Parallel Logic Simulation of VLSI Systems

MARY L. BAILEY

Department of Computer Science, University of Arizona, Tucson, Arizona 85721

JACK V. BRINER, JR.

Department of Mathematics, The University of North Carolina at Greensboro, Greensboro, North Carolina 27412

ROGER D. CHAMBERLAIN

Department of Electrical Engineering, Washington University, St. Louis, Missouri 63130

Fast, efficient logic simulators are an essential tool in modern VLSI system design. Logic simulation is used extensively for design verification prior to fabrication, and as VLSI systems grow in size, the execution time required by simulation is becoming more and more significant. Faster logic simulators will have an appreciable economic impact, speeding time to market while ensuring more thorough system design testing. One approach to this problem is to utilize parallel processing, taking advantage of the concurrency available in the VLSI system to accelerate the logic simulation task.

Parallel logic simulation has received a great deal of attention over the past several years, but this work has not yet resulted in effective, high-performance simulators being available to VLSI designers. A number of techniques have been developed to investigate performance issues: formal models, performance modeling, empirical studies, and prototype implementations. Analyzing reported results of these techniques, we conclude that five major factors affect performance: synchronization algorithm, circuit structure, timing granularity, target architecture, and partitioning. After reviewing techniques for parallel simulation, we consider each of these factors using results reported in the literature. Finally we synthesize the results and present directions for future research in the field.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*simulation*; B.7.2 [**Integrated Circuits**]: Design Aids—*simulation*; C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*distributed applications*; I.6.3 [**Simulation and Modeling**]: Applications; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event*; *distributed*; *parallel*

General Terms: Experimentation, Algorithms, Performance

Additional Key Words and Phrases: Circuit structure, parallel architecture, parallelism, partitioning, synchronization algorithm, timing granularity

1. INTRODUCTION

The design of large digital systems and, in particular, the design of VLSI systems have increased the importance of logic simulation in the overall design process. Extensive simulation-based design verification prior to fabrication is necessary because probing and repair of already-fabricated VLSI systems is currently

Mary Bailey was supported in part by the National Science Foundation under grant CCR-9212018; Jack Briner was supported in part by the National Science Foundation under grant MIP-9108906; and Roger Chamberlain was supported in part by the National Science Foundation under grant MIP-9309658.

CONTENTS

impractical. As systems have grown, simulation tasks have become significant bottlenecks in the design cycle. In an attempt to address this bottleneck, researchers have turned to parallel and distributed processing.[1]

VLSI systems exhibit a great deal of concurrency, which is inherent in their normal operation. Standard discrete-event simulation algorithms, however, serialize this activity, and therefore do not exploit the concurrency present in the underlying system (the VLSI circuit, in this case). If the concurrency inherent in the simulated system can be exploited by parallel versions of the simulation algorithms, parallel processors can be used to perform the simulation task and yield significant performance improvements over uniprocessor architectures. It is not unreasonable to believe that two to three orders of magnitude performance improvement may be achievable by using parallel processing.

Five major factors affect the performance of parallel logic simulation:

(1) Synchronization algorithm
(2) Circuit structure
(3) Timing granularity
(4) Target architecture
(5) Partitioning and mapping

A **synchronization algorithm** is used to coordinate the simulation across multiple processors. A number of synchronization algorithms have been proposed for discrete-event simulation on parallel machines, including synchronous, conservative, and optimistic approaches. We will also discuss an alternative synchronization algorithm, the oblivious approach, which is not based on events. The **circuit structure** of the VLSI system, as well as its input vectors, can have a dramatic effect on the performance of parallel simulations. Simulations of some circuits exhibit good parallel performance, while others have proven to be problematic. Even given the same circuit, different inputs vectors give dramatically different performance. The **timing granularity** of the underlying logic simulator also has an effect on simulation performance. There is a wide spectrum of timing granularities, ranging from fine-grained (e.g., 0.1 ns time resolution) to coarse-grained granularities (e.g., unit-delay or zero-delay).

The **target architecture** impacts the performance of the parallel simulation, as it does for all parallel programs. A related issue is the **partitioning** of the simulated circuit among the parallel processors. Prior to initiating one of the par-

[1] Here, we will not distinguish between parallel and distributed processing, and treat the two words as synonymous. Architectural differences between the two are discussed in Section 6.

allel simulation algorithms, the circuit elements must be partitioned and assigned or **mapped** to individual processors. This problem is related to the general problem of task assignment and load balancing on parallel machines.

Many of these factors are present in all parallel simulations. Indeed, there has been a great deal of work in general parallel and distributed simulation over the past few years. Unfortunately, there have been a limited number of general results, in part due to the wide variety of applications. Logic simulation is one application area that has received significant attention, largely because of its potential economic impact. Although Smith [1986] assessed the state of parallel logic simulation, a great deal of research has been performed since 1986.

In this survey we will discuss and analyze the current state of the art of parallel logic simulation by focusing on five factors: synchronization algorithm, circuit structure, timing granularity, target architecture, and partitioning. In particular, we are interested in understanding relationships between the factors.

The survey begins with a brief overview of logic simulation. Next we review common mechanisms for parallelizing logic simulation and the synchronization algorithms necessary to keep a parallel simulation consistent with an equivalent sequential one. Sections 4 through 7 describe and synthesize techniques and results of research investigating the five factors impacting performance. First, Section 4 reviews how researchers have compared the synchronization mechanisms using formal modeling techniques. In Section 5 we consider the relationship between circuit structure and timing granularity. In Section 6 we review target architectures. The effect of partitioning and mapping on performance follows. Once the five factors and work relating them have been reviewed, two sections, performance models and implementations, describe results of work that, directly or indirectly, studies the interrelationships among the factors. Finally, we conclude with a summary of the current state of the art and issues for future directions in parallel logic simulation.

2. LOGIC SIMULATION

VLSI circuits are simulated at a multitude of abstraction levels, from the circuit level to the behavioral level. In circuit-level simulation, node voltages are represented by continuous values, and the simulator solves numerically the differential equations representing the circuit. In logic-level simulation, node voltages are represented by discrete quantities and change state at discrete points in time. The term *logic simulation* is used in a number of ways. Some people use logic simulation to mean the simulation of gate-level circuit elements (e.g., NAND gates, flip flops). Others use a broader definition, using logic simulation to mean any discrete simulation of a VLSI circuit, where circuit components vary from transistors (modeled as ideal switches), through traditional logic gates, to high-level behavioral models (e.g., processors, multipliers). We use this broader definition throughout.

