VLSI Design
1994, Vol. 1, No. 2, pp. 169–180
Reprints available directly from the publisher
Photocopying permitted by license only

Distributed and Parallel Demand Driven Logic Simulation Algorithms*

K. SUBRAMANIAN
ZYCAD Co., 1380 Willow Road, Menlo Park, California 94025

M. ZARGHAM
Computer Science Department, Southern Illinois University, Carbondale, Illinois 62901

(Received June 12, 1989, Revised February 20, 1990)

Based on the demand-driven approach, distributed and parallel simulation algorithms are proposed. Demand-driven simulation ties to minimize the number of component evaluations by restricting to only those component computations required for the watched output requests. For a specific output value request, the input line values that are required are requested to the respective component. The process continues until known signal values are needed (system input signal values). We present a distributed demand-driven algorithm with infinite memory requirement (but still the memory required at each process is no greater than the sequential demand-driven simulation), and a parallel demand-driven simulation with finite memory requirement. In our algorithms, each component is assigned a logical process.

The algorithms have been implemented on the Sequent Balance 8000 Multi-processor machine. Several sample circuits were simulated. The algorithms were compared with the distributed discrete-event simulation. Our distributed algorithm performed many times faster than the discrete-event simulation for cases when few results were needed. Parallel algorithm performed 2 to 4 times faster than the distributed discrete-event simulation.

Key Words: *Logic simulation, Parallel, Distributed, Demand driven, VLSI.*

With the advent of VLSI, the demand for faster digital logic simulators has increased. Conventional logic simulation requires enormous computing resources as the logic to be simulated increases in complexity. Hardware accelerators are popular and provide high performance. However, they are very expensive, not easily expandable, and cannot be used for other purposes. Therefore, software solutions are worthwhile because of their flexibility and reduced cost.

In general, there are two types of simulation - time-driven and discrete-event. In time-driven simulation, all the components of the system are evaluated every time step. Each component is scheduled at each time step even if the inputs to the component have not changed. In discrete-event simulation, the input changes are kept in an increasing time-ordered event queue. The simulator removes the head of the event queue and evaluates the changes in the output of the components and the time of change because of the event. It inserts these output changes into the event queue. This process continues until either the event queue is empty or some other stopping criterion is met. Thus, the component of the system is started or simulated only at those time steps where at least one input value changes. It has been experimentally shown that only a small percentage of a system's components are active every time step [1]. Therefore discrete-event simulation cuts down the number of evaluations at each time step when compared with time-driven simulation.

The inherent nature of event queue manipulation limits the potential parallelism of discrete-event simulation. Although techniques for performing event queue manipulation and event stimulation in parallel have been suggested, the parallelism achievable is limited.

Parallelism can be improved by removing the necessity of the global event queue and global clock. With the advent of distributed systems consisting of a number of autonomous sequential processors with communicating capabilities, several algorithms have been proposed by distributing the components to different processes. In [2], a component is simulated by a process and each process maintains its own virtual

*Based on "Distributed and Parallel Demand Driven Logic Simulation" by K. Subramanian and M.R. Zargham which appeared in Proceedings of 27th ACM/IEEE Design Automation Conference, June 1990, pp. 485–490, © 1990 IEEE.

clock. The clock value of a process is the time-stamp of the latest input event which is processed. A process with multiple input lines must wait until it receives inputs from all lines before it can select a message for processing. This way it can pick up the input event with the minimum time-stamp. Each process makes sure that no information received later can affect its output. This may require waiting on a set of input lines. A cycle of such waiting processes results in a deadlock.

Chandy and Misra have suggested two approaches to solve the deadlock problem. One is deadlock avoidance [2]. This uses additional null messages to avoid deadlock. Whenever an output event is not generated for any one of the outputs of a component, the component sends out null messages along all the output lines with a time-stamp equal to that of the event. Therefore, cycles of waiting processes will not occur. However, when the components of the system have high fan-outs, the ratio of null messages to real messages can be very high. The second approach is deadlock detection and recovery. The simulation runs until deadlock occurs. Deadlock is detected either by using deadlock detection controllers [3] or by using transmission markers around the system [4]. Recovery is done by advancing the clock of a process to a clock value so that it won't receive an input with a smaller time-stamp. Both deadlock avoidance and deadlock detection-recovery techniques have additional overheads. Their relative performance depends on the system being simulated [5].

Jefferson [6] has proposed an elegant algorithm known as Time Warp Mechanism. In this approach also each component is simulated by a separate process. A process responds as soon as it receives a message indicating an input change, disregarding the value of time stamp on the message. This is an optimistic approach. If a process receives an input value with time-stamp less than the time-stamps of any of the input value changes that have been already processed, the conflict is resolved by rolling back to a consistent state. In order to roll back, extra processing must take place during simulation. The cost of rolling back involves breakpointing the state of the system, performing the roll back, and the wasted evaluations of the events with time-stamps after the event that caused the roll back. It requires a lot of memory to store the states so that it can roll back to a previous state when needed. Roll back frequently depends on the particular instance of the problem. It is very difficult to analyze the performance of the roll back approach, but the algorithm requires no synchronization and has room for parallelism.

