time. Then the RequestGenerator process continues execution looping back to lines 1(a) and (b) where it again draws a random interarrival time and reschedules itself to reactivate at the current time plus the computed interarrival time, with the reactivation point being after line 1(b). If we assume that the computed interarrival time was 6, then the event list after the execution of the RequestGenerator process is given below.

Event list		
Time	Process context	Stack contents
8	In dequeue subroutine	Dequeue local vars. Return after line 1(a) of DiskServer DiskServer local vars.
14	After line 1(b) of RequestGenerator	RequestGenerator local vars.

Next, the simulation engine would reactivate the DiskServer process in the Dequeue subroutine restoring the stack so that a subroutine return from the Dequeue subroutine will return to the DiskServer process. At this point, the DiskServer process computes a service time and reschedules itself to reactivate at the current time plus the computed service time with the reactivation point being after line 1(c). If we assume that the computed service time was 4, then the event list after the execution of the DiskServer process is as follows:

Event list		
Time	Process context	Stack contents
12	After line 1(c) of DiskServer	Local vars. for DiskServer
14	After line 1(b) of RequestGenerator	Local vars. for RequestGenerator

Next, the simulation engine would set the current time to 12 and reactivate the DiskServer process after line 1(c) restoring any local variables by restoring the stack. This process would then mark the current request as completed and loop back to line 1(a) where it would attempt to dequeue another disk request from the queue. Since the queue is once again empty, this would remove the DiskServer process from the event list and place it on the waiting list at the disk request queue. At this point, the event list would appear as follows:

Event list		
Time	Process context	Stack contents
14	After line 1(b) of RequestGenerator	Local vars.

We are back to the third event list condition that occurred above. Note that in the process-interaction strategy, queues take an active role in the simulation process. The queue blocks the dequeueing process until an entry is available and reactivates the process as soon as an entry is available. In general, a request queue will contain at least two internal queues: the actual request queue and the queue of blocked readers. If the queue has a maximum capacity, then it will also maintain a queue of blocked writers. One very positive effect of this approach is that the model itself requires no "nudging" signals as required in the event-scheduling strategy while avoiding unnecessary condition checks as required by the activity-scanning strategy. Note that the abstractions used in the process-interaction strategy are ideal for modeling operating systems and their components. This eases the task of the simulationist in implementing computer system models as they inevitably model parts of the operating system behavior.

3. RANDOM VARIATE GENERATION AND STATISTICS GATHERING

Simulation of computer systems normally requires the generation of random number sequences to provide input stimuli as well as internal response times. These random number sequences must obey the required probability law governing each component in the system. Among the common requirements are streams of random numbers that are independent and identically distributed according to the exponential, hyperexponential, normal, or Weibull distribution.

Thus, the simulation environment must have the capability to produce random variates from a variety of distributions. Fortunately, variates from a wide variety of theoretical and empirical distributions can be generated provided only that a sequence of independent random variates, each with uniform distribution on the interval (0, 1), can be generated. Such a random variate is often referred to as a uniform deviate.

We will describe some of the common techniques to generate uniform deviates in Section 3.1. In Section 3.2 we will describe how to generate random variates given a stream of uniform deviates.

3.1 Uniform Random Number Generation

The most common method generates the next random number in a stream of random numbers as a function of one or more previous random numbers. Since these random numbers are

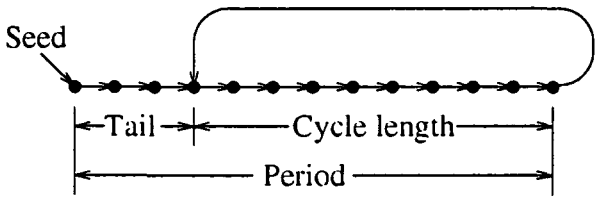

Figure 3.1: Cycle length, tail length, and period of a random-number generator

produced by a deterministic algorithm, they are not truly random, rather, they are *pseudorandom* numbers which satisfy various statistical properties such as independence and uniformity. Note that pseudorandom numbers are sometimes more desirable than truly random numbers as they are repeatable. This repeatability greatly simplifies the debugging and validation of models. Of course, if multiple runs are needed to produce statistically valid results, we have to ensure that each run is started at a different point in the random number sequence; otherwise, all runs would produce the same results. This starting position is called the *seed.*

Consider the following random number generation function,

$$x_n = 5x_{n-1} + 1 \pmod 8.$$

If we start with the seed, $x_0 = 2$, the first 16 numbers obtained by this procedure are 3, 0, 1, 6, 7, 4, 5, 2, 3, 0, 1, 6, 7, 4, 5, 2. Note that the number generated all lie between 0 and 7 and after the 8th number, the sequence repeats. In other words, this function generates only eight unique numbers in a cyclic repetition. Thus, this generator has a *cycle length* of eight. Some generators do not repeat an initial part of the sequence. This part of the sequence that they do not repeat is called the *tail.* In these cases, the *period* of the generator consists of the sum of the tail length and the cycle length. Figure 3.1 illustrates the tail, cycle length, and period of a random-number generator [16, page 438].

Among the classical methods for generating sequences of uniform random numbers are the linear congruential and Tausworthe generators [16, 17, 18]. In addition, recent research by Eichenauer and Lehn indicates that a new technique known as the inversive congruential

generator offers improved statistical properties when compared to those for the classical techniques [19, 20]. Graham compared four combined random number generators that provide an increased period and improved statistical properties [21].

3.1.1 Properties of good random number generators

A good random number generator should be computationally efficient and should generate a sequence of numbers where successive values are both independent and uniformly distributed.

Computational efficiency is important as simulations typically require several thousand or more random numbers to be generated for each run. Successive values have to be statistically independent (free from significant correlation) from each other in order to produce statistically valid results from the simulation runs. In order to obtain statistical independence, it is necessary that the random number generator also have a large period. This large period guarantees that the random number sequence will not recycle. When the sequence recycles, we can no longer be certain that our simulation will continue to produce useful results as our assumption of independence may become invalid.

Computational efficiency and period size are easy to determine; however, statistical independence and uniformity require that the random number generator pass a battery of tests.

3.1.2 Linear congruential generators

A linear congruential generator produces a series of positive integers x_i between 0 and some positive integer m. The following equation describes how to generate the next random number in the stream using a linear congruential generator.

$$x_i \equiv a x_{i-1} + c \pmod{m} \tag{3.1}$$

where a is a positive integer and c is a nonnegative integer. As can be seen, the example used at the beginning of this section was a linear congruential generator with $a = 5$, $c = 1$, and $m = 8$.

We call a the multiplier, c the increment, and m the modulus. The seed of this stream is simply x_0. Some of the advantages of this simple formula are (1) Statistical properties of the

resulting sequence are reasonably well-understood permitting us to select optimal values of a, c, m, and the seed, x_0; (2) it is computationally and memory efficient; and (3) the sequence can be easily reproduced by saving the seed. Over the years, a number of studies have investigated the statistical qualities of these random generators and have determined a few statistically adequate values for a, c, and m [18, 21]. Also note that the special case when $c = 0$ is called the *multiplicative congruential method.*

The period of a linear congruential generator is bounded by the modulus; thus, the modulus m should be large. In order for mod m calculations to be efficient, m should be a power of 2 permitting mod m to be calculated by truncating the result to k bits where $m = 2^k$. In order to generate the maximum possible period, c must not be set to 0. In fact, c must be relatively prime to m. Further, if $a - 1$ is a multiple of 4, we are guaranteed a *full-period* generator.

If $c = 0$ is chosen, then m must not be a power of 2 in order to obtain full period. Note that when $c = 0$, the full-period length is $m - 2$ since 0 must be excluded. Care must be taken to ensure that overflow does not occur as this will give incorrect results when the modulus differs from the maximum unsigned value representable on the computer system. See [22, 23] for a statistical analysis of multiplicative congruential random number generators for optimal values of the a parameter where $m = 2^{31} - 1$ is chosen.