In discrete-event simulation, system state variables are modeled as discrete-valued quantities that change value at discrete points in time. In logic simulation, the state variables represent typically signal levels on wires that interconnect circuit elements. In the simplest two-valued logic simulations, state variables are constrained to two quantities representing Boolean values (i.e., 0 or 1). Most modern logic simulators use multivalued variables to represent additional information. For example, many switch-level simulators add an X state to represent unknown or floating signals, and gate-level simulators add states to represent drive strength and high-impedance conditions. The IEEE standard logic system for VHDL simulation (STD_LOGIC_1164) uses a 9-valued logic; the allowable states are shown in Table 1 [Billowitch 1993].

In logic simulation, state changes are restricted to discrete points in time. The resolution of the simulation clock deter-

Table 1. IEEE Standard Logic System

State	Purpose
U	uninitialized state
X	forcing unknown
0	forcing zero
1	forcing one
Z	high impedance
W	weak unknown
L	weak zero
H	weak one
-	don't care

mines the allowable points in time when state variables can change. This is referred to as the timing granularity, and allowable time values are called time points or time steps. Using a fine granularity, a simulator can model timing-dependent behavior more accurately, with the possible penalty of an increase in execution time. Simulators with a course granularity provide little (if any) timing behavior, but generally have faster execution. The specific impact of timing granularity on performance is examined in Section 5.

For simulation purposes, we model a VLSI system as a collection of logic elements at varying levels of abstraction (e.g., transistors, NAND gates, flip flops, multipliers, etc.) and their interconnecting wires. Input vectors are then provided to exercise the circuit. A gate-level example using three logic elements is shown in Figure 1. This example has two inverters (labeled α and β) and a single AND gate (labeled γ); the interconnecting wires are labeled a through e. The delay through each gate (the time required for an input signal change to be reflected at the output) is indicated immediately below the gate.

State changes are represented by events in the simulation, such as the change in output value of an individual gate. As in general discrete-event simulation, pending state changes (those at some future point in simulated time) are retained in an event queue data structure, sorted by event time. The logic simulation algorithm is given in Figure 2. Assuming a two-valued simulation and a timing granularity of 1 ns, a representative simulation of the example circuit is described below.

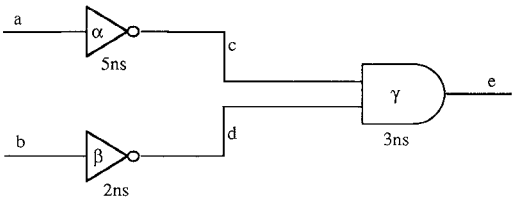

Figure 1. Gate-level example circuit.

The simulation is initialized at time $t = 0$ ns, with signals $a = b = 1$ and $c = d = e = 0$. The input signals are to change as follows: a changes to 0 at $t = 1$ ns, and b changes to 0 at $t = 2$ ns. This is represented by an initial event queue of $[(t = 1, a = 0); (t = 2, b = 0)]$. The initial simulation state is represented by the first line of Table 2, and the main loop of the simulation is then executed.

In the loop execution, the first event is removed from the event queue. Simulated time is updated to $t = 1$ ns; signal a is changed to 0; and gate α is evaluated. The result is that signal c will change to 1 at $t = 6$ ns (current simulated time of 1 ns plus the gate delay of 5 ns). This is scheduled in the event queue, and the simulation state is represented by the second line of Table 2.

In the second loop, simulated time is updated to $t = 2$ ns; b is changed to 0; and gate β is evaluated. The result is that d will change to 1 at $t = 4$ ns, which is scheduled in the queue. Notice that in the queue, signal d is ahead of signal c, since the signal d change has a smaller timestamp.

In the third loop, $t = 4$ ns; d is 1; and γ is evaluated. Since the output of γ does not change, no additional events are scheduled. At $t = 6$ ns, c is 1, and γ is evaluated again. This time, e is sched-

```
while (event queue is not empty)
  retrieve next event from event queue
  update simulated time to time of event
  update gate output to new value
  for each gate connected to gate output
    evaluate logic function
    if output changes then schedule change
        in event queue
  endfor
endwhile
```

Figure 2. Simulation algorithm.

Table 2. Simulation State

Time	Signals					Queue
t	a	b	c	d	e	
0	1	1	0	0	0	[(t=1,a=0);(t=2,b=0)]
1	0	1	0	0	0	[(t=2,b=0);(t=6,c=1)]
2	0	0	0	0	0	[(t=4,d=1);(t=6,c=1)]
4	0	0	0	1	0	[(t=6,c=1)]
6	0	0	1	1	0	[(t=9,e=1)]
9	0	0	1	1	1	[ ]

uled to change to 1 at $t = 9$ ns. At $t = 9$ ns, e is 1, and the event queue is empty. The final simulation state is represented by the last line of Table 2.

Note that even though the simulation ran up to time $t = 9$ ns, the simulation loop was not executed 9 times (once per ns). The loop was executed only 5 times (once per required gate evaluation). The event-driven nature of the algorithm allows the simulator to skip time points that have no circuit activity, thereby improving performance (i.e., decreasing execution time).

As VLSI integrated circuits increase in size (more than a million transistors on a chip and climbing), the time required to execute the simulation algorithm becomes unacceptably long, even using event-driven techniques. This is due to a number of factors. First, the number of required functional evaluations grows as the number of logic elements grows. Second, as the number of pending events gets larger, the overhead associated with managing the event queue increases. Third, with larger circuits, a larger number of input vectors are needed to verify proper circuit operation, further increasing the length of the simulation run.

For this reason, parallel machines are being investigated as a vehicle for increasing the performance of VLSI logic simulations. The event-driven algorithm (Figure 2) is serial in nature, executing events in sequential order. However, this is a limitation of the *algorithm*, not the VLSI system. For example, the evaluation of gates α and β could clearly be executed in parallel without altering the results of the simulation.

There are, in fact, a number of ways that parallelism can be exploited to improve simulator performance [Mueller-Thuns et al. 1990]. *Algorithm parallelism* uses pipelining techniques to accelerate the major loop by executing individual program steps on different processors (e.g., event queue management, functional evaluation). A limited amount of parallelism is available using this technique, since there are a limited number of steps in the major loop. *Data parallelism* uses different processors to simulate the circuit for distinct input vectors. This technique is quite effective for fault simulation, where a large number of independent input vectors need to be simulated. It is less effective, however, during design verification, where the goal is to minimize the completion time of an individual input vector. *Model parallelism*, alluded to in the previous paragraph, uses different processors to perform the functional evaluations for distinct logic elements. When state changes on one processor affect the simulation on another processor, a timestamped message is used to communicate both the state change and the simulated time the state change occurs in. A time synchronization algorithm is then needed to determine which functional evaluations can safely be executed in parallel. This survey concentrates on techniques for exploiting model parallelism, exploring the

major factors that impact the performance of parallel logic simulation.

