Implementing Parallelism in Random Discrete Event-Driven Simulation

Marc Bumble and Lee Coraor

The Pennsylvania State University, University Park PA 16801

Abstract. The inherently sequential nature of random discrete event-driven simulation has made parallel and distributed processing difficult. This paper presents a method of applying Reconfigurable Logic to gain some parallelism in event generation. Field Programmable Gate Arrays (FPGAs) are used to create a flexible and fast environment in which events may be generated according to various statistical models. The method presented accelerates both event generation and the elimination of some blocked events from the Event Queue.

1 Introduction

Using reconfigurable logic, this study explores a method of accelerating random discrete event-driven simulation. Random discrete event-driven simulation is inherently serial in nature due to its *causality* [8] constraints. In 1987, Reed et al reported that the parallel implementation [of simulations] rarely completes more quickly than the sequential implementation [when distributed simulations are run with a central server network] [9]. However, the benefits of faster simulation execution make parallelism a very attractive pursuit. Faster simulations could benefit traffic engineers in predicting and accommodating emergency changes in metropolitan traffic models. Analogous applications exist for simulating aerospace traffic and telephone networks.

Random discrete event-driven simulation is divided into an event generation phase and an event execution phase. This study uses a simple software simulation model to determine approximate durations for the two phases. For this model, event generation requires approximately 20% of the total simulation time, and event execution requires approximately 80%. Although the study concentrates on speeding up the event generation phase, the work is the first stage toward overall simulation speedup. The results presented here are *preliminary* and justify future work. The speedup of the event execution phase and of communications between multiple event processors is reserved for future studies.

The basic random event-driven simulation model is illustrated in figure 1A. The *Event Generator* creates random events, according to a user selected statistical distribution, and deposits the ordered events in the *Event List*. The Scheduler steps through the Event List in chronological order according to the global *Simulation Time Clock*, attempting to allocate resources to each event. If the resources are available, the event can execute. If not, the event is blocked.

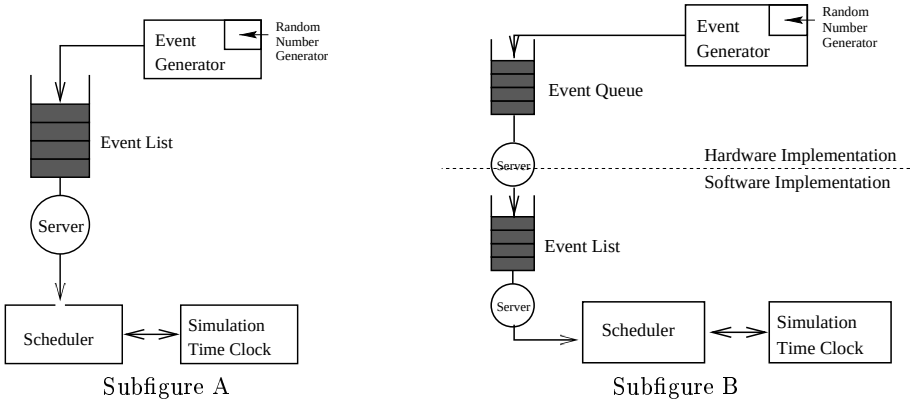


Fig. 1. The Basic and Modified Simulator Models

1.1 Methodology

Simulation speedup is accomplished via two independent enhancements. The first accelerates random event generation. The second facilitates faster detection and elimination of blocked events.

In the first enhancement, event generation speedup is accomplished by translating some simulation loop software into combinations of parallel, systolic, and reconfigurable logic. Reconfigurable logic allows the user to compile various statistical distribution models into hardware inexpensively. Existing hardware is reused instead of requiring a variety of Application Specific Integrated Chips (ASICs). Reconfigurable logic is *required* by this approach, as it is impossible to anticipate every statistical model which a user might desire. The reconfigurable logic can accommodate users by allowing the implementation of table generated statistical distributions.