Frequently, it is desirable to allocate ranges of random numbers to several streams. To do this while preserving the statistical independence of the streams, we have to be able to predict the nth next random number to be generated by a given linear congruential generator so that we can set the seed for the next stream to start after n random numbers in the previously allocated stream. The following formula, also a linear congruential generator, will predict the nth next random number in a given stream:

$$x_i = a^n x_{i-n} + c\left(\frac{1 - a^n}{1 - a}\right) \pmod{m}. \tag{3.2}$$

For other types of random number generators, tables of seeds separated by some fixed amount, say 10,000, are used to provide multiple independent random number streams.

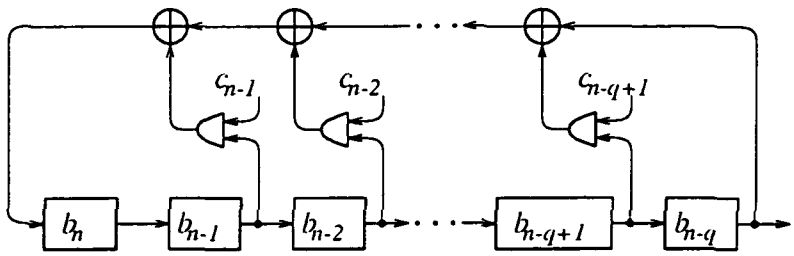

Figure 3.2: Feedback shift register implementation of a random-number generator

3.1.3 Tausworthe generators

Tausworthe random number generators, developed for cryptographic applications, can be used to generate long sequences of pseudorandom numbers. They generate one bit at a time as a function of the last n bits where n is the number of bits in the random number to be generated. The general form for a Tausworthe generator is as follows:

$$b_n = c_{q-1}b_{n-1} \oplus c_{q-2}b_{n-2} \oplus c_{q-3}b_{n-3} \oplus \cdots \oplus c_0 b_{n-q} \tag{3.3}$$

where c_i and b_i are binary variables and $\oplus$ is the exclusive-or (mod 2 addition) operation. This is frequently represented using the polynomial representation where the power of the polynomial variable represents the delay of the corresponding bit and the presence of the polynomial term indicates that the corresponding coefficient $c_i = 1$. When $c_i = 0$, no term is included in the polynomial. Therefore, the polynomial form of the above equation is as follows:

$$x^q + c_{q-1}x^{q-1} + c_{q-2}x^{q-2} + \cdots + c_0. \tag{3.4}$$

See Figure 3.2 for a feedback shift register implementation of a random-number generator using a general q-degree polynomial [16, page 446]. Note that the AND gates depicted in the illustration are not necessary when the coefficients, c_i, are known beforehand.

As an example, consider the following polynomial:

$$x^7 + x + 1.$$

Converting back to the original notation,

$$b_{n+7} = b_{n+1} \oplus b_n, \quad n = 0, 1, 2, \ldots.$$

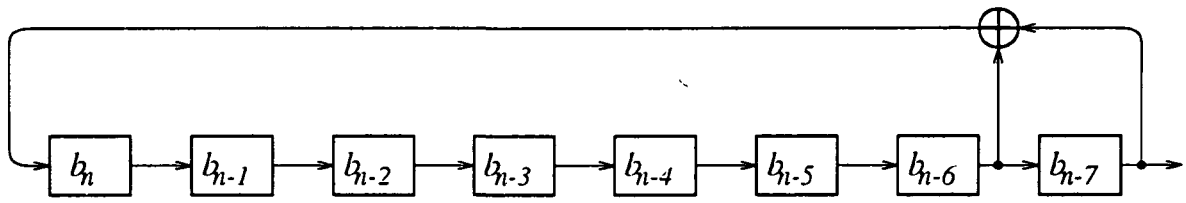

Figure 3.3: Feedback shift register implementation of $x^7 + x + 1$

If we start with $b_0 = b_1 = \cdots = b_6 = 1$, we obtain the following bit sequence:

$$\begin{aligned}
b_7 &= b_1 \oplus b_0 = 1 \oplus 1 = 0 \\
b_8 &= b_2 \oplus b_1 = 1 \oplus 1 = 0 \\
b_9 &= b_3 \oplus b_2 = 1 \oplus 1 = 0 \\
b_{10} &= b_4 \oplus b_3 = 1 \oplus 1 = 0 \\
b_{11} &= b_5 \oplus b_4 = 1 \oplus 1 = 0 \\
b_{12} &= b_6 \oplus b_5 = 1 \oplus 1 = 0 \\
b_{13} &= b_7 \oplus b_6 = 0 \oplus 1 = 1 \\
&\vdots
\end{aligned}$$

The feedback shift register implementation for this polynomial is given in Figure 3.3.

See [24] for a statistical analysis of combined Tausworthe random number generators for optimal statistical properties.

3.1.4 Inversive congruential generators

More recently, research by Eichenauer and Lehn has led to the development of a new inversive congruential pseudorandom number generator [19, 20]. This technique provides more desirable statistical properties of the resulting random number stream than either of the linear congruential or Tausworthe generators.

Let $w \geq 3$ be an integer. For integers a, b with $a \equiv 1 \pmod 2$ a function f is defined by

$$x_i \equiv 2^l a \left(\frac{x_{i-1}}{2^l}\right)^{-1} + b \pmod{2^w} \tag{3.5}$$

with nonnegative integer l and odd integer $x_{i-1}/2^l$, where 0 is identified with 2^w. The inversion of $z = x_{i-1}/2^l$ can be efficiently computed using the Euclidean algorithm [19, page 2]. The sequence generator has maximal period length 2^w if and only if $a = 1 \pmod 4$ and $b = 1 \pmod 2$.

Table 3.1: CDFs and their inverses for several common distributions

Distribution	CDF $F(x)$	Inverse $F^{-1}(u)$
Exponential	$1 - \exp^{-x/a}$	$-a \ln(u)$
Geometric	$1 - (1-p)^x$	$\lceil \frac{\ln(u)}{\ln(1-p)} \rceil$
Weibull	$1 - \exp^{-(x/a)^b}$	$a(-\ln u)^{1/b}$

Clearly this random number generation technique trades increased run-time for the greater statistical randomness inherent in its nonlinear technique.

3.2 Random Variate Generation

Random variates are generated using uniform random numbers as building blocks. Among the techniques used are the inverse transform method, the acceptance-rejection method, composition, and convolution. Each technique is applicable to only a subset of the distributions.

3.2.1 Inverse-transform method

Consider a continuous random variable X with density function $f_X(x)$ and cumulative distribution function (CDF)

$$F_X(x) = \int_{-\infty}^{x} f_X(\theta) d\theta. \tag{3.6}$$

Now, $F_X(x)$ is a nondecreasing function such that

$$\lim_{x \to -\infty} F_X(x) = 0 \quad \text{and} \quad \lim_{x \to +\infty} F_X(x) = 1.$$

The inverse-transform method is based on the observation that the CDF of a random variable maps the random variable X to a value between 0 and 1, namely, $F_X(x)$. The method works by taking the inverse of the CDF, $F_X^{-1}(u)$, generating a uniformly distributed random number, u, between 0 and 1, and computing $x = F^{-1}(u)$, which is a random variate distributed according to the original CDF. Table 3.1 gives the CDFs and their inverse for several common distributions.

3.2.2 Acceptance-rejection method

Consider a continuous random variable X with probability density function (PDF) $f_X(x)$. The acceptance-rejection method can be used if another PDF, $g(x)$, exists that, when multiplied by a constant c, majorizes $f_X(x)$. A function is said to majorize another function, if for all values of the independent variable, x, the majorizing function is greater than or equal to the other function. Given such a function that majorizes $f_X(x)$, namely, $cg(x)$, the following steps can be used to extract random variates according to the desired PDF:

1. Generate x with PDF $g(x)$.
2. Generate y uniform on $[0, cg(x)]$.
3. If $y \leq f_X(x)$, then accept x; otherwise, reject x and repeat from step 1.

