Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

ARCHITECTURE FOR A NON-DETERMINISTIC SIMULATION MACHINE

Marc Bumble
Lee Coraor

Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16801, U.S.A.

ABSTRACT

Causality constraints of random discrete simulation make parallel and distributed processing difficult. Methods of applying reconfigurable logic to implement and accelerate simulation service event queues are presented which process simulation events at a rate of one event per 80 nanoseconds. The event generator presented in our previous work (Bumble and Coraor 1998) is also capable of sustaining the 80ns clock rate, providing overall speedup rates which depend on the software comparison scenario. The software comparison cited in this work provides a 2 order of magnitude speedup. The speedup factor varies with the size of the software event queue. Field Programmable Gate Arrays (FPGAs) are used to implement and test the service queue design.

1 INTRODUCTION

Using reconfigurable logic, this study explores a method of accelerating random discrete simulation. Random discrete simulation is inherently serial in nature due to its *causality* (Nicol 1996) constraints. However, the benefits of faster simulation execution make parallelism a very attractive pursuit. Faster simulations could benefit traffic engineers in accommodating emergency changes to metropolitan traffic models. Analogous applications exist for simulating aerospace traffic and telephone networks.

The basic random simulation model is illustrated in Figure 1. The model applies to both time and event-driven simulation. The simulator is divided into the Event Generator, the Event Queue, the Scheduler, and the Simulation Time Clock. The Event Generator creates random events, according to a user selected statistical distribution. Events and their attributes are placed in the *Event Queue*. The Scheduler steps through the Event Queue 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.

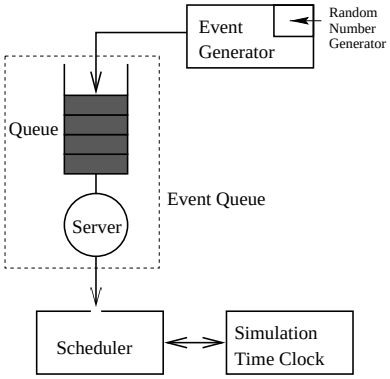

Figure 1: Simulator Model

One of the goals of discrete simulation research is to speedup simulations by processing events in parallel. Causality constraints force each event to execute in the simulation environment according to the results of the event's predecessor. An event which alters state information upon which the next event depends, must be executed before the dependent event. Simulations are characterized as being either *conservative* or *optimistic*. The conservative approach prohibits simulations from executing the next event until all events with earlier time-stamps have been executed. Optimistic simulators allow events to execute out of order, but maintain enough state information so that the simulation can return (*rollback*) to an earlier checkpoint. In optimistic simulation, *straggler* events with earlier timestamps create causality errors forcing rollbacks. The simulation must then undo the effects of any incorrect computations (Nicol 1996) resetting the simulation state variables and time clock to their checkpointed values.

The simulator presented in this study is a conservative simulator.

1.1 Methodology

Simulation speedup is accomplished via two independent enhancements. The first accelerates random event generation. The second facilitates faster handling of service events generated by scheduled arrival events.

The first enhancement accomplishes event generation speedup by translating some simulation loop software into parallel, systolic, and reconfigurable logic. Reconfigurable logic permits various statistical distribution models to be compiled from software into hardware implementations. Existing hardware may be 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 further accommodate user preferences by allowing the implementation of table generated statistical distributions.

Typical simulations contain data dependency between contiguous event arrivals and between event arrivals and their service durations. Each arrival is calculated as a random offset from the previous arrival time. Event service durations are calculated as random offsets from that event's arrival time. Because the random arrival and service offsets are not themselves dependent on anything, event generation can be accelerated. 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. By carefully maintaining the flow of data through hardware which calculates the sub-portions of each event, the proposed design alleviates some of the data dependency constraints. The resulting event objects are stored in the Event Queue which is accessible to the scheduling software.

The second simulation enhancement employs a split event queue. The queue stores the events in two separate queues, the *arrival* and *service* queues. The split queues are implemented as hardware, with the service queue able to select its minimum timestamp. A further enhancement allows the arrival queue to eliminate *impotent* events. An arrival event is considered to be impotent if the event requires unavailable resources, and no service event will occur in time to replenish the needed resources. The event can be eliminated without scheduler intervention.