In typical simulations, data dependency exists during event generation because event arrivals are calculated as random offsets from the previous event's arrival time. Service durations are calculated as random offsets from the current event's arrival time. The acceleration of event generation is possible because the random arrival and service offsets are not themselves dependent on anything. Data dependency among events arises when the random offsets are added to the previous event's arrival time. The Event Generator computes event arrival times, service times, and resource requirements with some partial parallelism. The resulting event objects are stored in the Event Queue which is accessible to the scheduling software. The Event Queue can further process and eliminate some events which are blocked due to unavailable resources, relieving the Scheduler (see figure 1A) of handling some impotent events.

The second simulation enhancement uses a *content addressable memory* queue. The queue stores data and also performs some minor processing, such as some

elimination of blocked events. Content addressable memory allows independent event filtering, freeing the processor to concentrate on event scheduling.

Arrival events, which are produced in time order by the event generator, are are added directly into the Event Queue. When these arrival events are executed, they produce *end of service* events. Executed end of service events may release resources for later reuse. It may seem that the generated end of service events could be scheduled quite far ahead of the currently serviced arrival event. However, the arrival and service rates are often determined such that the events are serviced faster than they arrive. Otherwise, the Event Queue fills with end of service events which tie up resources causing many of the new arrival events to be blocked. Assuming that service rates exceed arrival rates allows other Event Queue enhancements. If the service rates are faster than the arrival rates, the Event Queue will maintain a reasonable size. The queue may then be stored entirely in content addressable memory permitting faster searches for the earliest time stamped event.

Before the simulation, the user selects statistical distributions for generating event arrival and service times. The simulation model is then configured and downloaded by the host platform. The initial simulator configuration will remain for the duration of the simulation.

1.2 Sequential Random Discrete Event-Driven Simulations

Event-driven simulations typically have three [4] basic common denominators. First, they contain a set of *state variables* which denote the state of the simulation model. The state variables contain information such as the number and availability of system resources. Secondly, a typical random discrete event-driven simulation contains an *event list*, depicted in figure 1A. The event list contains pending events which were created by an event generator but not yet executed by the scheduler. These events require system resources to execute. The resources' availability are described by the state variables. Events often contain an arrival time-stamp and possibly a duration. The arrival time-stamp indicates when the event impacts the system's state variables. Event arrival times and service times are frequently generated based on statistical models. For example, events may arrive according to a Poisson Distribution. Finally, the third common denominator of random discrete event-driven simulations is the *global simulation clock* which keeps track of the simulation's progress.

A sequential random discrete event-driven simulation model jumps from one event to the next skipping the dead time between the events. Thus the model environment changes state at discrete points in time when the events occur. Event-driven simulations are often used to model traffic on telephone networks or roads. The modeled networks often contain *source* nodes where traffic emerges in the model. Examples of source nodes might be a tunnel or bridge entrance. The network source node might also be a boundary or edge point. Traffic can exit a model at a *sink* nodes. Sink node examples for a traffic simulation are analogous to the source node examples. As the event moves through the network of traffic, the event might require various resources. For instance, upon exiting

the network, the event might be required to wait in a queue before being granted entrance to an exit tunnel. A city block, which a vehicle needs to traverse, might also be considered a required resource which is occasionally blocked by delivery trucks, block parties, or accidents.

Software event-driven simulations often execute several loops. One loop may generate events which arrive based on a specific statistical frequency. The loop repeatedly creates the next simulation event based on an offset from the previous event's start time. A second simulation loop [4] generally removes the event with the smallest time-stamp from the event list. Each removed event is processed by making appropriate changes to the simulation model's state variables.

2 Event-Driven Simulation Software Model

To gather timing information on general characteristics of software event-driven simulations, a simple software model was created and its performance was measured. The software model and its timing characteristics are described in the following subsections.