Smith et al., [7] have proposed a different approach for simulation known as demand-driven simulation. Unlike the above approaches, where events propagate through a circuit in response to input changes, demand-driven simulation requires the request for a value to propagate backwards both through the circuit and time. In time-driven and discrete-event simulation, all values are evaluated until the stopping criterion is reached, whether the intermediate results are needed for the watched signal evaluation during the watched intervals or not. However, in the demand-driven approach, only needed signal values are calculated. If the time intervals at which values needed are widespread and not all the pins are watched, the demand-driven approach has the potential to save an enormous amount of redundant computation. However, the demand-driven approach has a large memory requirement to store both the intermediate output values and the input values of a component during computation.

In this paper, we describe an approach for implementing demand-driven simulation in a distributed environment. In this approach, components are mapped onto processes. A process receives requests for the values of its output. Upon receipt of such a message, the process sends similar requests for input line values at appropriate times. A process receives value for a request if the line is connected to primary input lines, or if the value of the input line is already defined. Upon receiving all the inputs needed for a request a process evaluates the output value using the appropriate component function and then sends the value to the process that requested the value. A process which is connected to a watched output line receives requests for the watched line at the watched request times. A server process is used to dispatch these requests to the processes. The algorithm does not have any redundant component evaluations. Since the intermediate output values of the simulation have to be stored to avoid redundant computation, this algorithm requires, in the worst case, memory of the order of time of simulation (i.e., the maximum time of the watched requests). Further, memory is required to store the input values of a component to evaluate the output of the component at a specific time. This memory requirement is very high for sequential circuits. This happens because a component in a feedback loop is revisited many times to finally evaluate an output of one of the components of the feedback loop.

We also propose a parallel algorithm for logic simulation using the demand-driven approach. This heuristic algorithm does not require more memory than distributed discrete-event simulation. The algorithm has two phases. The first phase reduces the number of input events using the demand-driven approach. The

The second phase is a parallel version of distributed discrete-event simulation. In the following section we present our distributed algorithm. Our parallel demand-driven simulation algorithm is also presented. We provide the implementation details of distributed discrete-event simulation, distributed demand-driven simulation, and our parallel demand-driven simulation. We also compare the performance of these algorithms with concluding comments.

DISTRIBUTED DEMAND-DRIVEN LOGIC SIMULATION (DD)

In this algorithm, similar to other distributed algorithms, a process p_i is used for each component i of the digital circuit. If component i has an output connected to component j, there will be a duplex line l_{ij} between p_i and p_j. In a shared memory environment the duplex line l_{ij} is implemented as a shared data structure. An example of the mapping of a circuit to the computation model is given in Figure 1. Process i has a separate output line for each fanout line of the component's output pin.

During the course of execution, a process receives three kinds of messages from the lines.

1) INPUT - this message informs the value of a line at a specific time.
2) OUTPUT - this message is a request for a line value at a specific time.
3) TERMINATE - this message informs the process that no more requests will come along any of the output lines.

We make some assumptions about the circuit being simulated. These assumptions are also made by the sequential demand-driven simulation algorithm, and they are:

1) Each line has a defined initial value at time zero.
2) All feed-back loops have positive delay.
3) Each process has an infinite memory buffer at its site.
4) Component functions have the following properties. Each output of a component depends only on the values of the inputs at a specific time. Each input has a different time delay with each output of the component. That is, $O(t) = f(i_1(t - d_1), i_2(t - d_2), \ldots i_n(t - d_n))$ where: $O(t)$ is the output value of a component at time t; $i_1, i_2, \ldots, i_n$ are the inputs to the component, $d_1, d_2, \ldots, d_n$ are the fixed delay between output O and the inputs $i_1, i_2, \ldots, i_n$, respectively.

In the proposed algorithm, each process executes the pseudo-code shown in Figure 2.

The server process sends along the watched lines the requests at watched times to the corresponding processes connected to these lines and at the end, when it has received the result for all the requests, it sends TERMINATE message to all the other processes.

Figure 2 and the server process mentioned in the above paragraph forms the algorithm. Using the assumptions stated above, it can be trivially proved that the algorithm is correct and it does terminate. The server process is the one that realizes the end of simulation and sends the TERMINATE to all the component processes of the circuit. (The correctness of the algorithm is derived from the correctness of the sequential demand-driven simulation algorithm [7].) The OUTPUT request traces through processes and through time and reaches the defined system input values or initial values. If these values are correct, the successive values computed by processes should also be correct. As each output request evaluation is independent of the other requests, and an infinite buffer is assumed in all the process sites, there is no chance of deadlock occurring. Thus, each watched output line value request will be evaluated correctly. Once all the watched output line requests have been evaluated, each process will receive a TERMINATE message.