In the general simulation model, arrival events, which are produced in time order by the event generator, 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. For the hardware simulator model presented in Section 4.1, it is assumed that before the simulation starts, the user selects statistical distributions for generating event arrival and service times. The simulation model is then configured and downloaded into reconfigurable logic by the host platform. The initial simulator configuration will remain for the duration of the simulation.

Computer architecture has generally used a *central processing unit* (CPU) as the focal point of its computation. Data was moved from memory or peripherals to the CPU where the computation was performed. Then the results were written back to a peripheral or memory. Currently, there is a trend to eliminate the central processing unit and move much of the processing hardware into memory and the peripherals (Gray 1998). With the current trend, computation is performed within a peripheral and moved to another point where additional computation may be performed, but basically the computation is performed at the communication endpoints. At these endpoints, instructions are fetched and decoded to perform work on the given operands or data. *Systolic Communications* (Kung 1988), on the other hand, allows the required processing to occur as data flows from one location to the next. Although proposed some time ago, systolic communications is difficult to implement, but with the advent of reconfigurable logic, this approach is very attractive for discrete event simulation. In systolic communications, the data flows through functional units so that computation is performed as part of the data transfer, giving rise to greater computational efficiencies. The process requires a fair amount of data independence and happens to work well with non-deterministic simulation models.

2 RELATED WORK

Historically, *deterministic* logic simulation has been the chief application for simulation accelerators. The first accepted logic simulation machine was the Boeing Computer Simulator (VanAusdal 1971). Both Abramovici (Abramovici et. al 1983) and Barto (Barto and Szygenda 1985) developed special purpose parallel processing architectures for handling logic simulation. However, machine development costs, limited applications, and the fast evolution of technology have deterred hardware research for parallel discrete simulation in the past. The development of Field Programmable Gate Arrays (FPGAs) (Brown et al. 1992) (Fawcett 1994) has significantly reduced hardware implementation costs. Standardized description languages, such as Verilog and VHDL, facilitate the transfer of existing designs to new evolutions of hardware allowing systems to more easily keep pace with their quickly changing underling technology.

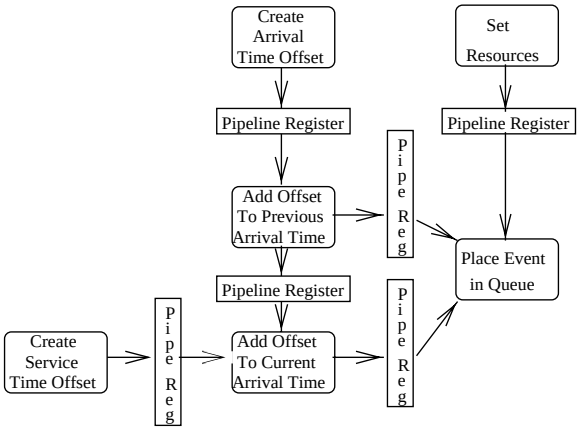

Figure 2: The Event Generator Flow Diagram

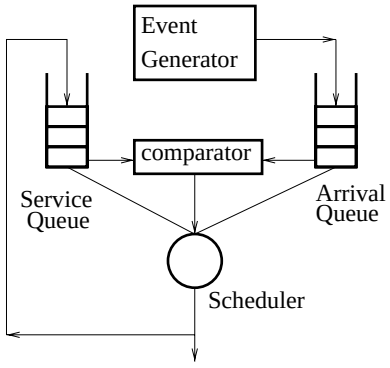

Figure 3: The Local Processing Element Design

Related studies of non-deterministic parallel event-driven simulation include work by Beaumont (Beaumont 1994). Reconfigurable logic was used to produce a communications ring which was capable of synthesizing application specific operators and control or communications circuits between parallel processing elements. Fujimoto et al developed the *Rollback Chip* (Fujimoto et al. 1992) which allows an optimistic simulator to quickly reset a simulation's state back to a previous checkpointed time.

2.1 The Event Generator

Event Generation, illustrated in Figure 2, 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* (Fawcett 1994) boosts the execution speed of event generation by avoiding much of the communications overhead required by parallel processors. *Systolic communications*, which facilitates a *dataflow* (Hennessy and Patterson 1990) model of computation, also accelerates the mechanism by avoiding the need to fetch instructions and operands from memory. Systolic communications provide the ability to transfer long streams of intermediate data between processes at high throughput rates with low latency (Hord 1993). The systolic array depicted in Figure 2, pumps data from one processing block to the next in regular time intervals, until the data circulates to the Event Queue. Reconfigurable logic implements adders, multipliers, and a natural logarithm, $ln()$, functional unit. These functional units are processing elements within the systolic array.