3. PARALLEL LOGIC SIMULATION

Prior to executing a parallel simulation, the logic elements are typically assigned to individual processors. The functional evaluations for each logic element are then executed by its assigned processor. To maintain correct simulation time, coordinating execution between processors is crucial. The simulation clock is the usual mechanism for this coordination.

In sequential, event-driven simulation, events are typically maintained on a time-ordered queue. As events are removed from the queue, simulated time is updated and the events evaluated, which may cause other events to be placed on the queue. In parallel simulation, there are often multiple queues, one per processor. Coordinating event evaluations and managing these queues are necessary to ensure correct simulation. There are several mechanisms for ensuring correctness; we refer to these as time synchronization strategies. We summarize the most common synchronization strategies in this section. For a more complete description of the current state of research in general parallel discrete-event simulation, see Fujimoto [1990].

3.1 Oblivious Simulation

The oblivious strategy is *not* event driven. Instead, all circuit elements are evaluated during every time step, whether or not their inputs have changed. The workload here is fixed for each time step, so scheduling can be performed statically at compile time, and no scheduling overhead is incurred at run time.

Rank ordering is often used in these simulators as a means of scheduling element evaluations. All elements, generally gates, are ordered according to the availability of their inputs. Gates whose inputs are also inputs to the simulation are at rank 0. A gate is at rank i if all of its inputs are produced by gates at ranks less than i and at least one of its inputs is produced by a gate at rank $i - 1$. Evaluating gates in rank order ensures that (1) the inputs for all gates will be stable, (2) each gate will be evaluated a single time, and (3) gates will be evaluated as soon as possible. In the example of Figure 1, gates α and β are at rank 0, and gate γ is at rank 1.

To parallelize the oblivious algorithm, three approaches can be taken. The first is to use a vector processor and design the simulation so that identical operations are performed on gates of a given type at the same level. The second approach is to use independent, general-purpose processors as a pipeline, evaluating one rank on every processor. The third approach schedules element evaluations among the processors by solving a general optimization problem, maximizing processor utilization while keeping the number of evaluations constant between processors and minimizing interprocessor communication [Kravitz et al. 1991].

In oblivious strategies, the major source of overhead for logic simulation is redundant evaluation of elements whose inputs have not changed. However, the computation per element is often much less than that in event-driven strategies. Thus in comparing the two strategies, one must consider both effects. Let the amount of computation for an individual element evaluation in the event-driven strategy be C times that in the oblivious strategy. Additionally, assume that E is the ratio of the number of events in the event-driven strategy to the number of evaluations in the oblivious strategy. Then for a sequential simulation, the oblivious strategy is preferred if $E > 1/C$. Currently values for C are approximately 100 for traditional simulators, making the oblivious strategy preferred if $E > 0.01$. Recently, techniques have been proposed that reduce C to between 20 and 50, making the oblivious strategy preferred if $E > 0.04$ to 0.10 [Lewis 1991].

Another criticism of the oblivious strategy is the coarse timing model typically used in these simulations. This crit-

icism has been addressed recently by the advent of oblivious algorithms for finer timing models, but their impact on C is unclear [Maurer and Lee 1994; Shriver and Sakallah 1992].

Comparing oblivious and event-driven strategies for parallel simulation is complicated by the addition of synchronization and communication mechanisms. Static scheduling is feasible for oblivious simulations, resulting in still more savings over the event-driven strategy. However, low circuit activities still favor the event-driven strategy. Large pipelined circuits may have enough activity to make the oblivious strategy an attractive alternative to the event-driven strategy; more research is needed to determine when each strategy is preferred.

3.2 Synchronous Algorithms

The most obvious synchronization algorithm is to have all processors work on the same time step in a synchronous lock-step fashion. Since the resulting simulated time is common across all processors, this is also referred to as a global-clock algorithm.

Consider the example simulation of Figure 1, and assume each gate is assigned to a distinct processor. The sequence of operations in a synchronous algorithm is illustrated in Table 3. Initially (t = 0 ns), messages are delivered from the primary inputs to gates α and β describing the input vector. At global time t = 1 ns, α is evaluated, causing a message to be sent from α to γ with a timestamp of t = 6 ns; at time t = 2 ns, β is evaluated, causing a message to be sent from β to γ with a timestamp of t = 4 ns; at time t = 4 ns, γ evaluates the message from β (the message with the smaller timestamp); and at time t = 6 ns, γ evaluates the message from α, sending a message to the output for t = 9 ns. Since no two evaluations occur at the same point in simulated time, no parallelism is exploited in this example. The amount of parallelism available in realistic circuits is examined in Section 5.

Table 3. Synchronous Example

Time	Evaluations	Messages	
0		in → α (t=1,a=0)	in → β (t=2,b=0)
1	α	α → γ (t=6,c=1)	
2	β	β → γ (t=4,d=1)	
4	γ		
6	γ	γ → out (t=9,e=1)	

The difficulties in this algorithm include determining when all processors have completed a time step and what the next time step should be. Determining completion can be accomplished with a simple barrier which may be supported by the parallel architecture or software. Determining the next time step depends on how the events are managed. If there is a central event queue, the central event queue simply finds the lowest time; however, insertions and deletions from the queue can serialize. When each processor has a local queue, a global minimum operation must be performed. Additionally, another problem develops, that of load balancing. Because the processors will likely have different numbers of events active during a given time step, some processors may finish earlier than others, resulting in potentially significant load imbalance.

3.3 Conservative Asynchronous Algorithms

To reduce the problem of load imbalance and central-queue contention, algorithms which allow the processors to proceed at independent rates with independent queues and clocks are attractive. If each processor or logic element maintains its own local simulation time, the algorithm is known as a local-clock or asynchronous algorithm. There are two classes of local-clock algorithms: conservative and optimistic.[2]

[2]The synchronous algorithm described earlier can also be classified as a conservative algorithm.

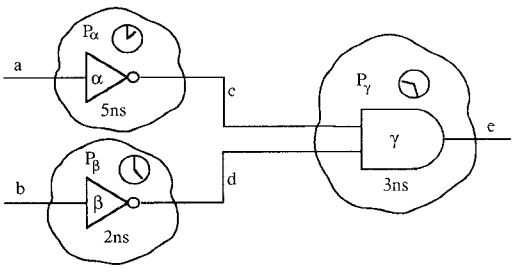

Figure 3. Local clock example.

Figure 3 shows the example circuit with each gate assigned to an individual processor. The clock symbol associated with each processor indicates the fact that simulation time is maintained locally, within the processor. The value of the local simulation time may therefore be different from one processor to the next.