2.1 The Software

Software simulation is divided into two functional blocks. The first block of code is the event generation block. The generation block created a priority queue of randomly generated events. The second software block is the event execution block which continually removes the event with the smallest timestamp from the event queue. The events for the simulation are generated according to time segments. So, for example, if the simulation was divided into 30 second time segments, the event arrivals would first be generated for that 30 second timeline. Then that timeline of events would be executed, scheduling new service events as well as keeping track of the number of events which were blocked due to unavailable resources. Towards the end of the timeline's execution, events which need to be scheduled beyond the length of the current timeline are carried over to the front of the next timeline segment.

2.2 The Event Generation Loop

The event generation loop of code creates events which are composed of random arrival and service times as well as two random resource values, **a** and **b**. The random number generator is composed of the ACG Class from the LibG++ standard library template class. The ACG Class is a variant of a Linear Congruential Generator (Algorithm M) described in [5]. The numeric result is permuted with a Fibonacci Additive Congruential Generator (ACG) to get good independence between samples. The results from the ACG class are then converted to the Poisson distribution by an application of formulas obtained from [10] or by the application of additional classes from the LibG++ standard library. A priority

queue, with a running time of *O(n*Log(n))*, is used to pop the earliest event from the queue. All events have the same priority.

The main event generation loop code is provided in table 1A. Events are generated in time segments. RandomDist1() generated random numbers derived from one of seven possible distributions: Poisson, Uniform, Erlang, Geometric, Normal, Lognormal, or the Weibull distribution. In table 1A, RandomDist2() generates random service times from the Uniform Distribution. Each event contained 2 randomly generated, 4-bit resources. Once instantiated, events are enqueued on the Event Queue.

```
while (clock <= time_segment) {
  create_event_cnt++;
  clock += RandomDist1();
  arrival_time = clock;
  service_time =  RandomDist2();
  Event_Class *event =
     new Event_Class(arrival_time,
                     service_time);
  queue->enq(*event);
};
```

Subtable A

```
while (!queue->empty()) {
  Event_Class event = queue->deq();
  if (event.Time() > time_segment) {
    queue->enq(event);
    break;
  }
  if (event.isArrival() == true) {
    if ((event.res.get_a() <=
            a_resource_count) &&
        (event.res.get_b() <=
            b_resource_count)) {
      a_resource_count -= event.res.get_a();
      b_resource_count -= event.res.get_b();
      event.SetEndOfServiceTime();
      queue->enq(event);
    } else {
      if (event.res.get_a() >=
            a_resource_count)
        block_a++;
      if (event.res.get_b() >=
            b_resource_count)
        block_b++;
    }
  } else {
    // Event was an End of Service event
    a_resource_count += event.res.get_a();
    b_resource_count += event.res.get_b();
  }
};
```

Subtable B

Table 1. Event Generation and Execution Loop Code

2.3 The Event Execution Loop

In the event processing loop, events are removed from the event queue in time order, starting with the smallest time stamped event. If the selected event is an arrival event, and its required resources are available, the resources are assigned and the resource counters are decremented. Next, an end of service event is scheduled. If the scheduler encounters an end of service event, the appropriate counters are incremented. If the resources required by an arrival event are unavailable, the event is considered to be blocked, and the appropriate block counter for the particular resource is incremented. The event execution loop

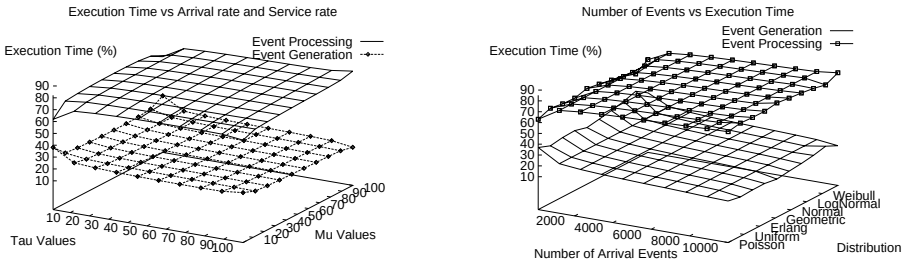
code is provided in table 1B. In table 1B, the blocked event counters, block_a and block_b, are used to generate simulation statistics in subsequent code.