Output from the event generator is placed directly into the arrival queue illustrated in Figure 3. The local processing element design uses two queues for each server. The arrival queue holds the sorted list of arrival events, which arrive in-order from the Event Generator, and the queue can be implemented as a FIFO queue. Service events, which are created from processing successful arrival events, are stored in the service queue. Two methods of maintaining the service queue are discussed in Section 3. The sorter array mechanism is the most appropriate method for this application. In the processing element of Figure 3, a comparator samples the heads of both queues and indicates where the next minimum local time-stamped event resides.

3 THE QUEUE MECHANISM

After events are created by the event generator, they are stored in the arrival queue in order. The arrival queue can be easily implemented as a FIFO queue. Successfully executed arrival events create service events. However, the service events are generated out of order. The smallest timestamped events, be they service or arrival, must be continually available to allow the events to be executed in time order. Therefore, Section 3 develops a sorter mechanism to order a list of service events. This section presents two service queue alternatives. The first method maintains a sorted queue and can select the n^{th} element in O(1), but requires O(4) steps to insert a new element. The second method inserts new elements in O(1) and can pop the smallest element off in O(1) but does not maintain a sorted queue.

3.1 The Service Event Sorter

The first method, the Service Event Sorter, can sort events in 4 cycles, significantly faster than standard software sorts. This sorting mechanism maintains a sorted array facilitating selection of the kth smallest element. The hardware consists of the *input register*, a *content register array*, a *marked array*, and a *maxbit register*. The input value is compared against the content array values and inserted in the correct position within the content array. Auxiliary hardware registers and logic are used to quickly

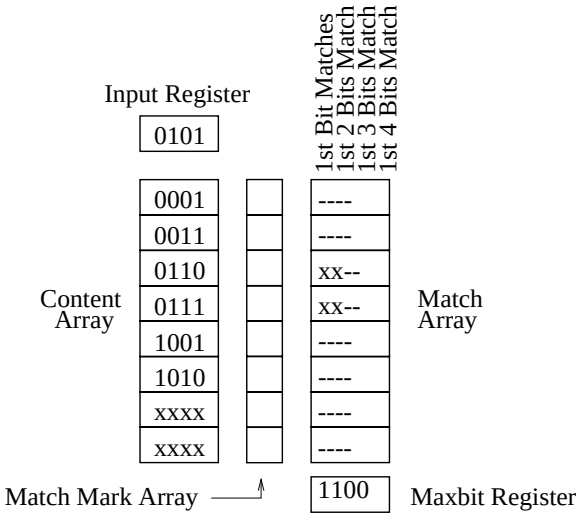

Figure 4: Service Event Sorter: Cycle 1

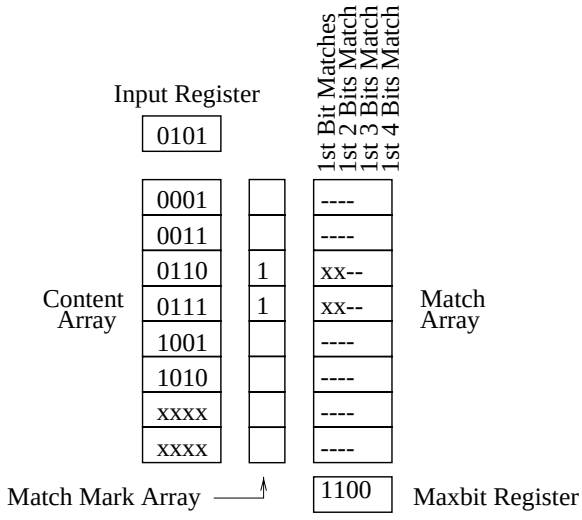

Figure 5: Service Event Sorter: Cycle 2

locate the correct insertion point for the new value. Longer queues can be created by chains of smaller queues.