Consider the following example which illustrates the use of the acceptance-rejection technique to generate random variates from the Beta distribution with parameters $(\alpha, \beta) = (2, 3)$. The PDF for the Beta(2, 3) is

$$f(x) = 12x(1-x)^2, \quad 0 \leq x \leq 1.$$

This function reaches a maxima at $x = 1/3$ where $f(x) = \frac{16}{9} \approx 1.78$. It can be bounded by a constant function of height 1.78. Thus, we can use a uniform distribution with $c = 1.78$, and

$$g(x) = 1, \quad 0 \leq x \leq 1.$$

To generate the Beta(2, 3) variates, we use the following algorithm:

1. Generate x uniform on $[0, 1]$.
2. Generate y uniform on $[0, 1.78]$.
3. If $y \leq 12x(1-x)^2$, then accept x; otherwise, reject x and repeat from step 1.

Steps 1 and 2 generate a point (x, y) uniformly distributed over the rectangle shown in Figure 3.4. If the point falls above the beta density function $f(x)$, then step 3 rejects x.

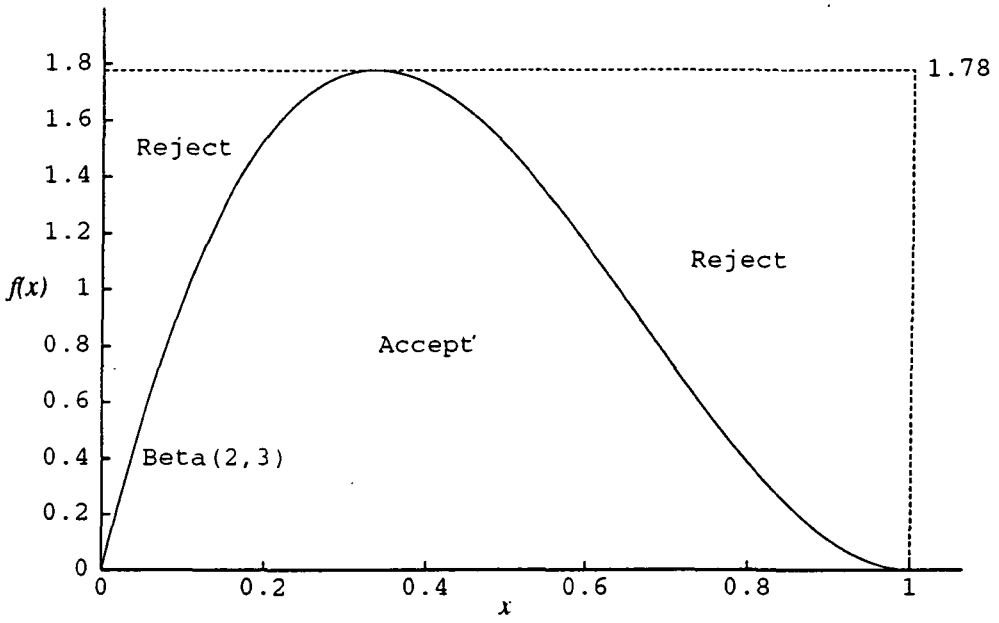

Figure 3.4: Acceptance-rejection method for generating a beta distribution

3.2.3 Composition

This technique can be used if the desired CDF $F(x)$ can be expressed as a weighted sum of n other CDFs, that is,

$$F(x) = \sum_{i=1}^{n} p_i F_i(x). \tag{3.7}$$

Of course, $0 \leq p_i \leq 1$, $\sum_{i=1}^{n} p_i = 1$, and F_i's are CDFs. This same technique is sometimes called *decomposition* referring to the fact that the desired CDF can be decomposed into the sum of n other CDFs.

The steps to generate variates using composition are as follows:

1. Generate a random integer I such that $Prob(I = i) = p_i$ using the inverse-transform method.

2. Generate x with the ith CDF $F_i(x)$ again, using the inverse-transform method and return x.

3.2.4 Convolution

This technique can be used if the random variable x can be expressed as a sum of n easily generated random variables, that is,

$$x = y_1 + y_2 + \cdots + y_n \tag{3.8}$$

Here, x can be generated by simply generating $y_1, y_2, \ldots, y_n$ and then summing them.

This technique is called the convolution technique because the PDF of a random variable x that is the sum of n random variables can be obtained by a convolution of the PDFs of the y_i's. During random number generation, no convolution is required. The Erlang-k distribution may be generated using this technique as it is the sum of k exponential random variates.

3.3 Statistics Gathering and Reporting

Statistics gathering during the simulation and reporting afterwards are necessary in order to interpret the results. Generally, the simulation environment will provide facilities to collect statistics by providing histogram objects. In the process-interaction model, limited resources are modeled using servers which contain resource queues. These queues generally gather statistics for queue length, response time, service time, and throughput rate distributions [25].

4. MOTIVATION

Computer systems are most easily modeled using the process-interaction strategy because of the relatively large size of computer system models and the ease with which the process-interaction strategy maintains the sequence of operations occurring in the system. For higher-level models such as those employed in dependability studies, much of the actual computation has been removed from the model in order to permit simulation of the system over long periods of time. This results in models which have short sections of code between statements that cause simulation time advancement either directly, by waiting for a resource to become available, or through synchronization. Note that even when the models are more detailed, time advancement statements will tend to be nearly as frequent since the system is being modeled in greater detail. Whether we are using structural models to permit simulation for long periods of time or more detailed models which will be simulated over shorter periods of time, models will tend to have short sections of code between time advancement statements.

This characteristic leads to poorer than expected performance because of the high frequency and relatively long time involved in simulation process reactivation. Recall that reactivation involves saving CPU registers for the previous simulation process, switching to the next process's stack, restoring CPU registers, and returning control to the next simulation process. Modern RISC processors have anywhere from 47 (i860) to 160 (SPARC) 32-bit registers that must be saved/restored at each simulation process reactivation (see Table 4.1). This represents quite a large fraction of the simulation time in models that switch between simulation processes

Table 4.1: Register counts for several recent microprocessors

	i860	MIPS	M88000	SPARC	Pentium
32-bit Integer Registers	31	31	31	31	6
Floating-point Registers	30 x 32 bits	16 x 32 bits	0	32 x 32 bits	8 x 80 bits
Register Window Size				128 to 136	
Total 32-bit Registers	61	47	31	164	26

frequently. Further, newer architectures tend to include more rather than less registers on-chip resulting in even higher reactivation times as compared to actual model execution times. Finally, these problems are further exacerbated by the fact that the fault-tolerance analysis requires very large run times.

Our approach combines the efficiency of the event-scheduling strategy with the ease of model specification of the process-interaction strategy by transforming an input process-interaction model using compiler-based techniques to avoid the need for stack switching to achieve simulation process reactivation. Simulation process reactivation is essentially replaced with a subroutine call and return and local simulation process variables are allocated explicitly in memory. A slightly increased overhead is added to the model in that modified variables must be saved to memory upon return, whereas previously they could, in certain cases, remain in CPU registers that were saved and restored by reactivation. The major increase in performance comes from avoiding the saving and restoring of large numbers of CPU registers. Tests performed on the SPARC architecture indicate that we have successfully reduced the event-to-event times from 210 μs to 2.5 μs. And, of that original 210 μs, 200 were in the save/restore register code which involves a system trap to flush the register windows. On an IBM RS/6000 system, the performance improvement is less dramatic due to fewer registers that must be saved and restored, but it is still significant with event-to-event switch times reduced from 35.0 down to 3.5 μs

5. COMPILER-BASED PROCESS-INTERACTION STRATEGY

Our approach is to use compiler techniques to transform a model based on the process-interaction strategy into a simpler event-driven model. This event-driven model then executes in a new run-time environment based on the previously developed DEPEND simulation-based environment [9, 10, 11]. Figure 5.1 illustrates our simulation-based environment.