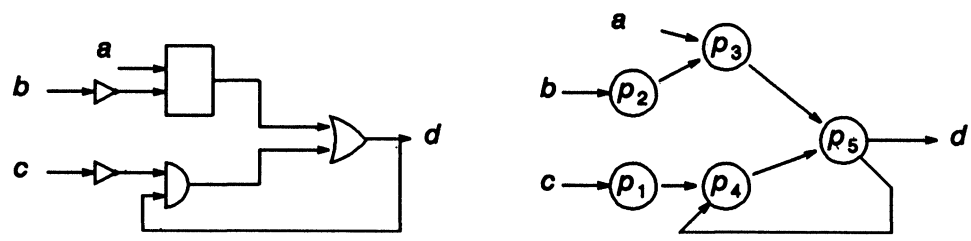

FIGURE 1 Mapping circuit to model.

```
PROCESS COMPONENT

begin:
set terminate to FALSE.
while ( not terminate)
    {
    while (message waiting)
        {
        pick up a message.
        case (message type)
            {
            INPUT:
                Find the output line request for which this message is intended.
                Store the value in the buffer of the output line request.
                If all the input values for the output line computation have arrived
                    {
                    Calculate the output value using the component function.
                    Store the output value for later requests.
                    Send INPUT message along the output line with this value.
                    }

            OUTPUT:
                If the value requested available at the site
                    send INPUT message along the output line
                    else
                    {
                    send OUTPUT message to all the input lines that are needed for the
                    evaluation of the output line. The time of request for each of the
                    input lines is t − d_i where t is the time of request of the output line
                    value and d_i is the delay between the output line and the input line
                    i.
                    Reserve a buffer to store the input value reports.
                    }
            TERMINATE:
                Set terminate to TRUE.
            }
        }
    }
end.
```

FIGURE 2 Process code for a component process.

Even though the above algorithm is concurrent and does not have any extra component evaluation, it requires a large amount of memory to store both the intermediate time values and the input values for a particular request. In the case of sequential circuits, if a component is in a feed-back loop, that component is revisited many times for a single watched output value. For each of these visits, a buffer is allocated to store the input values. If the time of simulation (i.e., the maximum time of any request) is high, this memory requirement is very high. In the next section, we provide a heuristic algorithm that does not require any more memory than distributed discrete-event simulation algorithm of [2].

PARALLEL DEMAND-DRIVEN SIMULATION (PD)

One of the factors which affects the computational time of discrete-event simulation is the number of input events. However, not all the input events are

needed for the evaluation of watched output line results. If the input events can be reduced ensuring that the simulation results are correct at least for the observation points, then computational time can be saved. The parallel demand-driven simulation uses the demand-driven approach in the first phase to reduce the number of input events. In addition, it also eliminates the components that are not needed for any of the output request evaluations. Phase I requires finite memory. Phase II of the algorithm is discrete-event simulation as described in [2, 8].

SIMULATION ALGORITHM

Phase I: Input events reduction

Similar to distributed demand-driven simulation, the components are mapped to processes. In addition, as shown in Figure 3, the algorithm uses two server processes, server 1 and server 2.

Server 1 marks the input events. Server 2 dispatches one message per watched line. The message is sent to the process which has this watched line as its output. A message has the following format.

1. time; The time at which the request sent by server 2 reaches the receiving process.
2. line; The watched line for which the message is being sent. This is the initial source line of this message.
3. max_time; The maximum time of request at the watched output line. This watched output line is the initial source of the message.

Server 2 sets the message time to the maximum watched request time at the watched line when it sends a message.

A component process, upon receipt of a message, sets a flag which indicates that the component needs to be simulated during the second phase. (The flags were reset at the start of simulation.) It also sends similar messages to the component processes of the input lines. The times of these messages are set to incoming message time (message.time) $- d_i$, where d_i is the delay between the input line i and the output line. When the server 1 receives a message, it finds the difference between the max_time of the message and incoming message time. This difference is subtracted from all of the watched request times of incoming message line and marks the events corresponding to these times in the input line.

Therefore, during phase II, only those events that are marked are used for simulation and only those components that are marked are simulated. The formal algorithm for phase I is given in Figure 4.

Each process detects the termination condition by referring to a shared variable. Each process, when its message queue is empty, checks whether any other process is working. It terminates if no process is working on a message.

Characteristics of PD

The worst case processing time for each message which server 1 receives is $O(N_j \cdot \log_2 m_i)$, where j is the source line of the message, N_j is the number of requests on watched output line j, and m_i is the number of input events at the current input line. This is because we use binary search to mark a particular event out of all the m_i events. This binary search is done for each of the N_j requests on the watched output line j. The number of such messages received depends on the circuit and the number of watched output lines and the maximum time at which the watched request lines are watched. If the circuit does not have any feed-back or feed-forward loops, then for every input, at most one message will be received for each of the watched output lines.