In the first cycle, illustrated in Figure 4, all words in the content array are compared to the input register value. If any word in the content array has the same most significant bit (MSB) as the input register, the first bit of the maxbit register is set. If any content array word has the same top two MSB's as the input register, then the first two bits of the maxbit register are set. So if bit three of the maxbit register is set, it indicates that at least one word in the content array has the same top 3 MSB's as the input register. In the example depicted in Figures 4, 5, and 6, two words have the same top two MSB's as the input register value.

The proposed algorithm works as follows. All registers in the content array are compared with the input register. A network of nodes, called the match array, is used to determine the number of most significant bits which each content register has in common with the input data register. A single register, the *maxbit* register, records the result. For example, if one or more content array registers match the data input register on all 3 of the 3 most significant bits, then the first 3 bits of the maxbit register are set to logic 1's.

In the second of the four cycles, all words in the content array matching the input register with the maximum number of MSBs are marked by setting bits in a *marked* array. The marked array consists of one bit per word of the content array. Content array words which have the maximum number of MSBs matching the input register are marked as illustrated in Figure 5. These words will cluster due to the binary nature of the search.

During the third cycle, the required words in the content array will be moved down to allow room for the insertion of the input register contents. If the least significant marked bit in the maxbit register is i, then the i+1 bit of the input register is checked. When the i+1 bit of the input register is a zero, all registers in the content array from the marked register to the end of the array are shifted down one register in a single clock cycle. Otherwise, if the i+1 bit of the input array is a one, then all registers below the marked registers are shifted down one position creating a spot for the new word to be inserted as shown in Figure 6. For the given example, the i+1 bit of the input register is a zero, so all words including and below the marked registers are shifted down. Finally, in the fourth step, the input register value can be inserted into the array in proper order.

3.2 The Linear Array

The second service queue mechanism consists of a *linear array* approach which is described in (Leighton 1992) and illustrated in Figure 7. However, instead of using the linear array to sort the values, the array will simply maintain fast access to the minimum timestamped event. All new simulation events are passed into the leftmost array element, the queue head, and when removed, the elements are also popped off the queue head. Each element of the queue contains two registers and a comparator. The larger of the two resident elements may be passed to the right, and the smaller of the two elements may be passed to the left. Therefore the smallest entry is always at the leftmost queue element. Comparators in each queue element and

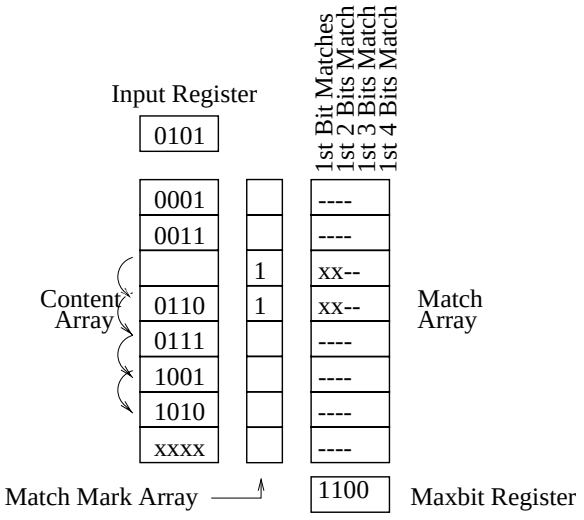

Figure 6: Service Event Sorter: Cycle 3

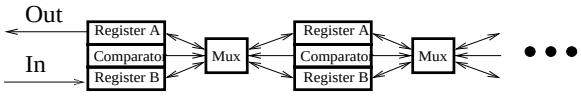

Figure 7: Linear Array Queue

the queue push/pop signal steer the 2x2 multiplexor logic to route the correct entries in and out of the processing element registers. Larger queue elements are passed to the right and smaller elements are passed to the left.

The service queue is required to always have the smallest element ready. The availability of the smallest element can be reasoned as follows. Assume that at some time, t, the queue contains N elements. Therefore, the leftmost element, K, has examined a sequence of N values, retaining the smallest value. This value can be popped off in 1 move. The element to K's right, K-1, has examined at least N-1 values, so the 2nd smallest value can be either at element K, or at element K-1, but it must be in one of those two places, and can be accessed in 2 moves since the smallest element must be removed first.