The key compiler technique used is to break down processes into a set of constituent event subroutines. Unfortunately, this also requires our run-time environment to explicitly maintain the call-return stack within processes and to explicitly allocate memory to contain what were formerly local variables for those subroutines. We combine these two functions into a single data structure that is called an activation record. This increased overhead for handling activation records explicitly is more than offset by the simpler simulation engine that is possible in our approach as compared to that for the process-interaction model. Furthermore, the event-scheduling and activity-scanning strategies cannot handle multiple instances of a given event whereas our compiler-based strategy handles these multiple instances through maintaining local variables (as does the process-interaction model). A simplified diagram of our compiler-based process-interaction run-time environment is given in Figure 5.2.

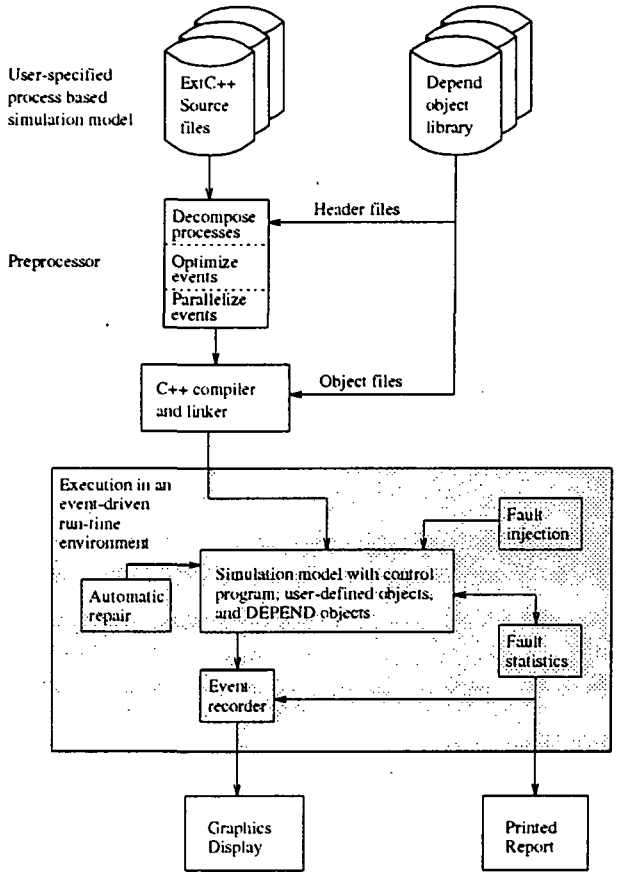

Figure 5.1: DEPEND simulation-based environment

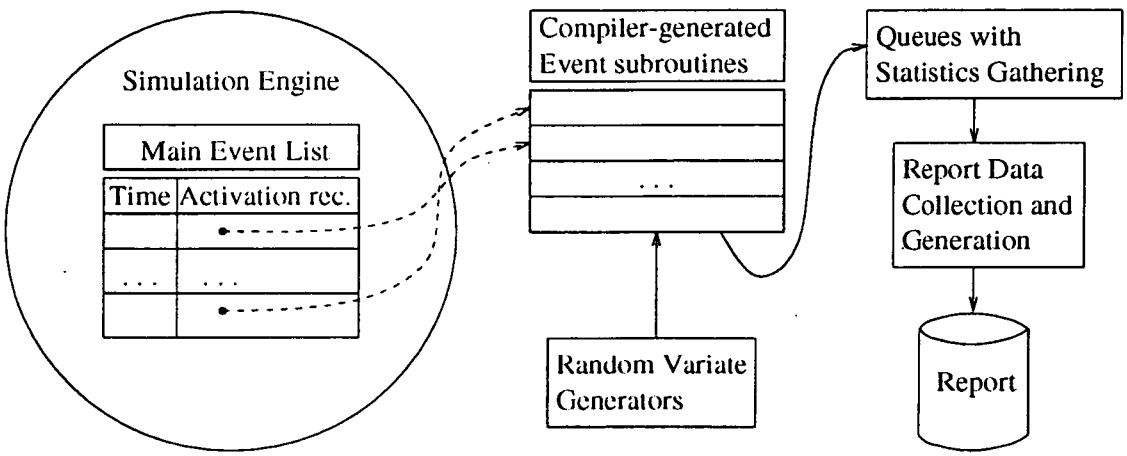

Figure 5.2: Compiler-based process-interaction run-time environment

RequestGenerator:

1. while (!done)
   (a) Draw an interarrival time from an exponential distribution
   (b) Wait for the computed interarrival time to pass
   (c) Draw number of sectors to access from an exponential distribution rounding to the nearest integer
   (d) Enqueue the request onto the disk request queue

DiskServer:

1. while (!done)
   (a) Dequeue a request from the disk request queue (implicitly waiting if none available)
   (b) Wait until request available
   (c) Compute service time as the sum of the seek, rotational latency, and transfer times
   (d) Wait for the computed service time to pass
   (e) Mark the service as completed and log waiting and service times

Figure 5.3: Process-interaction pseudocode for the CPU/disk model

5.1 CPU/Disk Model

Recalling the CPU/disk model used as an example in the description of the classical discrete event simulation world views, we will use this same model, now, to demonstrate our technique. Figure 5.3 duplicates the pseudocode given earlier for the process-interaction model so that it can be more easily compared with the transformed code in Figures 5.4 and 5.5.

As done for each of the simulation world views, we will now step through the simulation of this model. After the model has been initialized, but before the simulation has actually begun, the event list would appear as follows:

Event list	
Time	Activation record
0	Uninitialized local vars. and pointer to RequestGenerator1 event
0	Uninitialized local vars. and pointer to DiskServer1 event

RequestGenerator1:

1. if (!done)
   (a) Draw an interarrival time from an exponential distribution
   (b) Change reactivation routine to RequestGenerator2
   (c) Return to wait for the computed interarrival time to pass
2. otherwise, terminate the RequestGenerator process

RequestGenerator2:

1. if (!done)
   (a) Draw number of sectors to access from an exponential distribution rounding to the nearest integer
   (b) Enqueue the request onto the disk request queue
   (c) Draw an interarrival time from an exponential distribution
   (d) Return to wait for the computed interarrival time to pass
2. otherwise, terminate the RequestGenerator process

Figure 5.4: Event pseudocode for the RequestGenerator process after transformation

The simulation would proceed by setting the current time to 0 and calling the RequestGenerator1 event using the uninitialized activation record. The event would compute the interarrival time and return specifying the time to be reactivated along with the event to call upon reactivation, RequestGenerator2. If we assume that the computed interarrival time was 8, then the event list after the execution of the RequestGenerator1 event is as follows:

Event list	
Time	Activation record
0	Uninitialized local vars. and pointer to DiskServer1 event
8	Local vars. and pointer to RequestGenerator2 event

Next, the simulation engine would call the DiskServer1 event using the uninitialized activation record. The event would check if a request were available on the disk request queue. Since no requests are currently on the queue, it would place itself on the waiting list at the disk service queue and return specifying that the event to call upon reactivation would be DiskServer2. Thus, the event list would appear as follows after the initial execution of the DiskServer1 event:

DiskServer1:

1. if (!done)
   (a) Change reactivation routine to DiskServer2
   (b) Dequeue a request from the disk request queue
   (c) Return to wait until request available
2. otherwise, terminate the DiskServer process

DiskServer2:

1. Compute service time as the sum of the seek, rotational latency, and transfer times
2. Change reactivation routine to DiskServer3
3. Return to wait for the computed service time to pass

DiskServer3:

1. Mark the service as completed and log waiting and service times
2. if (!done)
   (a) Change reactivation routine to DiskServer2
   (b) Dequeue a request from the disk request queue
   (c) Return to wait until request available
3. otherwise, terminate the DiskServer process

Figure 5.5: Event pseudocode for the DiskServer process after transformation

Event list	
Time	Activation record
8	Local vars. and pointer to RequestGenerator2 event

Next, the simulation engine would set the current time to 8 and call the RequestGenerator2 event using the activation record. When this event enqueues a disk request in line 1(b), the disk request queue will automatically place the DiskServer process (as represented by the activation record currently pointing to the DiskServer2 event) in the event list scheduled for the current time. Then the RequestGenerator2 event continues execution to lines 1(c) and 1(d) where it draws a random interarrival time and reschedules itself to reactivate at the current time plus the computed interarrival time by returning to the simulation engine. Note that the RequestGenerator2 event remains the current event for this process until the simulation is completed. If we assume that the computed interarrival time was 6, then the event list after the execution of the RequestGenerator2 event is

Event list	
Time	Activation record
8	Local vars. and pointer to DiskServer2 event
14	Local vars. and pointer to RequestGenerator2 event.