The approach requires the minimum amount of memory when compared to DD, and does not require more memory than discrete-event simulation. If the components are not reachable from any of the watched lines they are not evaluated. Even the com-

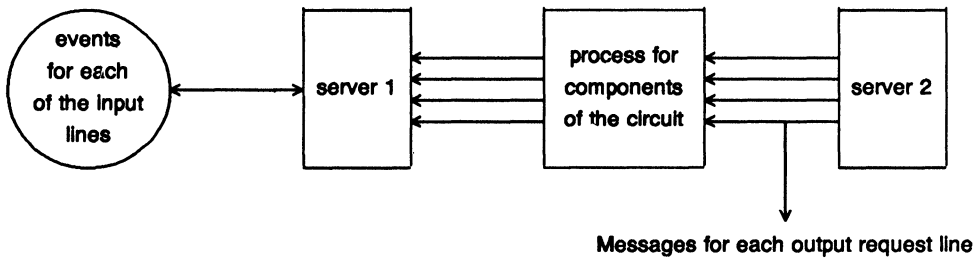

FIGURE 3 Processes in PD Phase I.

```
Algorithm PD, phase I:

begin:
    INPUT :
        Circuit configuration.
        Watched output lines and their watched times.
        Input events.

    OUTPUT :
        Marks on the input events that are sufficient to find the watched output lines at
        watched times.
        Sets the flag of the components that are to be simulated.

Server 2 :
{
    for each watched line j {
        max_time= max_watched_time(j);
        put message (max_time,j,max_time) in the queue of the component process
        which has j as one of its output lines.
    }
}


Component process C :
    A copy of this process is run for each component of the circuit.
{
    while ( not terminate) {
        while (msg_there) {
            pick up a message;
            component.flag = TRUE;
            for each input line i of component C {
                send message (message.time - d_i, mesh sage.line,message.max_time)
                where d_i is the delay time between line i and the output line.
            }
        }
    }
}


Server 1 :
{
    while (not terminate) {
        while (msg_there) {
            pick up message;
            difference = message.max_time - message.time;
            for each request time r of message.line {
                mark input event with time r - difference;
            }
        }
    }
}
end.
```

FIGURE 4 Algorithm for phase I.

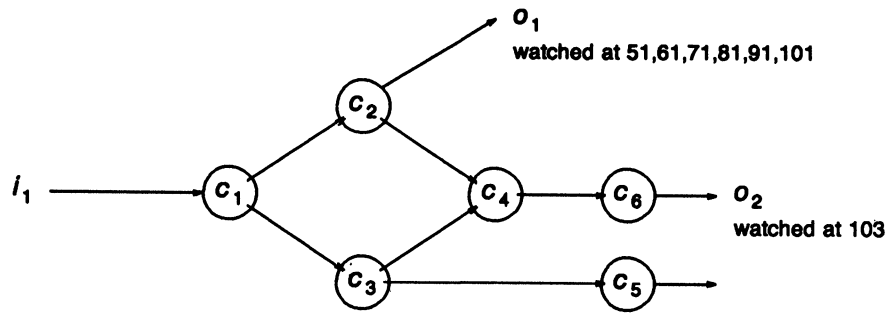

FIGURE 5 Redundant computation in PD.

ponents that are farther in time from all the watched output lines by more than the maximum watched time of each of the watched lines are also not evaluated. The number of component evaluations in this algorithm is greater than or equal to that in demand-driven simulation. This is illustrated by using an example. In Figure 5 each component has unit delay. The output of c_2 is watched at times 51, 61, 71, 81, 91, and 101. The output of c_6 is watched at time 103. The input line 1 is marked for the events at 49, 59, 69, 79, 89, and 99 by phase I of PD. The output line 2 is simulated for all these input events. However, for its watched output, only event at 99 needs to be simulated. Table 1 presents all the evaluations involved in DD and PD algorithms.

If the circuit is simulated many times with different input vectors and watched times, all of phase I need not be repeated every time. The algorithm can be easily modified so that the parts of phase I are not performed for each instance of the input vector.

The distributed implementation of phase I would require extra overhead for distributed termination detection. Additional messages would be needed for termination detection.

IMPLEMENTATION AND PERFORMANCE

Implementations

The proposed algorithms have been implemented on the Sequent Balance 8000 with 6 processors and 12 M Byte of memory. Each Balance 8000 processor is a National Semiconductor NS32032 Microprocessor, and all processors are connected to a shared memory by a shared bus.

The language used for the implementations is 'C'. The I/O and initialization times are not included in the execution time comparison of the algorithms. For timing purposes, we used the Gprof utility available in the UNIX operating system, which produces an execution profile of C programs.

We used two approaches in implementing the simulation algorithms. They are static and dynamic assignment of nodes to UNIX processes.