The n^{th} smallest element to enter the array is in any position from K down to $K-(n-1)$. Then the next smallest element to enter the queue will be in any position from K down to $K-((n-1)-1)$, which provides our inductive step for the $n-1$ smallest element. So our n^{th} smallest element can always be retrieved in n steps. This queue allows us to push and pop each element in O(1) time. The queue is illustrated in Figures 8 and 9.

Figure 8 illustrates a sequence of values being pushed into the array. The top array illustrates the first time step, with each successive array below depicting the same array during the next clock cycle. Comparators on each processing element and their associated multiplexors steer

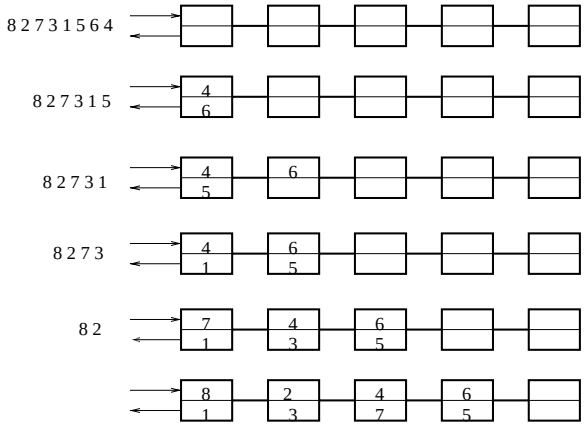

Figure 8: Linear Sort Array Input Example

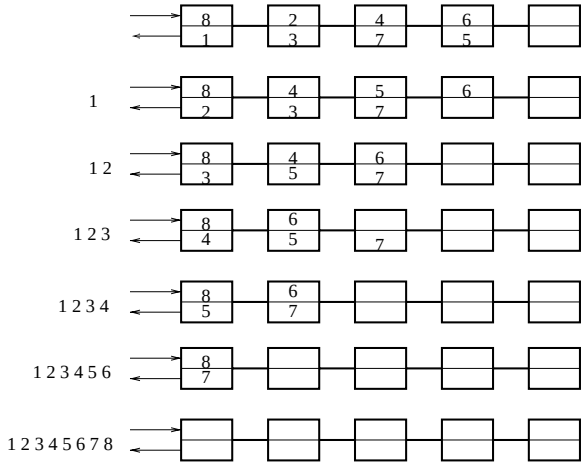

Figure 9: Linear Sort Array Output Example

the values into each element of the array. Larger elements are pushed to the right. When events are popped off the queue, the analogous sequence of steps is illustrated in Figure 9. Smaller elements are pushed to the left.

4 RESULTS AND CONCLUSIONS

Previous work (Bumble and Coraor 1997; 1998) demonstrated a speedup of 225 for an event generation hardware implementation of Figure 2 over its software analogue. The implemented array calculates event arrivals and durations based on the Exponential Distribution (Banks, Carson II, and Nelson 1996; Walrand 1991). Other logic arrangements could also be engendered, and the proposed system would have a variety of distributions available to the user.

This research focuses primarily on the service queue. The software version is implemented as a GNU *LIBG++* XPPQ Priority Queue class. In the C++ software simulation, the time required for the insertion and extraction

of events to and from the event queue increases as the queue strays from its optimum size. The proposed hardware queue speed, on the other hand, is not affected by its size, and it provides a 10^2 speedup over the software model.

4.1 The Hardware Model

The hardware service queue we implemented was a five element design closely resembling Figure 7. At this time, there is no need for the fully ordered list of Figure 5. The linear array queue was capable of pushing one 16-bit value per 40 nanoseconds. The smallest queue value could also be popped out at that rate. It is assumed that each simulator cycle would need to push one event and pop one event from the service queue. Therefore, our queue achieves the system 80ns cycle time. Queue data values would also require pointers to the event data so that pairs of values would need to be pushed and popped off the queue. Conversely, when new elements are pushed into a software data structure, the existing software elements must be fetched from memory to allow the CPU to compare the stored elements to the new arrival, so that the insertion point can be determined, or an address must be calculated to determine a proper bin on which to chain the new entry for hashing. Software methods require more time and variable amounts of it.