Next, the simulation engine would call the DiskServer2 event. At this point, it computes a service time and schedules the DiskServer3 event to activate at the current time plus the computed service time and returns to the event scheduler. If we assume that the computed service time was 4, then the event list after the execution of the DiskServer2 event is

Event list	
Time	Activation record
12	Local vars. and pointer to DiskServer3 event
14	Local vars. and pointer to RequestGenerator2 event.

Next, the simulation engine would set the current time to 12 and call the DiskServer3 event using the activation record. This event would then mark the current request as completed and then attempt to dequeue another disk request from the queue. Since the queue is once again empty, it would place itself on the waiting list at the disk service queue and return specifying that the event to call upon reactivation would be DiskServer2. At this point, the event list would appear as follows:

Event list	
Time	Activation record
14	Local vars. and pointer to RequestGenerator2 event

We are back to the third event list condition that occurred above. Note that like the process-interaction strategy, queues are still active in the simulation process; they stall readers until queue entries are available and reschedule them as soon as a queue entry is available. Next, we will examine in a more formal manner the source of speedup over the process-interaction approach.

5.2 Source of Speedup over Process-Interaction Simulation

Our technique achieves its improved performance by avoiding the costly context switches used in process-interaction simulation. In effect, the context switch is replaced by a subroutine call/return pair and a few additional memory references. The following equations express the expected speedup from our technique.

$$speedup = \frac{t_{old}}{t_{new}} \tag{5.1}$$

where

$$t_{old} = \sum_{\text{all events}} (t_{\text{old event}} + t_{resched} + t_{cs}) \tag{5.2}$$

$$t_{new} = \sum_{\text{all events}} (t_{\text{new event}} + t_{resched} + t_{cr}) \tag{5.3}$$

$$t_{\text{old event}} = \text{time to execute the user-written event code} \tag{5.4}$$

$$t_{\text{new event}} = t_{\text{old event}} + n_{\text{extra memrefs}} * (t_{memacc} - t_{regacc}) + frac_{\text{time adv sub calls}} * t_{\text{alloc activa. record}} \tag{5.5}$$

$$t_{resched} = \text{time to reschedule this event on the future event list} \tag{5.6}$$

$$t_{cs} = \text{context switch time} \tag{5.7}$$

$$t_{cr} = \text{subroutine call + return time} \tag{5.8}$$

and t_{cs} includes the time to save and restore all of the user-visible CPU registers plus the time to flush register windows (if any). Naturally, this time is architecture dependent, but for many

RISC architectures, this represents a significant overhead when compared to $t_{\text{old event}}$. For the SPARC architecture with its register windows,

$$t_{cs} = 130 * (t_{memWrite} + t_{memRead}) \tag{5.9}$$

In the next chapter, we will describe in greater depth this compiler-based transformation as well as the underlying run-time support for our technique.

6. DETAILED DESCRIPTION OF OUR SIMULATION TECHNIQUE

The run-time environment for the simulation uses a modified form of the event-scheduling strategy. All simulation execution is controlled by the event list which is maintained by the simulation kernel. The event list maintains the simulation time ordering of all scheduled events. Before the simulation executive is entered, the user invokes one or more coroutines which will be executed at simulation time 0. Note that in this context when we refer to a coroutine, we are referring to a simulation process as specified in the input to our extended C++ precompiler.

Once the simulation executive is entered, simulation proceeds until there are no more events on the event list. Figure 6.1 illustrates the data structures associated with the main event list. Note that the actual data structure used to time-order the events in the main event list is not shown. The current implementation includes singly and doubly linked lists and a B-tree-based indexed list (for more efficient event-list manipulation when many processes are active at the same time).

Each simulation process (coroutine) instance is managed by its corresponding coroutine control block. Since, by definition, a coroutine instance cannot have more than one next event at a time, this control block also serves as the unit of event scheduling (much like a process control block in an operating system). This control block contains the status of the coroutine, active or blocked, the reactivation time for the coroutine when active, and a pointer to the top of the activation record stack for the current context of the coroutine. All coroutines on the main event list must be active (i.e., ready to execute at some definite future simulation time).

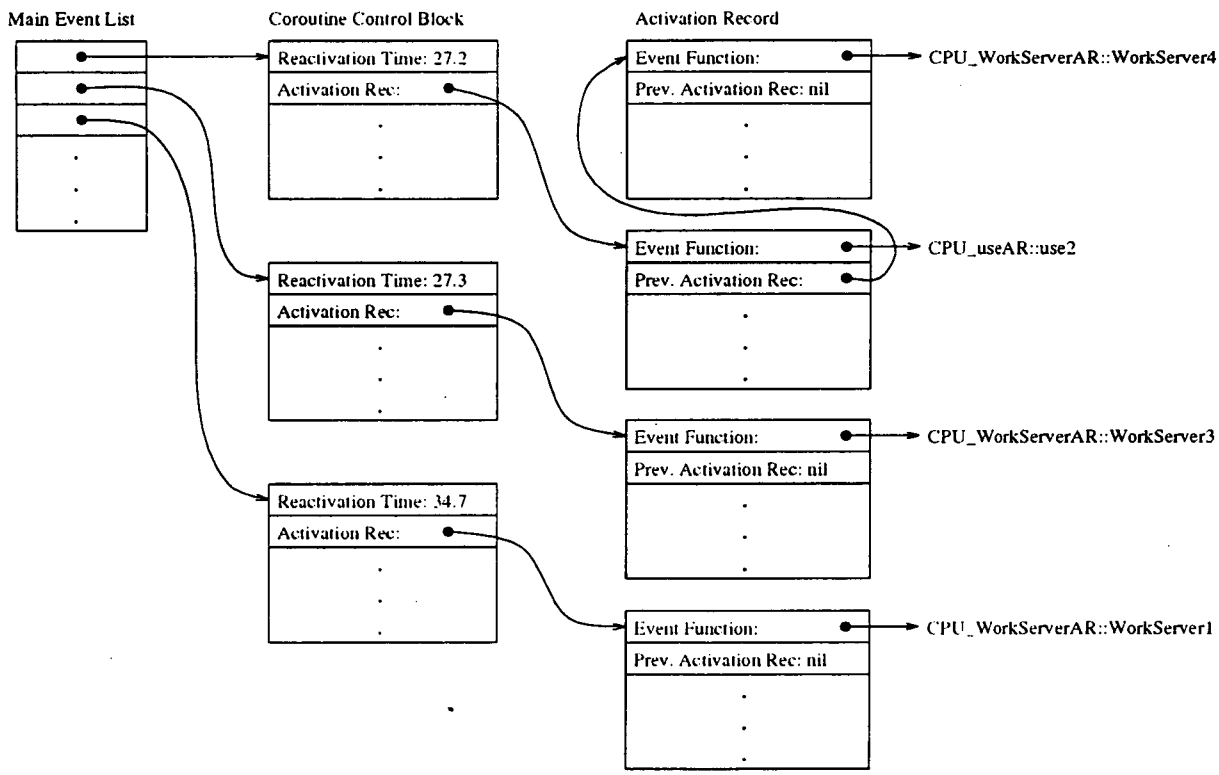

Figure 6.1: The main event list data structures

In addition to what was listed above, the coroutine control block also contains an identification number.