1) Static approach: A set of nodes of the distributed algorithm is assigned to a UNIX process. Normally, the number of nodes would be much greater than the number of physical processors. So, the nodes should be clustered and these clusters are assigned to individual physical processors. One advantage of static assignment is that communication between nodes in a cluster can be done "locally" without the overhead for message queue locking.
2) Dynamic assignment: Logical processes are assigned to physical processors dynamically during the simulation. Idle processes obtain work from a shared queue or stack of unassigned logical processes. This queue or stack must be locked before a process can be allocated an unassigned logical process. After picking up a logical process, the process satisfies any outstanding work of the logical process. One ad-

TABLE I
Times at Which Components are Evaluated for PD and DD

Component	DD	PD
c_1	49, 59, 69 79, 89, 99	49, 59, 69 79, 89, 99
c_2	100, 50, 60 70, 80, 90	50, 60, 70 80, 90, 100
c_3	100	50, 60, 70 80, 90, 100
c_4	101	51, 61, 71 81, 91, 101
c_5	102	52, 62, 72 82, 92, 102
c_6	None	None
Total Evaluations	15	30

vantage with dynamic assignment is that it naturally balances the work load between the parallel processors but incurs synchronization overhead normally for access to the data structures of the logical processes and also to access the global work queue or stack.

Distributed demand-driven simulation

The DD has been implemented in both static and dynamic approaches. In the static approach, a set of components is selected randomly and assigned to one of the available processes. This process handles the messages that are intended for any of these components. The components are divided in such a way that each available process has a near equal number of components. Each process has a queue of its own. All of the watched output line requests are initially queued to the processes which have the components of these lines. A message generated is added to a process's queue if the destination component is assigned to that process. The process terminates if its own queue is empty and no other process is working on a message.

In the dynamic scheme, each process has a local stack and there is a global stack. At first all the watched line REQUEST messages are put in the global stack. Whenever a process is idle, it takes a message from the global stack and pushes all the messages generated by this message evaluation into its own stack. It processes these messages until the stack is empty. When a process is idle, if the global queue is empty, it takes a message from the local stack of some other process and processes the message. A process terminates if the global queue is empty and all the local stacks are empty and no other process is working on a message. In the dynamic approach, in addition to locking the global and the local stacks while accessing, some component data structures need to be locked before accessing.

Figure 6 illustrates the average speedups of static and dynamic approaches for ten randomly generated combinational circuits, each with 500 components. As can be seen in Figure 6, the dynamic approach has better performance. This is because in static algorithm no heuristics are used to assign components to processes. Thus, the load of computation is not evenly balanced among the processes. Because of the better performance of the dynamic over the static approach, only the dynamic approach is used for comparison with later algorithms.

Parallel demand-driven simulation (PD)

Both phase I and phase II were implemented using the dynamic approach. The phase I implementation is very similar to the implementation of DD. We will describe the implementation of phase II in the following paragraph.

Phase II: The implementation is similar to the implementation of distributed discrete-event simulation (DDE) in [2]. In the implementation, the component can be in two states. A component process is either ready for next computation or is waiting for I/O

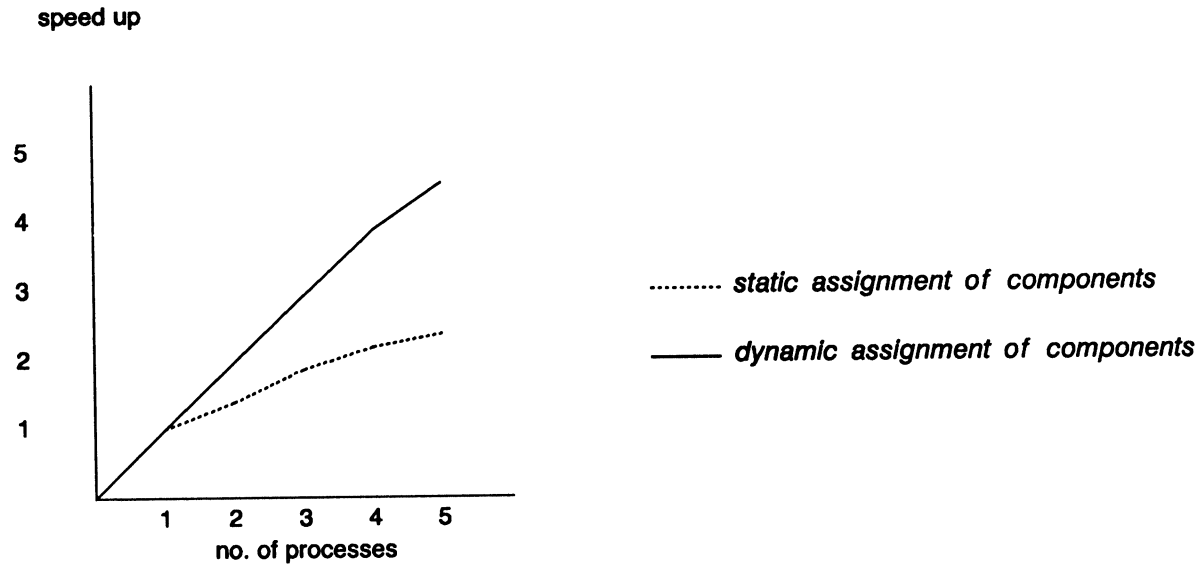

FIGURE 6 Average speed up for dynamic and static methods.