Conservative asynchronous algorithms have their origins in Chandy and Misra [1981], and Bryant [1977]. They require that the local simulated time associated with a logic element is only advanced to the extent that the advance cannot violate causality in the system being modeled (i.e., before a logic element will advance its local simulated time to t, it must know that it will receive no additional messages with timestamps less than t). In order to be able to draw conclusions about the timestamps of messages it might receive in the future, the conservative algorithms require that messages from one logic element to another be sent in nondecreasing timestamp order.

To ensure compliance with the conservative requirements, two constraints are placed on the logic elements. The first is called the input waiting rule, which constrains the advancement of local simulated time to be the minimum timestamp associated with the last message received from any other logic element. Thus the input waiting rule ensures that messages are processed in timestamp order. The second constraint, the output waiting rule, ensures that messages to other processes arrive in timestamp order. Messages waiting for output must not be sent before it is certain that all other output messages will have later timestamps. If different events have different propagation times (such as different rise and fall times for a gate element), then output events must be queued to ensure that all messages arrive in timestamp order. An assumption is made here concerning the underlying system, that it supports FIFO message delivery on each channel.

Consider the simulation of our example circuit, again assigning each gate to a separate processor. The local time for each gate is maintained independently, and is initialized at 0 ns. Table 4 shows the sequence of operations in the conservative algorithm.[3] The first round of messages communicate the input vector to α and β. This updates their local time to $t = 1$ ns and $t = 2$ ns, respectively. Gates α and β can then be evaluated (in parallel), triggering two messages to γ. Two additional messages are sent from the inputs to α and β, indicating no more input changes will take place, updating their local times to $t = \infty$. The need for these two messages will be described below. As a result of the input waiting rule, γ's local clock can now be updated to $t = 4$ ns.

All three gates can now be evaluated (again in parallel), triggering two more messages to γ. Again following the input waiting rule, γ can now update its local time to $t = 6$ ns, since it now has a message from β indicating no additional messages will come between $t = 4$ ns and $t = 6$ ns. Gate γ is then evaluated, and a message is sent to the output at $t = 9$ ns.

As another example, consider the circuit in Figure 4. Assume that the propagation delay of each gate is 3 ns, and each gate is on a separate processor. The local clock of the processor containing the top gate has a simulated time of 1 ns, while the local clock of the lower proces-

[3] We are using lookahead to relax the output waiting rule.

Table 4. Conservative Example

Local time			Evaluations	Messages			
α	β	γ					
0	0	0		in $\rightarrow \alpha$ (t=1,a=0)	in $\rightarrow \beta$ (t=2,b=0)		
1	2	0	α, β	$\alpha \rightarrow \gamma$ (t=6,c=1)	$\beta \rightarrow \gamma$ (t=4,d=1)	in $\rightarrow \alpha$ (t=∞)	in $\rightarrow \beta$ (t=∞)
∞	∞	4	α, β, γ	$\alpha \rightarrow \gamma$ ($t = \infty$)	$\beta \rightarrow \gamma$ ($t = \infty$)		
∞	∞	6	γ	$\gamma \rightarrow$ out (t=9,e=1)			

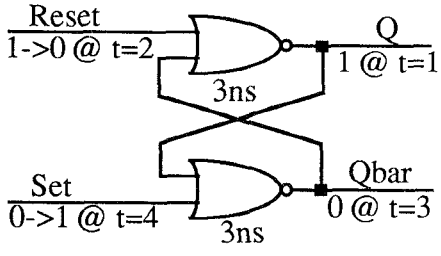

Figure 4. Flip flop example circuit.

sor is at time 3 ns. The processor containing the top gate can process the message changing the value of *Reset* from 1 to 0, since the value of *Qbar* will not change until after time 3 ns. However, the processor containing the lower gate *cannot* process the message changing the value of *Set* at time 4 ns, because it cannot determine that the value of *Q* will not change before time 4 ns, even though in this example *Q* would not change until time 5 ns.

As presented above, the conservative algorithm is prone to deadlock. For instance, if *Qbar* in the example is at time 1 ns instead of time 3 ns, then neither the *Set* nor *Reset* change can take place because there is no assurance that either *Q* or *Qbar* will not change before time 2 ns.

A number of techniques have been proposed to deal with the deadlock problem. These techniques can be broadly categorized into two classes: deadlock avoidance and deadlock detection and recovery. Deadlock avoidance techniques use a special message type that has a timestamp but no content (a *null* message) [Misra 1986]. Whenever a logic element receives a message, it must send a message on each of its outputs. If the simulation does not require a regular message to be output on a channel, a null message is sent in its place. The algorithm eliminates the potential for deadlock, but with the penalty of increasing substantially the total number of messages required to execute the simulation. In the first example, the messages from the input to gates α and β at $t = \infty$ are null messages.

The deadlock detection and recovery algorithms allow the basic conservative algorithm to deadlock, detect the deadlock condition, and invoke a recovery algorithm to break the deadlock [Chandy and Misra 1981]. Deadlock detection algorithms can be either centralized, typically only detecting global deadlock, or decentralized, typically using circulating-marker algorithms that can detect local deadlock conditions. The deadlock recovery algorithm often depends upon the type of detection algorithm used, but one algorithm usable in all cases is to perform a global minimum over all pending simulation events on all logical processes. The local simulated time can safely be advanced to the result of this global minimum operation, and the events at that simulated time processed, thereby breaking the deadlock.

There are other conservative approaches which have been reported in the literature. These involve using knowledge about the application to reduce the overhead associated with the Chandy-Misra algorithm. Lubachevsky [1989] uses a moving time window in which only events whose timestamp lies in the time window are eligible for processing. Lookahead is another approach which has been effective in reducing overhead [Fujimoto 1989]. In this case, a process with local time t knows all events it will produce up to time $t + L$, where L is the lookahead. In practice, logic gates often have a minimum delay, L, and will *not* produce any events before time $t + L$, which can be used to reduce the number of null messages sent.

3.4 Optimistic Algorithms

The original optimistic asynchronous algorithm, Time Warp, was devised by Jefferson [1985]. Here, whenever a message is received by a logic element, the process advances its local simulated time to the timestamp on the message and simulates the effects of the incoming message. This simulated time advance is performed independent of the fact that future messages might have a lower timestamp, thereby potentially invalidating the work performed when the original message arrived.

In the first example circuit (Figure 3), the simulation starts out as in the conservative algorithm, with messages from the input to gates α and β (see Table 5). Gates α and β are both evaluated (in parallel), each sending a message to γ. Both of these messages are evaluated by γ, and the local time of γ is optimistically updated to $t = 6$ ns, assuming no additional messages will come from β between 4 ns and 6 ns. Gate γ sends an output message for $t = 9$ ns.

In the second example circuit (Figure 4), both processors would process messages, assuming that no messages would arrive "in the past." If the top gate has a delay of 3 ns, then this assumption holds, and the simulation proceeds correctly.