Using Altera'sMax+Plus II® FPGA simulation package, the Event Generator and the Service Queue have been simulated as individual parts running with a clock rate of 80ns. The service queue was simulated at a rate of 40ns, allowing it to push an event during the first half cycle and pop an event during the next. A 5 processing element queue was implemented on one Altera EPF10K20TC144-3 chip utilizing 90% of the chip's resources. The Event generator is synthesized as a combination of five chips. Four logarithm units are required, two producing their results on the even clock cycles and the other two producing results on the odd clock cycles. The *Create Arrival Time Offset* and the *Create Service Time Offset* blocks of Figure 2 each require one odd and one even logarithm unit. The system FPGA components are listed in Table 1.

One interesting facet gleamed from the FPGA research is that FPGA implementation methods are very important to minimizing their clock speeds. It is difficult for the hardware compilers to fully optimize designs. For instance, two methods of allowing a 16 bit D-flip/flop register to hold its value can be compared. One method routes the output back through an input multiplexor, so that the output is re-inserted on the next clock strike. The second method simply disables the register bits which then retain their value. The first method requires 16 lines to be routed efficiently within the FPGA. The second requires one signal to be chained to each element of the register bits.

Table 2: Event Generation

```
// intended to maintain number of arrival events
while (create_event_cnt <= num_events) {
  create_event_cnt++;
  arrival_time = clock + rnd1();
  service_time =  rnd2();
  Event_Class *test = new Event_Class(arrival_time,
                                      service_time);
  queue->enq(*test);
};
```

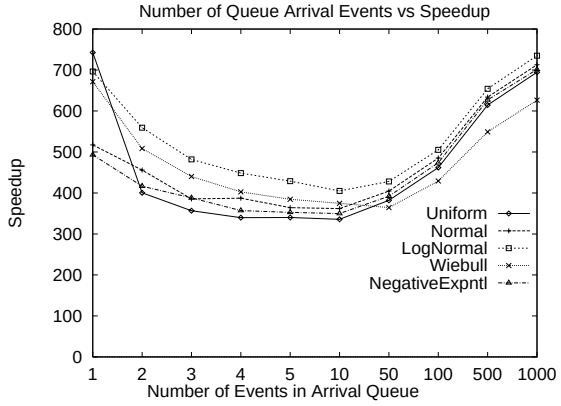

Figure 10: Speedup vs for Event Generation, Arrival and Service Queues

The second method was much more efficient producing better timing results.

4.2 The Software Model

The software simulation model we used for comparison was written in C++ and is illustrated in Tables 2 and 3. Some additional processing is performed when the event data structure is allocated. The arrival and service queues are maintained as a single heap data structure, unlike the proposed dual queue hardware mechanism illustrated in Figure 3. To gather accurate timing results, the number of events in the event queue was kept constant. The extra time used to generate additional arrival events in order to maintain the queue size is not included in the speedup plot of Figure 10. The software model used to obtain the timing values is provided in Tables 2 and 3.

Figure 10 illustrates the speedup expected for other distributions if the 80ns clock is maintained in their hardware implementations. The hardware distribution implemented provides the speedup illustrated for the NegativeExpntl curve in Figure 10. Excessive amounts of speedup were derived for queue sizes of 1 element probably due to the C++ destructor de-allocating the queue as its size drops below one. The speedup shown must also

Table 1: FPGA Chip Implementation

Function	Quantity	Chip Type	Chip Percent Utilized
Event Generator			
Logarithm Unit	4	EPF10K40RC208-3	95%
Event Logic	1	EPF10K30RC240-3	71%
Service Queue			
Linear Array	1	EPF10K20TC144-3	90%

Table 3: Event Execution Loop Code

```
while (events <= num_events) {
  events++;
  Event_Class event = queue->deq(); // included in
                                    // speedup test
  // took event off, so put it back to maintain num
  // arrival events
  arrival_time = clock + rnd1();
  service_time =  rnd2();
  Event_Class *arriv = new Event_Class(arrival_time,
                                       service_time);
  queue->enq(*arriv);  // not included in speedup test
  if (event.getArrival() == true) {
    if ((event.res.get_a() <= a_resource_counter) &&
        (event.res.get_b() <= b_resource_counter)) {
      a_resource_counter -= event.res.get_a();
      b_resource_counter -= event.res.get_b();
      event.SetNextArrivalTime();
      // push service event
      queue->enq(event); // included in speedup test
    } else {
      if (event.res.get_a() >= a_resource_counter) {
        block_a++;
      }
      if (event.res.get_b() >= b_resource_counter) {
        block_b++;
      }
      if (double_or_more_imp > 1)
        total_saves++;
      double_or_more_imp++;
    }
  } else {
    double_or_more_imp = 0;
    a_resource_counter += event.res.get_a();
    b_resource_counter += event.res.get_b();
  }
};
```