The activation record pointed to by the coroutine control block contains the contents of all of the nontemporary local variables in the original subroutine that this activation record represents. In addition, it contains a pointer to the event function to be invoked when the coroutine is reactivated and a pointer to the previous activation record, if any. The previous activation record pointer is used to support coroutines calling other functions which have the potential for advancing simulation time.

This ability for a coroutine to call a subroutine which itself advances simulation time is unique and permits the user to write well-structured code for the simulation processes. Some other simulation environments such as Maisie require the user to "flatten" the process hierarchy since they do not allow subroutines which advance simulation time [26, 27, 2]. The most common function to use this capability is the use function associated with each server queue. Without this feature, a much greater portion of DEPEND would have to be written into the C++ precompiler.

6.1 Extended C++ Precompiler

The input language to our precompiler is an extended version of C++. The extensions include two additional function specifiers, *coroutine* and *timeAdvance*, and a pseudofunction, *hold*. All other features of process-interaction simulation are supplied through the included C++ object library.

We also provide for the specification of fault-handling functions which are conceptually similar to the proposed exception handling system for C++. The default fault-handler behaves as follows: If the coroutine is scheduled on the future event list, it is rescheduled to execute at the current time (after the fault injection is completed); otherwise, no action is taken. The fault-handler receives as input a pointer to the fault description. Note that fault handlers have access to all local variables of the time-advancement function so that they can easily determine

```
1:  int done = 0;
2:  timeAdvance void scrubMem( Memory* mem, int memoryLoc,
                               int scrubAmount );
3:
4:  coroutine memScrubProc( Memory* mem, int scrubIncrement,
                            double scrubInterval )
5:    {
6:      int memoryLoc = 0;
7:
8:      while (!done)
9:        {
10:         hold( scrubInterval );
11:         scrubMem( mem, memoryLoc, scrubIncrement );
12:         memoryLoc = (memoryLoc + scrubIncrement) %
                        mem->getMemorySize();
13:       }
14:   }
```

Figure 6.2: Example source code illustrating keywords

the state of the coroutine being injected. Furthermore, multiple fault-handlers can be defined, if desired, where the fault-handler is changed by the user as the function proceeds.

Simulation processes are identified by the presence of the *coroutine* function specifier in the function prototype and definition. Coroutines return a pointer to the created coroutine class. This returned pointer may be used to control the coroutine. This permits aborting holds early or suspending, resuming, or terminating the coroutine. This kind of control permits importance sampling techniques to be implemented. A coroutine is created by invoking a function just as would be done in a subroutine call. In order to permit separate compilation, subroutines which advance time (or call other subroutines that advance time) must be flagged with the *timeAdvance* keyword. All explicit time advancement is specified through the use of the *hold* function. Figure 6.2 illustrates the use of each of these keywords in a simple memory scrubbing process.

During the precompilation step, a control flow graph (CFG) is constructed for each coroutine and time-advancement subroutine. Nodes in the CFG represent statements which are executed at the same simulation time (i.e., during a single event). The edges in the CFG which represent

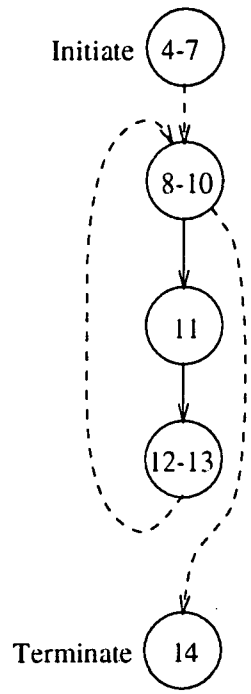

Figure 6.3: Control flow graph for the memory scrubbing process

time advancement are solid, all other edges are dashed. Figure 6.3 shows the CFG for the memory scrubbing process. Note that each node in the graph contains the statement numbers assigned to that node.

In addition to the construction of the CFG, the precompilation step also extracts all function argument variables and local variables. Storage for these extracted variables is allocated in a structure associated with each time advancement routine called an activation record. When a coroutine calls a time advancement routine, a new activation record is created for the time advancement routine and pushed onto the coroutine's stack of activation records. This stack of activation records replaces the individual coroutine stacks employed in typical implementations of process-based simulation. In addition, the activation record stores the next reactivation point for each time advancement subroutine. Figure 6.4 gives the activation record created for the example memory scrubbing process.

```
class memScrubProcAR : public ActivationRecord
  {
  private:
    // Coroutine argument variables
    Memory* mem;
    int scrubIncrement;
    double scrubInterval;
    // Coroutine local variables
    int memoryLoc;
    // Component event function prototypes
    double memScrubProc1(), memScrubProc2(), memScrubProc3();
  public:
    memScrubProcAR( Memory* p1, int p2, double p3 )
      :  mem( p1 ), scrubIncrement( p2 ),
         scrubInterval( p3 ), memoryLoc( 0 )
      { func = (AR_FUNC) &memScrubProcAR::memScrubProc1; }
  }
```

Figure 6.4: Activation record for memory scrubbing coroutine

The control flow graph is then used to decompose the coroutine into a coroutine creation function and a set of event functions. Figure 6.5 shows the coroutine creation and event functions.

After the precompiler has transformed all coroutines and time-advancement subroutines into their constituent event functions and activation records, the resulting C++ source code is compiled and linked with the DEPEND object library to produce the simulation program.

6.2 Optimization Techniques

Once the coroutines and time-advancement routines are decomposed into their constituent events by the control flow graph, these events can be optimized. Recall that the edges in the control flow graph can be categorized as either time advancement edges or re-entry edges. The re-entry edges were necessitated by the fact that one or more time advancement instructions existed inside of a loop. These edges can be removed by concatenating the target node's code onto the end of the source node's code. Heuristics will be employed to avoid excessive

```
double scrubMem(Memory* mem, int memoryLoc, int scrubAmount);

Coroutine* memScrubProc( Memory* mem, int scrubIncrement,
                         double scrubInterval )
  {
    return new Coroutine( new memScrubProcAR( mem,
                                    scrubIncrement,
                                    scrubInterval ) );
  }

double memScrubProcAR::memScrubProc1()
  {
    if (!done)
      {
        func = (AR_FUNC) &memScrubProcAR::memScrubProc2;
        return scrubInterval;
      }

    // Terminate this coroutine
    delete this;
    return -1.0;
  }

double memScrubProcAR::memScrubProc2()
  {
    func = (AR_FUNC) &memScrubProcAR::memScrubProc3;
    return scrubMem( mem, memoryLoc, scrubIncrement );
  }

double memScrubProcAR::memScrubProc3()
  {
    memoryLoc = (memoryLoc + scrubIncrement) %
                mem->getMemorySize();
    return memScrubProc1();
  }
```

Figure 6.5: Coroutine creation and event functions for the memory scrubbing process