Table 5. Optimistic Example

Local time			Evaluations	Messages	
α	β	γ			
0	0	0		in $\rightarrow \alpha$ (t=1,a=0)	in $\rightarrow \beta$ (t=2 b=0)
1	2	0	α, β	$\alpha \rightarrow \gamma$ (t=6 c=1)	$\beta \rightarrow \gamma$ (t=4,d=1)
1	2	6	γ	$\gamma \rightarrow$ out (t=9 e=1)	

However, if the top gate has a delay of 1 ns, then the lower processor has now processed a message at time 4 ns and will later receive a message at time 3 ns (from Q). It must undo the damage caused by processing the change to *Set* at time 4 ns. Thus, the event from Q triggers a roll back, forcing the lower processor to return to the state it had before it evaluated the *Set* event. Additionally, the event from *Set* could have caused an erroneous event to be sent to the top gate. To remove this event, an antimessage would be sent from the lower processor to the top processor. If the antimessage arrives before its corresponding real message is processed, both events are removed; otherwise the antimessage triggers a roll back on the receiving processor.

In order to perform roll back, processors must have saved the state of the circuit. During a roll back, local simulated time is backed up to the value associated with the incoming message; the system state associated with the logical process is restored from an earlier copy; and antimessages are sent out along output channels to invalidate any previously transmitted messages with timestamps greater than the new local simulated time. When these antimessages are received at their destination logical processes, they may trigger roll back on those processes as well.

Clearly, the Time Warp algorithm requires overhead to perform roll backs and save state. The saved state also takes up memory. To reduce the memory requirements, old states can be removed when no longer needed. The minimum of the processors' local times and messages in transit is known as the *global virtual*

time (GVT). No message can arrive at a processor earlier than GVT, so states and messages saved before GVT, called fossils, can be discarded.

Gafni's [1988] *lazy cancellation strategy* reduces the impact of roll back on the performance of simulation. Instead of *aggressively* cancelling previously sent messages whenever roll back occurs, the lazy cancellation algorithm waits to cancel the message until it is known that the wrong message had been sent. Thus, if the right event had been delivered for the wrong reasons, the receiving processor is not inhibited because of excessive causality constraints.

As with the conservative algorithms, there are variants on the optimistic strategy. One such variation is the Moving Time Window (MTW) algorithm proposed by Sokol et al. [1988]. It attempts to exploit the observation that the most likely events to be rolled back are those that are farthest ahead in simulated time. The MTW algorithm establishes a window immediately ahead of GVT and only allows local simulated time for a logical process to be advanced to a point within the time window. If an incoming message has a timestamp that is ahead of the window, it is placed in the local event queue and processed once GVT is advanced enough that the timestamp falls within the window.

4. SYNCHRONIZATION ALGORITHMS

Comparing synchronization algorithms for general parallel simulations is difficult. However, there has been some success in using formal models to compare these algorithms in the logic simulation domain. We summarize these results here.

While considering the effect of timing granularity on circuit parallelism, Bailey [1992b] considered two synchronization strategies: the synchronous strategy and an idealistic conservative strategy. The idealistic conservative strategy is a lower bound on the execution time for all conservative algorithms (including both synchronous and conservative asynchronous). She shows that if overheads are ignored and all events have the same evaluation time, the idealistic conservative algorithm will perform at least as well as the synchronous algorithm. The two algorithms will perform identically for unit-delay timing. Since synchronous algorithms are generally simpler than conservative asynchronous algorithms, then with reasonable load balancing, one would expect a synchronous simulation to outperform a conservative asynchronous simulation if unit-delay timing is used.

Bailey and Lin [1993] extend this work to include four different synchronization strategies: synchronous strategy, the conservative asynchronous strategy, the optimistic asynchronous strategy, and the conservative optimal strategy. The conservative optimal strategy is an artificial strategy that uses knowledge of all events in the simulation to construct an optimal scheduling of events, with the constraint that messages on a given processor are evaluated in timestamp order. Two assumptions were made for all synchronization strategies to keep the analysis tractable. First, it is assumed that there is a fixed, positive time delay associated with each logic element. This precludes elements from having a delay of zero, which can occur in some simulators. This also precludes having different time delays for the same element, such as is found in RNL [Terman 1983]. Second, it is assumed that every evaluation element is on its own processor.

Bailey and Lin's first result shows that the synchronous strategy is slower than the conservative optimal strategy. Communication costs are assumed to be negligible in the synchronous simulation, eliminating the costs of maintaining a global event queue and synchronizing at the end of each time step. Next the conservative optimal strategy is shown to be faster than the conservative asynchronous strategy with null messages. Communications costs for the conservative asynchronous strategy are not assumed to be zero, although it is assumed that the presence of null messages does

not degrade the performance of the system by increasing resource contention in the communications structure or by taking evaluation time on the target processor. There are mixed results in comparing the synchronous and conservative asynchronous strategies. If the circuit is strongly connected, an unlikely situation for a logic simulation, then the synchronous strategy will be faster. If the fanout is limited, then the conservative asynchronous strategy may be superior.

The remaining results pertain to the Time Warp or optimistic strategy. The cost of saving state is ignored, as well as the cost of restoring state during roll backs. Other roll back costs, such as the cost of sending antimessages, are included. Under these assumptions, Time Warp with either aggressive or lazy cancellation outperforms the conservative optimal strategy. The issue of limited processors is also addressed. If all logic elements on a given processor are considered as a single process, then the above analysis holds. However, this means that progress is delayed until all inputs to the logic block are known, as opposed to each individual logic element. This can degrade performance. A similar problem occurs in Time Warp upon roll back. Under this assumption the entire logic block is rolled back instead of rolling back just the element which receives the antimessage. Without these restrictions, the above results cannot be proven; more research is needed to address these issues.

Thus using simple analytic models, Bailey and Lin have shown that the optimistic synchronization strategy is preferred, although several unrealistic simplifications were necessary in order to obtain these results. It would be nice to eliminate many of these simplifications to determine whether these conclusions hold given the complex factors involved with implementing each algorithm on real hardware.

5. CIRCUIT STRUCTURE AND TIMING GRANULARITY

The information inherent in the circuit being simulated and the input vectors used to exercise the circuit can have a large impact on the performance of the simulation algorithm. Circuit structure includes such aspects as circuit topology, circuit size, abstraction level, fanout, feedback, circuit type, and circuit activity.

Circuit topology refers to the interconnection pattern between circuit elements. The abstraction level is the underlying model assumed for individual elements (e.g., switch level, gate level, etc.). The circuit type classifies the circuit in terms of its design style and goals, distinguishing between combinational circuits and sequential circuits, clocked and self-timed circuits. Circuit activity is concerned with the dynamic nature of signal value changes—how frequently signals change value, number of simultaneous value changes, etc.