2.4 Software Implementation

The test simulation was written in C++ and compiled with the GNU g++ compiler, version 2.7.2 on a Sun Microsystems Ultra 1 Sparcstation. The event priority queue was an instantiation of the GNU standard library XPPQ, which implements a 2-ary heap via XPlexes. The random number generator used was the ACG class from the LibG++ library. More information about the GNU LibG++ classes can be found in their relevant INFO pages, available with the LibG++ library.

2.5 Software Event-Driven Simulation Performance

The event-driven simulation software performance demonstrated that event generation may be expected to require a significant amount of computation time. In the tests performed, event generation always required more than 10% of the total simulation execution time. The test simulations typically required 20% or more of their total processing time for event generation. In figure 2A, the software simulation execution time is split into Event Generation and Event Processing. The simulations were run varying the event arrival, τ, and service, μ, rates. Figure 2A illustrates that the Event Generation time remains relatively level at 20% of the total event simulation time for the plotted ranges of τ and μ. Additional work illustrated that Event Generation time asymptotically approaches 20% as the number of events is increased. Figure 2B illustrates that varying the distributions does not change the simulation event generation/processing execution time ratio.



Subfigure A Subfigure B

Fig. 2. Event Generation verse Event Execution Time

3 The Acceleration Mechanism

The enhanced hardware implementation of the accelerator consists of two parts. The Event Generator mechanism is illustrated in figure 3A [2]. The Event Generator creates random events using a pipelined systolic array [6]. Statistical distributions implemented in hardware determine event start and finish times. This work assumes that uniform random numbers are generated in hardware [7]. The resulting random values are then used to generate the event start and finish times. The second part of the Acceleration Mechanism splits the Event List of figure 1A into two queues found in figure 1B, the Event Queue and the Event List. The Event Queue consists of a time-ordered list of arrival events. The Queue is a hardware mechanism. It receives its events in sorted order from the event generator. In addition to serving as a FIFO buffer, the Event Queue removes and records events which are blocked due to insufficient resources. The Event List, which is a software mechanism, holds the service events, whose arrival order and time dispersal are not guaranteed.

3.1 The Event Generator

Event Generation, illustrated in figure 3A, is subdivided into the creation of arrival and service times. Event generation is accomplished by a two-dimensional reconfigurable systolic array which allows the two time offsets to be created in parallel. *Reconfigurable logic* [3] boosts the execution speed of event generation by avoiding much of the communications overhead required by parallel processors. The systolic array depicted in figure 3A, pumps data from one processing block to the next at regular intervals, until the data circulates to the Event Queue. Reconfigurable logic was configured to implement adders, multipliers, and a natural logarithm, $ln()$, functional unit. These functional units are processing elements within the systolic array. The array calculates event arrivals and durations based on the Poisson Distribution. [1][10]

The systolic array [6] is implemented in reconfigurable logic, which consists of a large collection of flexibly connected elementary logic units. Reconfigurable logic allows the same hardware to implement various statistical distributions according to user preference. Field Programmable Gate Arrays (FPGAs), the reconfigurable logic, can be reconfigured in place, meaning that the chip need not be removed to be reprogrammed.

3.2 The Event Queue

The second phase of this work consists of the Event Queue. The Event Queue buffers the output of the Event Generator depicted in figures 1A and 1B. The enhanced Event Queue also performs some filtering of the events it holds. The Event Queue compares the number of resources which the lead events require versus the number of resources available. If the next event, or events in the queue require more resources than are available, the Event Queue removes the impotent event/events and updates its block count.