operations i.e. waiting for one or more of its input values or waiting to send the previous output events. There is a global queue of processes which have completed one or more of their I/O operations of the algorithm. An idle process fetches a component from the queue and checks for the completion of the I/O operations. If the operations are complete, it finds the output values for the next time value and tries to put the output values to the input components of these output lines. The input components waiting for these line values are queued to the global queue. While the process is trying to put the output values, it also waits on those input lines whose clock values are equal to the component's virtual clock value. Further details about the methods of maintaining the clocks can be found in [9]. If the I/O operations are not complete, the process fetches the next component from the global queue. In this implementation, deadlock occurrence is avoided by using null messages [7, 9].

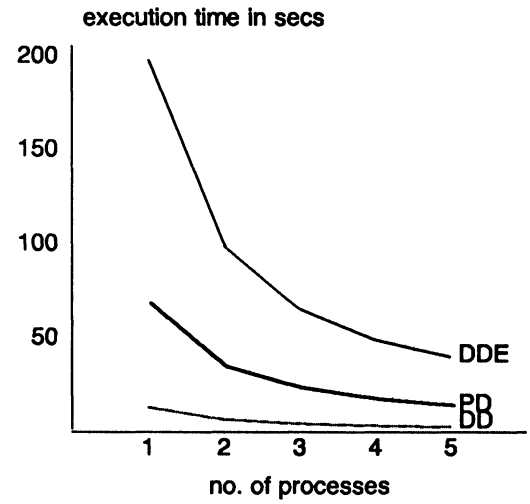

FIGURE 7 Average execution times for DD, PD, and DDE with 500 components.

Performance

The performance of demand-driven simulation algorithms depends on many factors, namely, the number of watched requests, the time of watched requests, the level of the circuit (maximum delay between an input and a watched output), feed back loops, feed forward loops, and the interrelation between these loops. We compared demand-driven algorithms and discrete-event simulation algorithm by varying the number of watched output requests. We have varied the number of watched requests as a percentage of the total number of input events.

The algorithms, distributed demand-driven simulation I (DD), parallel demand-driven simulation (PD), and distributed discrete-event simulation (DDE) have been used to simulate randomly generated circuits.

Combinational circuits

Figure 7 gives the average performance of the algorithms when circuits with 500 components were simulated. The number of processes used against the execution times are plotted for the algorithms. The percentage of watched outputs was fixed at 10. DD performed the best. The PD is on the average 5 times slower than the DD. PD performed 3-4 times better than DDE. The speedups were linear in the number of processes for all the algorithms. This is because we did not implement a separate sequential discrete-event simulation. In addition, we used null messages to avoid deadlock. This gives concurrency at the cost of extra computation.

Figure 8 gives the effect of an increase in the watched request percentage on the algorithms DD and PD. When the above percentage was increased to 70, both PD and DDE had near equal execution times. For 5000 components, the execution times became near equal only when the percentage was 180. This is shown in Figure 9.

Figure 10 presents the effect of circuit level on DD, PD, and DDE. The circuits were simulated with 500 components, changing the circuit level while keeping the watched request percentage constant. In Figure

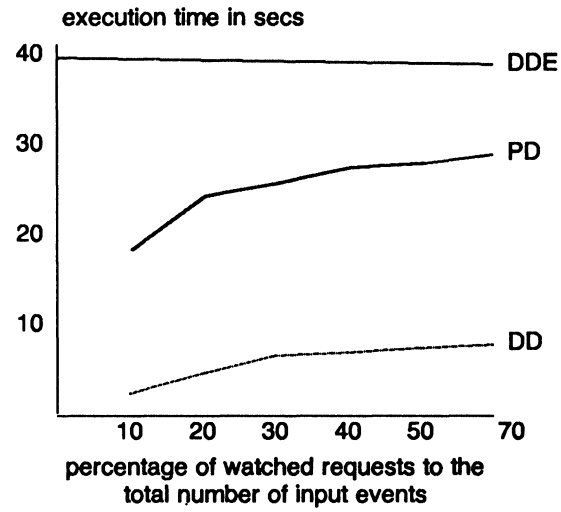

FIGURE 8 Execution times for DD, PD, and DDE with different watched request percentages for a circuit with 500 components.