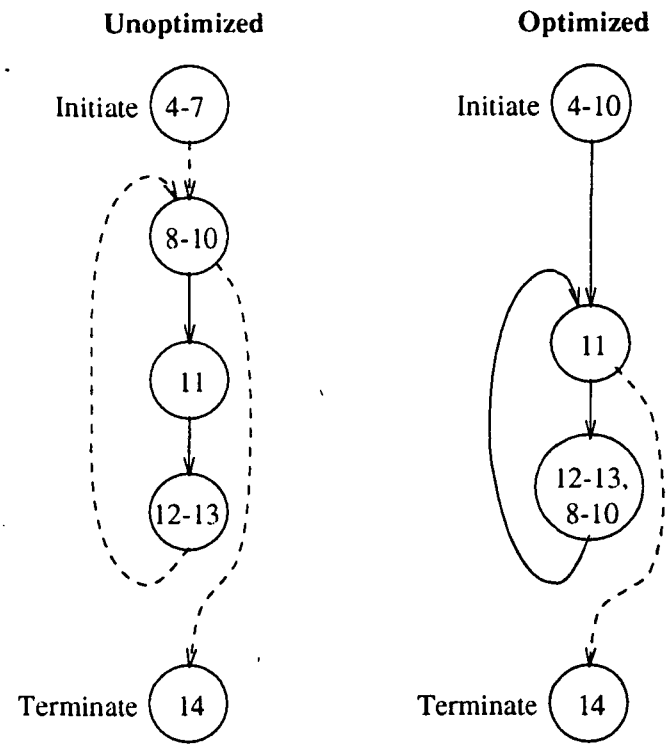

Figure 6.6: Optimized control flow graph for the memory scrubbing process

code duplication. This can be illustrated using the memory scrubbing coroutine from the previous section. Here the first and second nodes as well as the fourth and second nodes can be concatenated to form the re-entry edge optimized control flow graph. Figure 6.6 shows both the optimized and unoptimized control flow graphs.

Further optimizations can be obtained by analyzing the possible interactions between a coroutine and other coroutines. In particular, adjacent time advancement statements may be combined into a single time advancement that holds for the total time of the original statements as long as only local variables are updated between the holds. Figure 6.7 illustrates a time-advancement subroutine where this optimization can be taken.

This subroutine scrubs memory by reading the specified range of memory locations one at a time. To simulate the time it takes to scrub a single memory location, this subroutine advances time 80 ns for each read access. In this case, these time advancements can be combined together by removing the hold statement from the loop and issuing a single hold after the loop for the

```
1:  int done = 0;
2:  timeAdvance void scrubMem( Memory* mem, int memoryLoc,
        int scrubAmount )
3:    {
4:      for ( int i = 0; i < scrubAmount; i++, memoryLoc++ )
5:        {
6:          mem->read( memoryLoc );
7:          hold( 80.0E-9 );
8:        }
9:    }
```

Figure 6.7: Example to illustrate time-advancement optimization

```
1:  int done = 0;
2:  timeAdvance void scrubMem( Memory* mem, int memoryLoc,
        int scrubAmount )
3:    {
4:      for ( int i = 0; i < scrubAmount; i++, memoryLoc++ )
5:        {
6:          mem->read( memoryLoc );
7:        }
8:      hold( 80.0E-9 * scrubAmount );
9:    }
```

Figure 6.8: Time-advancement optimized code

total time to scrub the specified range of memory addresses. The resulting code is given in Figure 6.8. As a result of taking these optimizations, the number of memory scrubbing events has been reduced by a factor of the scrubAmount.

When one coroutine sends a message to a message queue served by another coroutine, instead of scheduling an event for the receiving coroutine, in certain cases the message can begin to be processed by this coroutine.

6.3 Parallelization Techniques

Several techniques are available for the parallelization of the simulation model to execute on a multiprocessor system. These techniques include

1. the offloading of simulation support tasks such as random variate generation, statistics gathering/reporting, and event-list management to other processors,

2. the execution of multiple instances of the complete model in parallel using different random number seeds for each simulation run, and

3. the parallelization of the actual model by allocating simulation processes to pods which run on individual processors.

We believe that the first two techniques will yield the greatest improvement for most computer system models because of the inherent coupling that exists between components in the computer system. Parallelization can succeed if a good partitioning of the model exists that results in relatively independent submodels.

The model chosen for parallelization can be implemented on both shared/distributed memory and message-passing systems with equal ease. The model groups simulation processes into pods which are assigned to processors at simulation run-time along with the simulation support tasks. The simulation support tasks behave in a similar way to process pods. Pods then communicate through explicit message passing using ports. Ports can be declared as either one-way or bidirectional connections. The network of port interconnections between pods provides the necessary information for maintaining time synchronization. Figure 6.9 illustrates the network of pods interconnected by ports.

For example, to implement the off-loading of random variate generation, the random variate stream class is converted into a client of the random variate generator task. When the class is initialized, the variate parameters (e.g., desired mean and variance) are sent to the random variate generator task as a message. This message includes the identification of the port to receive the resulting stream of random variates. Then, as the random variate generator task has time, it precomputes the first batch of random variates and sends them to the initially specified port. Then, when the simulation requests a random variate from that stream, the message is read and the first variate extracted from the message. The important point here is that as long as the random variate generator task can "keep up," the stream of random variates is generated

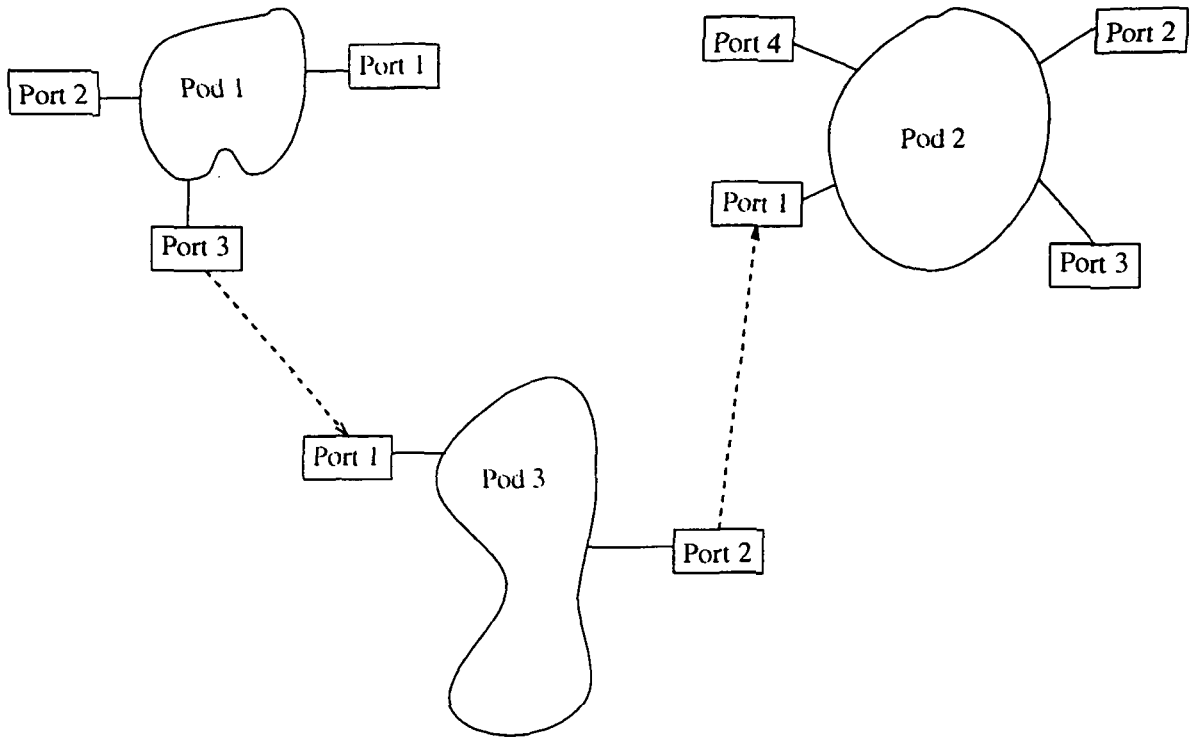

Figure 6.9: Network of pods interconnected by ports

for the cost of receiving a message divided by the number of random variates packed in each message.