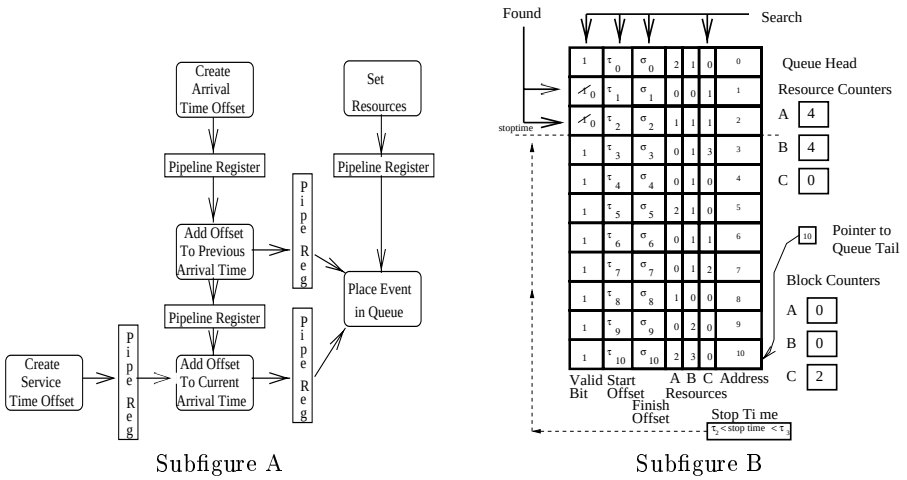


Fig. 3. The Acceleration Mechanisms

The Event Queue is composed of *Content Addressable Memory* (CAM) cells. Each bit of a word consists of one CAM cell. When conventional random access memory is used, it is often necessary to specify the physical address of a particular object in the memory. To search for a particular value in the memory, conventional memory requires that all entries be scanned sequentially. Then each value in a given memory address is compared with the value sought. Content Addressable memory eliminates the sequential search by simultaneously examining all entries and selecting those which match the search criteria. In general, *content addressable memory* (CAM) permits any stored items to be accessed directly using the contents of the stored items as their addresses. The subfield chosen to address the memory is referred to as the *key*.

The event queue, illustrated in figure 3B, operates in one of three modes, *Load, Pop* or *Enable*. The *Load* mode allows new events to be entered at the head of the queue. The Event Generator loads the events which are composed of start and finish times as well as the number and type of resources the event requires. As events are stored in content addressable memory, the head pointer is incremented to point to the parallel CAM cells where the next event received from the Event Generator will be stored. In *Pop* mode, as events are popped off the event queue, the queue tail pointer is incremented to point to the memory location containing the next event to be removed from the queue. The processor also updates the queue's stoptime, which indicates when the first event in the Scheduler's Event List finishes, possibly releasing resources. The *Enable* mode allows the queue to remove some of the events which will be blocked. Event blockage can be predicted by knowing the current resource stock, the stoptime, and the number of resources required by events waiting in the queue. The Scheduler provides the Event Queue with the current number of resources, and the time of the next

finish event. The Event Queue can then determine if start events, which occur before the stoptime, will have access to enough resources to be viable. If resources for a start event are insufficient, the start event is blocked and recorded. The invalidated event is popped off the queue. The Scheduler can infer that the block occurred due to an increment on the appropriate block counter.

An example queue is illustrated in figure 3B. The events in this queue can require three possible resource types, A, B, and C. In figure 3B, the first arrival event, τ_0, requires 2 A resources, 1 B resource, and no C resources. Arrival (τ_x) events which required depleted resources can be eliminated in parallel from the queue. In the figure above, the C resource has been depleted. To eliminate impotent category C arrival events, a search is performed which looks for all category C arrival events occurring before the first service (σ_x) event stored in the Event List. The service time is stored in the *stop time* register. In this example, the stop time register contains a value between τ_2 and τ_3. The results of the search set the valid bits of the criteria matches to 0 eliminating events τ_1 and τ_2.