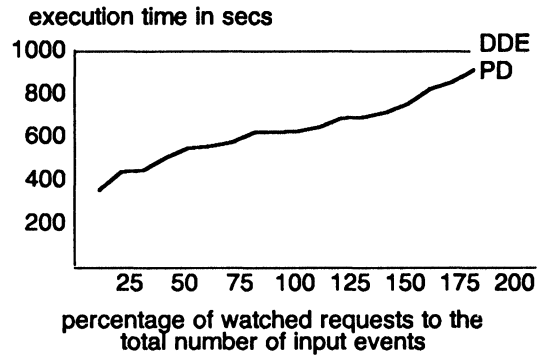

FIGURE 9 Execution times for PD and DDE with different watched request percentages for a circuit with 5000 components.

10, as the levels were increased, the execution time DDE also increased. However, DD and PD did not show this tendency. The execution times started to decrease when we increased the number of levels over 13. This is due to the fact that the implementation of PD does not simulate a component above the maximum time of the output required at that component. Thus, the components at the front end of the circuit are simulated a fewer number of times in PD.

Sequential circuits

Similar simulations were done with sequential circuits. Figure 11 shows the average execution time for ten different sequential circuits with 50 components. Figure 12 presents the execution time for a data path with 65 components. (The data path includes a control unit, a multiplier, an adder, and

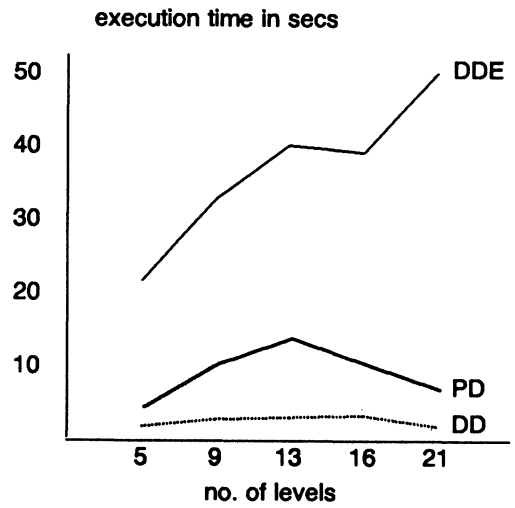

FIGURE 10 Effect of level on DD, PD, and DDE.

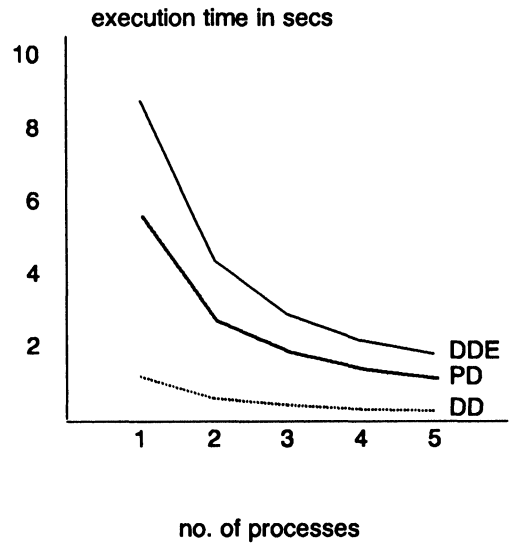

FIGURE 11 Average execution time for DD, PD, and DDE for sequential circuits with 50 components.

several registers.) The watched request for both experiment was fixed at 5. In Figure 13, the execution times with watched output percentages 5, 10, 15, 20 and 25 are plotted. As can be seen, PD is very sensitive to an increase in the watched request percentage. This can be attributed to the fact that, in sequential circuits, for a single request many events and components are marked. Phase I of the algorithm becomes very expensive. However, in DD, because most of the requests stop well before their input, the increase of the watched request percentage has less effect on the execution time. This would

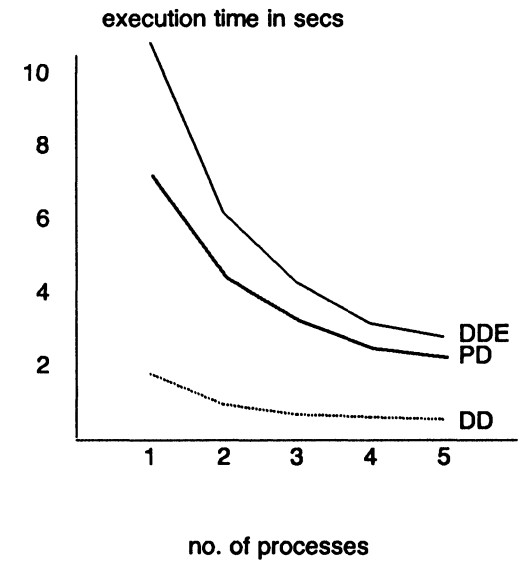

FIGURE 12 Average execution time for DD, PD, and DDE for a data path with 65 components.