7. CASE STUDY

This chapter presents a preliminary evaluation of our new approach through the use of a simple case study. In this chapter, we describe a simple four-processor, bus-connected ring system which has been implemented in two different simulators. One implementation was completed in the previous version of DEPEND based on the CSIM process-interaction simulator. The other was hand-translated using the techniques developed in this thesis. Figure 7.1 illustrates the system being modeled.

Every second, each CPU sends an "I'm alive" message to the CPU to its right. Faults are injected into the CPUs, according to a Weibull distribution, causing them to fail. Faults are also injected at an exponential rate into the interconnecting bus causing messages to be lost. The system fails if a processor fails to receive an "I'm alive" message from its left neighbor for at least 2.1 seconds. Note that this can be either due to a CPU failure or multiple link failures.

The model divides into four processes as follows:

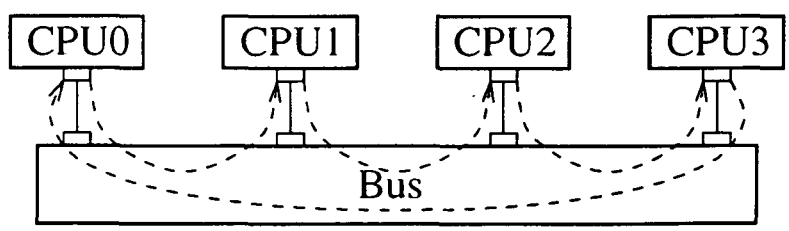

Figure 7.1: Four processor, bus-connected ring

Table 7.1: Simulation results for the four-processor ring system

Hybrid technique	Process-interaction	Speedup
7.06	227.8	32.27

1. A heartbeat sending process for each CPU which sends an "I'm alive" message to its right neighbor once a second.

2. A heartbeat monitoring process for each CPU which waits for up to 2.1 seconds to receive an "I'm alive" message from its left neighbor. If the monitoring process times out, it shuts the CPU down.

3. A fault injection process for each CPU, injecting faults at a rate set by the Weibull distribution with parameters $a = 10^5$ and $b = 2.0$, where the cumulative distribution function for the Weibull distribution is as given below.

$$1 - \exp^{-(x/a)^b}$$

4. A fault injection process for the bus injecting faults at a rate set by the Exponential distribution with mean $a = 10^4$, where the cumulative distribution function for the exponential distribution is as given below.

$$1 - \exp^{-x/a}$$

This example was implemented with the CSIM-based version of DEPEND and on the new simulator. Each simulation was timed and the results are given in Table 7.1.

8. CONCLUSIONS

The new compiler-based techniques including our hybrid process-interaction/event-driven simulation strategy provide improved performance for existing models while still using the straightforward process-based modeling. The case study demonstrated that a speedup of 32 times can be obtained for system dependability studies. We believe that with continued development, these techniques can lead to significant run-time performance gains on realistic performance and dependability models. As multiprocessing machines become more commonplace, we expect to exploit parallelism to further reduce the simulation run-time.

REFERENCES

[1] V. Jha and R. L. Bagrodia, "Transparent implementation of conservative algorithms in parallel simulation languages," in *Proceedings of the 1993 Winter Simulation Conference* (G. W. Evans, M. Mollaghasemi, E. C. Russell, and W. E. Biles, Eds.), pp. 677–686, IEEE, 1993.

[2] R. Bagrodia, K. M. Chandy, and W.-T. Liao, "A unifying framework for distributed simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 348–385, Oct. 1991.

[3] R. L. Bagrodia and W.-T. Liao, "Transparent optimizations of overheads in optimistic simulations," in *Proceedings of the 1992 Winter Simulation Conference* (J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, Eds.), pp. 637–645, IEEE, 1992.

[4] V. F. Nicola, M. K. Nakayama, P. Heidelberger, and A. Goyal, "Fast simulation of dependability models with general failure, repair and maintenance processes," in *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, June 1990.

[5] P. Shahabuddin, V. F. Nicola, P. Heidelberger, A. Goyal, and P. W. Glynn, "Variance reduction in mean time to failure simulations," in *Proceedings of the 1988 Winter Simulation Conference*, pp. 491–498, 1988.

[6] K. K. Goswami, "Design for dependability: A simulation-based approach," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1993.

[7] R. Fujimoto and D. M. Nicol, "State of the art in parallel simulation," in *Proceedings of the 1992 Winter Simulation Conference* (J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, Eds.), pp. 246–254, IEEE, 1992.

[8] D. M. Nicol, "The automated partitioning of simulations for parallel execution," Ph.D. dissertation, University of Virginia, Aug. 1985.

[9] K. K. Goswami and R. K. Iyer, "DEPEND: A design environment for prediction and evaluation of system dependability," in *Proceedings of the 9th Digital Avionics Systems Conference*, pp. 87–92, Oct. 1990.

[10] K. K. Goswami and R. K. Iyer, "DEPEND: A simulation-based environment for system level dependability analysis," Tech. Rep. CRHC Report #92-11, University of Illinois – Center for Reliable and High-Performance Computing, June 1992.

[11] K. K. Goswami and R. K. Iyer, *The DEPEND Reference Manual.* University of Illinois – Center for Reliable and High-Performance Computing, Urbana, Illinois 61801, Oct. 1990.

[12] B. P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation.* London, UK: Academic Press, 1984.

[13] B. P. Zeigler, "System theoretic foundations of modelling and simulation," in *Simulation and Model-Based Methodologies: An Integrative View, Proceedings of the NATO Advanced Study Institute held at Ottawa, Ontario/Canada, July 26-August 6, 1982* (T. I. Oren, B. P. Zeigler, and M. S. Elzas, Eds.), pp. 91–118, Berlin, Germany: Springer-Verlag, 1984.

[14] G. S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation.* New York, NY: John Wiley & Sons, 1973.

[15] J. B. Evans, *Structures of Discrete Event Simulation: An Introduction to the Engagement Strategy.* New York, NY: Halstead Press, 1988.

[16] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* New York, NY: John Wiley & Sons, 1991.

[17] H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology.* Reading, MA: Addison-Wesley, 1978.

[18] G. S. Fishman, *Principles of Discrete Event Simulation.* New York, NY: John Wiley & Sons, 1978.

[19] J. Eichenauer-Herrmann and H. Grothe, "A new inversive congruential pseudorandom number generator with power of two modulus," *ACM Transactions on Modeling and Computer Simulation*, vol. 2, pp. 1–11, Jan. 1992.

[20] H. Niederreiter, "New methods for pseudorandom number and pseudorandom vector generation," in *Proceedings of the 1992 Winter Simulation Conference* (J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, Eds.), pp. 264–269, IEEE, 1992.

[21] W. N. Graham, "A comparison of four pseudo random number generators implemented in Ada," Tech. Rep., Applied Research Laboratories, University of Texas at Austin, Oct. 1991.

[22] J. A. Fisher, "Object oriented random number generators," *Computers and Industrial Engineering*, vol. 25, pp. 561–563, Sept. 1993.

[23] G. S. Fishman and L. S. Moore, III, "An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$," *SIAM Journal of Scientific Computation*, vol. 7, pp. 24–45, Jan. 1986.

[24] S. Tezuka and P. L'Ecuyer, "Efficient and portable combined Tausworthe random number generators," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 99–112, Apr. 1991.

[25] H. Schwetman, "CSIM reference manual (revision 16)," Tech. Rep., Microelectronics and Computer Technology Corporation, June 1992.

[26] R. L. Bagrodia and W.-T. Liao, "Maisie user manual," Tech. Rep., Computer Science Department, University of California at Los Angeles, June 1993.

[27] R. L. Bagrodia and W.-T. Liao, "Maisie: A language for the design of efficient discrete-event simuluations," Tech. Rep., Computer Science Department, University of California at Los Angeles, 1992.

[28] K. K. Goswami and R. K. Iyer, "Simulation of Software Behavior Under Hardware Faults," in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993.