The interrelationships between circuit structure and the other factors are significant enough that it is difficult (if not impossible) to isolate the impact that circuit structure alone has on the performance of parallel simulation. For this reason, the impact of circuit structure will primarily be addressed in conjunction with the other factors rather than in isolation. An exception to this is circuit activity, which has received extensive study.

One of the best understood relationships among the factors affecting performance is that between circuit activity and timing granularity. Work in this area began by simply measuring circuit activity. More recently, formal models have been developed that relate circuit activity and timing granularity.

5.1 Circuit Activity

VLSI designers have long been interested in measuring activity in their circuits. Circuit activity has a broader interest than parallel logic simulation: for example, it affects power requirements in CMOS designs directly. In the early 1970's, Rattner instrumented a logic simulator to measure the average number of gates which were active during simu-

lation runs (personal communication). He found that, on average, approximately 2.5 percent of the gates were on the event queue at any given time during a simulation run.

A few years later, research in circuit activity increased due to its importance in event-driven parallel simulation. The focus changed from measuring the percentage of simulation elements on the event queue to the average number of simulation elements evaluated in the same time step.

Frank [1985; 1986] published a fairly extensive study of circuit activity as part of his work on a parallel data-driven logic simulation engine, the Fast-1. This simulation engine used an event-driven algorithm, so its potential speedup was influenced by the activity in the circuits. Frank did not directly measure circuit activity, but rather estimated the potential speedup of the parallel Fast-1 over a uniprocessor version by considering the number of instructions the sequential and parallel versions required. The ratio of the number of sequential instructions to the number of parallel instructions provided an upper bound on speedup and a rough estimate of the circuit activity. Using 13 circuits ranging in size from 78 to 20,300 transistors, he found potential speedups ranging from 4.1 to 192.1, with a mean of 49.5. The low values surprised Frank, and he was not optimistic about the potential for the parallel Fast-1 engine.

Soon after Frank's work, other experiments were performed using existing sequential simulators to consider the potential of parallel event-driven simulation. The metric used in these experiments is usually referred to as *circuit parallelism*, which is defined to be the average number of events executed per active simulation time step. Time steps in which no events are executed are ignored, since there is no overhead for skipping them in an event-driven simulator. Circuit parallelism provides an upper bound on the speedup one can obtain using a parallel, synchronous, event-driven simulator.

Wong et al. [1986] were the first to report actual circuit parallelism measurements. They used a gate- and switch-level simulator and measured the parallelism of five circuits ranging from 650 to 8000 transistors, using fixed-delay timing. The parallelism values ranged from 2.1 to 55 with an average of 18.6. They scaled these parallelism values to estimate the circuit parallelism of 100,000 component circuits. The scaled values ranged from 80 to 3,294 with an average of 1,279. In contrast to Frank, Wong et al. were optimistic about the potential for parallel simulation, based on the scaled parallelism values.

Soule and Blank [1987] and Soule [1992] were the first researchers to consider the impact of different abstraction levels on circuit parallelism. Four different abstraction levels were presented: instruction, behavioral, RTL, and gate. THOR, a multilevel, event-driven simulator, was used here to measure the idealized speedup of three circuits using the four abstraction levels. Two of the circuits (3400 and 5000 elements) were simulated at the gate level. A third circuit was simulated using a different functional simulator. The speedup measurements were obtained by simulating the event trace with an "ideal" parallel simulator having no cost for scheduling, no memory contention, and equal cost for event evaluation. For 1000 processors, the speedup was less than 10 for all but one circuit and abstraction level, where the speedup was near 100. Additionally, they found speedup to be relatively constant over all four abstraction levels, and element activity between 0.1% and 0.5% at any particular time point.

During the following two years, Bailey [1992a] and Bailey and Snyder [1988] presented additional circuit parallelism measurements using the switch-level simulator RNL. RNL models a transistor as a resistance in series with a voltage-controlled switch and provides timing estimates with 0.1 ns resolution [Terman 1983]. The nine circuits used in these measurements ranged from 200 to 61,600 transistors. The resulting circuit paral-

lelism values ranged between 2.8 and 23. The measurements in Bailey [1992a] included a different activity metric, the queue metric, which corresponds more closely with Rattner's early measurements. The queue metric measures the average length of the simulation queue. The values measured using the queue metric will be higher than those found using the average parallelism, since there will usually be additional elements on the event queue which are not executed in the current time step. Using the same nine circuits, they found, on average, between 0.22% to 8.9% of the nodes were on the queue at each time step. These values have much more variance than Rattner found in his measurements.

Additionally, Bailey [1992a] presents empirical evidence to demonstrate that circuit parallelism does not generally scale linearly with circuit size as was assumed in Wong's optimistic parallelism measurements. In one circuit family, the shift register, parallelism did scale almost linearly. For other circuit families, this was not the case. The parallelism does generally increase with circuit size, but it is not a simple linear function.

Rather than using circuit parallelism as the mechanism for defining activity, Briner [1988] and Briner et al. [1988] devised a new metric, the spanning metric, to estimate the potential for parallel simulation using the interdependence of model evaluations. Two additional sources of parallelism are measured with this technique. First, additional parallelism is measured because a signal change may cause more than one model evaluation due to fanout. If event handling is inexpensive compared to model evaluations, this is a more accurate measure of the parallelism available for simulation. Second, just because events happen at different times does not mean that there is a causal effect between them. Thus, events at different time steps may be processed in parallel if model evaluations caused by the earlier event do not impact the later event. By extracting causal data from a sequential simulation, Briner et al. estimated the parallel activity in three circuits, ranging in size from 700 to 15,000 transistors, and found values ranging between 4.7 to 19.5. They compared this to the circuit parallelism for the same circuits and found that the spanning metric provided 4 to 10 times more potential parallel activity than was found using circuit parallelism.

Thus there have been several studies of circuit activity over the past few years, with conflicting conclusions. There are several reasons for these differences. First, different researchers used different metrics for evaluating circuit activity. Second, different sequential simulators were used, having different model abstractions and different timing resolutions. Finally, different circuits were used for the various measurements, resulting in differences due directly to the structure of the individual benchmark circuits.