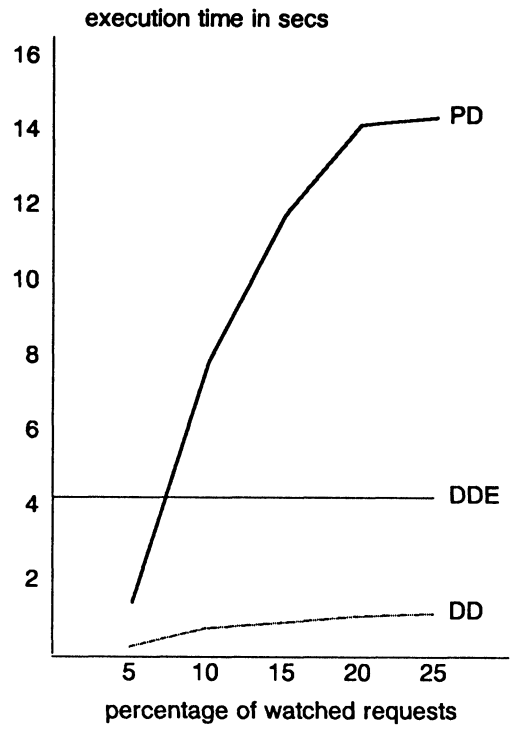

FIGURE 13 Execution times for DD, PD, and DDE with different watched request percentages for a 50 component sequential circuit.

happen only if the intermediate values evaluated are stored at all of the output lines for future use.

CONCLUSION

In the paper, distributed and parallel simulation algorithms were proposed. In our distributed algorithm, the order of messages sent and received at the source and destination processes need not be maintained. This makes algorithm robust for distributed implementations. The algorithm performed many times faster than the distributed discrete-event simulation for the circuits we simulated. In our parallel algorithm, we have reduced the infinite memory requirement of distributed demand-driven simulation. Our parallel algorithm performed two to three times faster than discrete-event simulation for combinational circuits. Our algorithms have used one process/component approach. This is done to simplify the presentation of the algorithms. This approach has been used in many distributed algorithms in the literature [2, 3, 5, 6, 8, 10]. In practice a set of these logical processes needs to be handled by a single physical process, similar to our implementation. In our implementation, we partitioned components randomly. We have not described an effective scheme to partition the components so that the computational load is distributed across processors and also the communication overhead is reduced. This is an interesting area yet to be investigated. Further work needs to be done to find some more heuristics to reduce the cost of Phase I of parallel demand-driven simulation.

Acknowledgments

The authors wish to thank the anonymous referees for their helpful suggestions.

References

[1] L. Soule and T. Blank, "Statistics for Parallelism and Abstraction Level in Digital Simulation," *Proc. of the 24th ACM/IEEE Design Automation Conference*, June 1987, pp. 588–591.

[2] K.M. Chandy, and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of distributed programs," *IEEE Trans. on Software Eng.*, vol. SE-5, no. 5, Sept. 1979, pp. 440–452.

[3] K.M. Chandy, and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of ACM*, Apr. 1981, pp. 198–206.

[4] J. Mistra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, March 1986, pp. 39–65.

[5] F.J. Kaudel, "A Literature Survey on Distributed Discrete Event Simulation," *Simuletter*, June 1978, pp. 11–21.

[6] D.R. Jefferson, "Virtual Time," *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 404–425.

[7] S.P. Smith, M.R. Mercer, and B. Brock, "Demand Driven Simulation: BACKSIM," *Proc. of the 24th ACM/IEEE Design Automation Conference*, June 1987, pp. 181–187.

[8] R.E. Bryant, "Simulation on a Distributed System," *Proc. of the 1st Intl. Conference on Distributed Computer Systems*, Oct. 1979, pp. 544–552.

[9] D.A. Reed, A.D. Malony, and B.D. McCredie, "Parallel Discrete Event Simulation Using Shared Memory," *IEEE Trans. on Software Engineering*, Vol. 14, No. 4, April 1988, pp. 541–553.

[10] D.R. Jefferson, and H. Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism," *Distributed Simulation, Simulation Council Inc.*, La Jolla, Jan. 1985, pp. 63–69.

Biographies

KRISHNAMURTHY SUBRAMANIAN received B.Tech. degree in Computer Science from Regional Engineering College, India, the M.S. degree in Computer Science from Southern Illinois University at Carbondale, in 1985, 1989, respectively. He is presently Simulation Engineer at Zycad Corporation, Menlo Park, CA. His current research interests include modeling of computer systems, and accelerated hardware circuit simulation.

MEHDI R. ZARGHAM is an Associate Professor in the Computer Science Department of Southern Illinois University at Carbondale. He received his MS and Ph.D. degrees in computer science from Michigan State University in 1980 and 1983, respectively. He has held visiting position at International Computer Science Institute, Berkeley, California. His research interests include routing and placement in VLSI design, parallel processing, and neural networks.