be considered in light of the fact that the Sparc Ultra's were attached to a Network File System (NFS) and so network communications delays within the building to the file server must be considered. However, once the file is initially fetched to the local machine, this delay ends. It also needs to be noted that the proposed linear array queue need not be implemented as reconfigurable logic. The queue could be implemented as an Application Specific Integrated Chip (ASIC), and would probably be able to function at an even faster clock rate with many more queue elements. The code execution times were clocked on a Sun Microsystems Ultra 1 Sparcstation running Solaris 5.6. The code was compiled using the GNU G++ compiler, version 2.7.2.

Future work will investigate joining individual processing elements into a simulation network. This course will directly effect the arrival queue which will need to handle unsequenced arrival events. Synchronizing a network of different local processing element clocks to follow a global simulation clock will also be challenging.

ACKNOWLEDGMENTS

The authors wish to thank Dr. Piotr Berman of Penn State University Department of Computer Science for his assistance.

REFERENCES

Miron Abramovici, Ytzhak H. Levendel, and Premachandran R. Menon. A logical simulation machine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-2(2):82–94, April 1983.

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.

R. Barto and S. A. Szygenda. A computer architecture for digital logic simulation. *Electronic Engineering*, 52(642):35–66, September 1985.

C. Beaumont, P. Boronat, and J. Champeau et al. Reconfigurable technology: An innovative solution for parallel discrete event simulation support. In *8th Workshop on Parallel and Distributed Simulation (PADS '94). Proceedings of the 1994 Workshop on Parallel and Distributed Simulation*, pages 160–163, Edinburgh, UK, July 1994. IEEE, SCS, San Diego, CA, USA.

Stephen D. Brown, Robert Francis, Jonathan Rose, and Zvonko Vranesic. *Field-programmable gate arrays*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1992.

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.

Marc Bumble and Lee Coraor. Implementing parallelism in random discrete event-driven simulation. In *Lecture Notes in Computer Science 1388, Parallel and Distributed Processing*, pages 418–427. IEEE Computer Society, Springer, March 1998.

Bradly K. Fawcett. Taking advantage of reconfigurable logic. *Seventh Annual IEEE International ASIC Conference and Exhibit*, pages 227–230, Sept. 1994.

Richard M. Fujimoto. Parallel discrete event simulation. In *Communications of the ACM*, volume 33 no. 10, pages 30–53. ACM, October 1990.

Richard M. Fujimoto, Jya-Jang Tsai, and Ganesh C. Gopalakrishman. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Transactions on Computers*, 41(1):68–82, January 1992.

Jim Gray. International parallel processing symposium keynote address, April 1998.

John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., first edition, 1990.

R. Micheal Hord. *Parallel Supercomputing in MIMD Architectures*. CRC Press, Inc., Boca Raton, Florida, 1993.

H. T. Kung. Systolic communications. *Internation Conference on Systolic Arrays*, pages 695–703, May 1988.

F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

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.

A. W. VanAusdal. Use of the boeing computer simulator for logic design confirmation and failure diagnostics programs. *Proceedings of the Advances in the Astronautical Sciences 17th Annual Meeting*, 29:573–594, June 1971.

Jean Walrand. *Communication Networks: A First Course*. Aksen Associates, Inc., 1991.

AUTHOR BIOGRAPHIES

MARC BUMBLE is a graduate student in the Computer Science and Engineering department at the Pennsylvania State University in University Park, PA. He received his B.S. and M.S. degrees in electrical engineering from the University of Pennsylvania in Philadelphia. His current research investigates architectures for accelerating non-deterministic simulation, including the application of reconfigurable logic.

LEE CORAOR is an Associate Professor of Computer Science and Engineering at the Pennsylvania State University in University Park, PA. He received his Ph.D. in Electrical Engineering from the University of Iowa. Dr. Coraor has worked on the design, implementation and performance evaluation of decoupled architectures and is currently investigating FPGA architectures and applications.