5.2 Timing Granularity

Logic simulation covers a broad spectrum of model representations, each with different timing granularities ranging from very fine-grained timing (such as 0.1 ns) to coarse-grained timing (such as zero-delay). Timing granularity can significantly impact simulator performance. Simulators using fine-grained timing attempt to model time accurately in the simulator. Often these simulators use a time resolution in the range of 0.1 ns or smaller. An additional issue in fine-grained simulators is whether a given element always has the same delay. For instance, some gate-level simulators may use a single delay for a given gate type, independent of the output value of the gate. Others have different delays depending on output capacitance and whether the signal is rising or falling. In transistor-level simulators with fine-grained timing, delay computations can be even more complex, resulting in a large number of different possible delays for a single signal. We will consider fine-grained timing to include both small time resolutions and a large number of possi-

ble delays in the circuit. As either the number of possible delay values decreases or the time resolution increases, we say that the timing resolution is coarser.

Unit-delay and zero-delay timing are at the extreme of the coarse-grained timing granularities. Both are quite common in logic simulations. Unit-delay timing assumes that every element has a propagation delay of one unit. Zero-delay, used for sequential circuits, is even coarser. Here, only functionality is preserved, with no attempt to measure timing.

5.3 Relating Circuit Activity and Timing Granularity

The effect of timing granularity on circuit activity has been investigated via both empirical studies and formal models. The first empirical study was an extension of Bailey's [1992a] earlier circuit parallelism results. A unit-delay simulator, SwitchSim, was used to measure the circuit parallelism of the circuits previously measured using RNL. SwitchSim, written by Frank, is based on the algorithms developed for the Fast-1 simulation engine. The circuit parallelism measured by the unit-delay simulator was always larger than that measured by RNL. The parallelism values ranged from 35 to 593 on circuits ranging in size from 200 to 61,600 transistors. These values were larger than the RNL measurements by factors ranging from 3.6 to 25.8.

Even though this work compared the effect of two different timing granularities on circuit parallelism, it failed to provide a good characterization of the relationship between timing and parallelism. However, Bailey [1992b; 1993] has since developed two formal models to compare the effects of time resolution on circuit parallelism. Both models begin with the same initial abstraction, a graph representing the execution of a given circuit. In the graph, nodes correspond to events in the simulation, and edges represent causality. It is assumed that no more than one change occurs at any instant in time; an infinite resolution clock is used for timing. It is also assumed that exactly one event causes a subsequent event. These two assumptions ensure that the graph is a tree. Edges in the tree are labeled with the delay between the event and its parent. Because of the infinite resolution clock, every event is in its own time step, and there is no circuit parallelism.

In order to investigate the relationship between parallelism and time resolution, each model has a mechanism for increasing the timing granularity of the simulation. In the time-based model, events are placed in time steps of larger resolutions by their simulation time, preserving causality constraints [Bailey 1992b]. This differs from the way in which simulators place events into time steps, but the resulting analysis is simpler. For example, consider the situation with three dependent events, the first one occurring at time 0, the second at time 1, and the third at time 4. If the simulation clock has a time base of 2, these events are placed in time steps 0, 2, and 4, respectively using the time-based model. Using this model, it can be shown that circuit parallelism is a nondecreasing function of time resolution. The parallelism found using unit-delay timing provides an upper bound on the circuit parallelism for all time resolutions.

In the second model, the delay-based model, events are placed in time steps according to the delays between events [Bailey 1993]. This corresponds more closely to the placement of events in actual event-driven simulators, so it is more realistic than the time-based model. In the above example, the last event is placed in time step 6 rather than time step 4 since the delay between it and its parent event is 3. Unfortunately, the results obtained from this model are more complex than those in the time-based model, and circuit parallelism is no longer a nondecreasing function of time base. There are instances where increasing the time base actually decreases parallelism. However, as the resolution increases, circuit parallelism tends to increase or remain constant. More precisely, Bailey

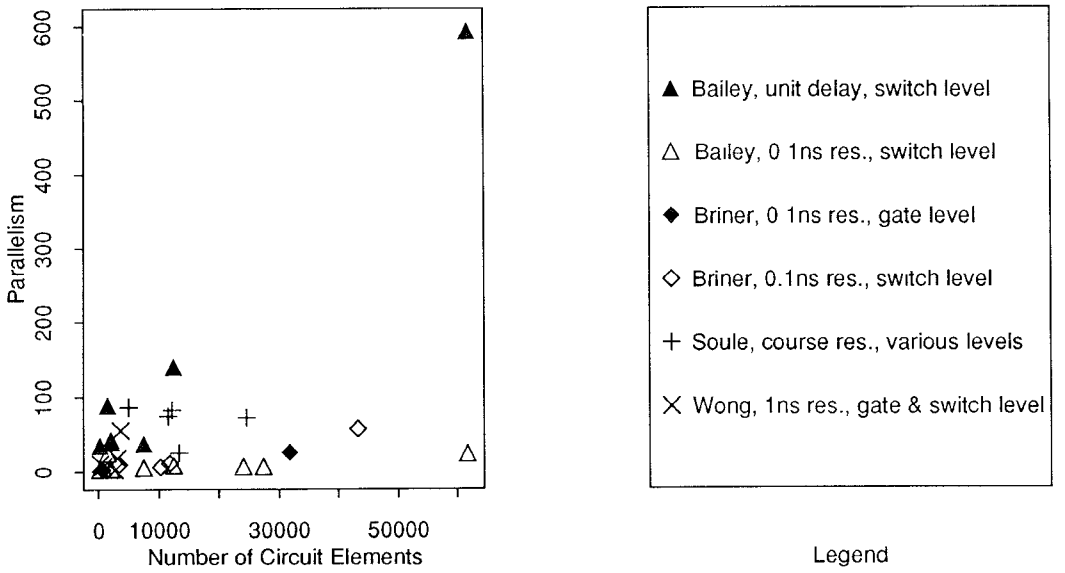

Figure 5. Average parallelism measurements.

considers the family of all circuits with the same unit-delay parallelism. The unit-delay parallelism provides an upper bound on the circuit parallelism for these circuits, over all time resolutions. The lower bound on the circuit parallelism for these circuits is a nondecreasing function of time resolution, which equals the unit-delay parallelism when the time resolution is sufficiently large. Bailey confirms the delay-based model predictions by effectively changing RNL's time resolution and measuring the resulting effects on circuit parallelism.

Figure 5 shows a composite graph of many of the parallelism results [Bailey 1992a; Briner 1990; Soule and Blank 1987; Wong et al. 1986]. Included are measurements from a variety of timing granularities (fine-grained, fixed-delay, and unit-delay), and from a variety of abstraction levels ranging from switch to functional. The largest parallelism value is obtained from the largest circuit using the coarsest timing granularity (unit-delay). In order to view the results for the smaller circuits better, an enlargement of the lower left quadrant of the figure is shown in Figure 6. Overall, the higher parallelism values result from measurements taken with coarser timing granularities, as predicted by Bailey's model. It is not clear whether the abstraction level makes a significant difference in the measurements, but the type of circuit (together with its input vectors) appears to make a significant difference.