4 Performance

The initial results for speeding up just the event generation portion of discrete random event-driven simulation have been promising. In software, allocating memory and then generating Poisson arrival and service times requires approximately 44851 nanoseconds. The timing measurement was taken with calls to the SunOS gethrtime() function. The software written in C++ is illustrated in table 1A, where some additional processing is performed when the event data structure is allocated. The code execution times were clocked on a Sun Microsystems Ultra 1 Sparcstation running Solaris 5.5.1. The code was compiled using the g++ compiler, version 2.7.2.

When the Event Queue is added to the hardware system, the software modeled by the hardware would include an extra call to insert each event item into the FIFO queue of pending events. The extra call to insert the object in the linked list increases the run time to approximately 52293 nanoseconds.

The hardware version was modeled using Altera's Max+Plus II® FPGA simulation package. The design was created and simulated in the Altera Flex 10K series of FPGAs. The Flex 10K series has the following features. The devices contain 10,000 to 250,000 typical gates. Additional routing features on the chip facilitate predictable interconnect delays which provide reliable simulation results. Software design support and automatic place-and-route tools are provided by Altera's Max+Plus II® development system.

The design depicted in figure 3 was translated into AHDL, Altera's High Definition Language. The AHDL was compiled into three Altera 10K series FPGA chips. The Event Queue was simulated in one Altera EPF10K50BC356-3, the Event Generator in one EPF10K30RC240-3, and the Natural log function was implemented in one EPF10K50BC356-3. In simulation at a 200 nanosecond clock

rate, the event generator hardware version requires 200 nanoseconds, producing one event per clock cycle. Therefore, we achieve a speedup of 225.

The Event Queue was also modeled using Max+Plus II® in AHDL. The Event Queue is capable of loading and popping one event per 200 nanosecond clock cycle. In the enable mode, the Queue processes a select subset of blocked events at the rate of one event per clock cycle.

5 Future Directions

Futures studies will examine the speedup of the event execution phase and of the communications required between multiple event processors. Hardware enhancements can be applied to maintain the Event List in time order. During the execution of the event lists, more random events are generated as new event arrivals schedule future end of service events. Both the Event Queue and the Event List may be combined and fully implemented as hardware. The hardware could then order the newly generated service events along with the existing arrival events and continually have the event with the smallest timestamp available for processing, avoiding costly software searches.

References

1. Jerry Banks, John S. Carson II, and Barry L. Nelson. *Discrete-Event System Simulation.* International Series in Industrial and Systems Engineering. Prentice Hall, Upper Saddle River, New Jersey 07458, second edition, 1996.
2. Marc Bumble and Lee Coraor. Introducing parallelism to event-driven simulation. In *Proceedings of the IASTED International Conference–Applied Simulation and Modelling, ASM '97, Banff, Canada, July 27-August 1, 1997.* The International Association of Science and Technology for Development, August 1997.
3. Bradly K. Fawcett. Taking advantage of reconfigurable logic. *Seventh Annual IEEE International ASIC Conference and Exhibit*, pages 227–230, Sept. 1994.
4. Richard M. Fujimoto. Parallel discrete event simulation. In *Communications of the ACM*, volume 33 no. 10, pages 30–53. ACM, October 1990.
5. Donald E. Knuth. *The art of computer programming.* Addison-Wesley, 1968.
6. H. T. Kung. Why systolic architectures. *IEEE Computer*, 15(1):37–46, January 1982.
7. Sean Monaghan. A gate-level reconfigurable monte carlo processor. *Journal of VLSI Signal Processing*, 6(2):139–153, August 1993.
8. David M. Nicol. Principles of conservative parallel simulation. In J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, editors, *Proceedings of the 1996 Winter Simulation Conference*, pages 128–135, 1996.
9. Daniel A. Reed, Allen D. Molony, and Bradley D McCredie. Parallel discrete event simulation: A shared memory approach. *Proc of the 1987 ACM SIGMETRICS Conf on Meas and Model of Comput Syst*, 15(1):36–38, May 1987.
10. Jean Walrand. *Communication Networks: A First Course.* Aksen Associates, Inc., 1991.