Thus we have evidence that coarser-grained timing can result in dramatically higher levels of circuit parallelism, and there is a definite relationship between timing resolution and circuit activity. If higher levels of circuit activity imply better performance by parallel simulators, then coarser-grained timing appears more promising for parallel simulators. Note that the results do not cover all timing granularities, for example, zero-delay timing. Additionally, the range of parallelism measurements, even using the same simulator, indicate that circuit activity also depends on other aspects of circuit structure. Unfortunately, these studies shed little light on the exact nature of these relationships.

6. TARGET ARCHITECTURES

Parallel architectures are classically partitioned into MIMD (multiple-instruc-

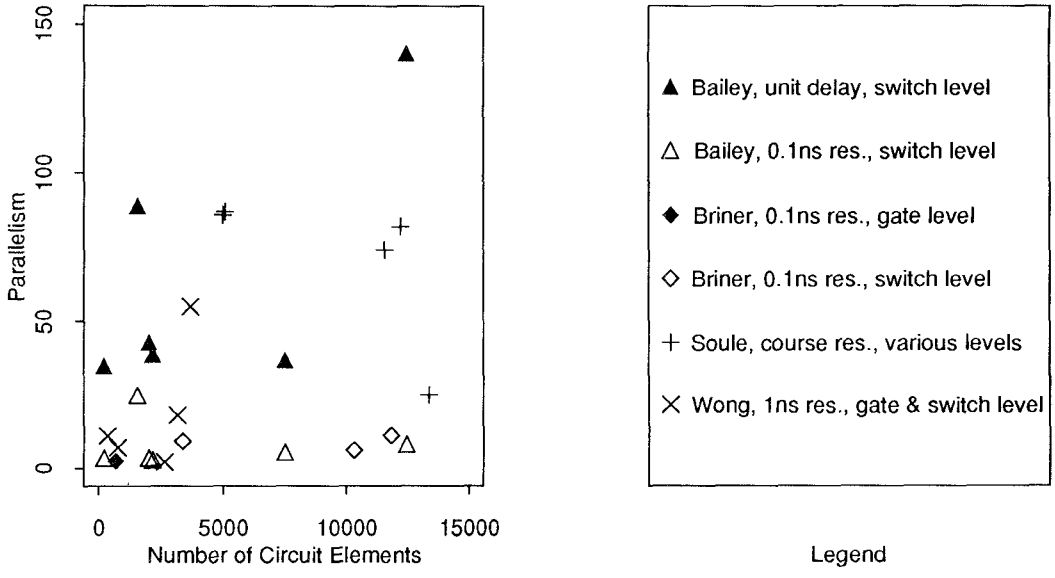

Figure 6. Average parallelism measurements for small circuits.

tion, multiple-data) machines, where each processor executes code independently, and SIMD (single-instruction, multiple-data) machines, where all processors execute the same instruction on independent data [Flynn 1966]. MIMD machines can be further classified as shared memory, where a common global address space is used to implement data sharing and synchronization between processors, or distributed memory, where communications is via explicit messages.

In SIMD architectures, processors execute instructions synchronously in lockstep. Processors may be programmed to avoid computing during a step if desired. Since all processors must perform the same instruction, only one type of gate is modeled at a time. A table lookup is often performed to help mitigate this restriction. If there are many types of models (as is the case in hierarchical system descriptions), simulation performance will be greatly diminished. Processors are often connected by a grid which allows neighboring processors to communicate quickly with each other. If nonadjacent processors must communicate, the messages must be routed through other processors or a global router. This is typically more expensive than nearest-neighbor communication. Most logic simulations are not limited to nearest-neighbor communication, complicating partitioning and mapping.

Shared-memory MIMD architectures utilize a common global address space to communicate between processors. Small-scale parallelism is typically implemented via a bus architecture, in which processors contend for access to a single physical memory located on the bus. These machines exhibit a uniform memory access time, independent of the processor or memory address. When the number of processors is large, a general-purpose interconnection network is used to communicate between memory modules associated with each processor. In these machines, memory access times are nonuniform, and depend upon whether the referenced address is local or remote. Communication between processors is relatively fast, usually on the order of microseconds. Depending on the interconnection network, however, contention

can be a problem. On a bus-based architecture with common memory and local caches, contention for the bus, false sharing, and protocol overhead can be very expensive. On machines which have nonuniform memory access, one of the partitioning goals is to avoid excessive communication to slower remote memory.

In distributed-memory MIMD architectures, the memory associated with each processor is local to that processor, and communication between processors is handled via explicit messages. These machines are typically constructed using a scalable topology, such as a mesh, torus, or hypercube. Message latencies can be long relative to functional evaluation times. If a signal being transmitted from one processor to another happens to be one of the circuit's synchronization signals, simulation performance can be seriously degraded [Briner 1990].

A parallel execution platform that has become increasingly popular is a network of workstations. Generally similar in style to distributed-memory MIMD machines, these platforms have several unique features. Their communication capabilities are strongly influenced by the fact that message delivery is via a general-purpose network. This implies significantly longer message latency. Also, multiple users are often executing programs while tightly coupled multicomputers often are dedicated resources.

In addition to general-purpose machines, there have been a number of special-purpose architectures proposed and built to implement parallel logic simulation [Blank 1984; Goering 1988]. Unlike general-purpose machines, these engines typically restrict the type of simulation that can be performed. Usually a single synchronization mechanism is employed, and only limited modeling levels are available. Many industrial companies have built logic simulation engines. For example, the Yorktown Simulation Engine [Denneau et al. 1983; Pfister 1986] and EVE [Beece et al. 1988] were designed at IBM; the MARS accelerator was designed at AT & T [Agrawal and Dally 1990]; NEC built HAL [Takasaki et al. 1986]; Fujitsu developed the SP [Saitoh 1988]; and Zycad Corporation manufactures an entire line of machines. Additionally, several logic simulation engines have been proposed and/or prototyped in universities; the Munich Simulation Engine [Hahn 1989] and a modified data flow architecture [Mahmood et al. 1992] are two of these. In this survey we will not focus on logic accelerators, although many of the issues discussed here also pertain to the effectiveness of these engines.

7. PARTITIONING AND MAPPING

The placement of circuit elements on the processors of a parallel machine can greatly affect the simulation of a VLSI system. One goal of partitioning elements for parallel simulation is to adjust the balance of computation among processors by assuring that each processor has useful work. The most common technique attempts to achieve load balance by ensuring that processors have a nearly equal number of components. However, this technique assumes that all components are equally active. Both Soule and Blank [1987] and Briner et al. [1988] have shown that a circuit's activity is usually uneven during simulation and varies over time. Further, it is difficult to know a priori which parts of the circuit will be active concurrently. Some researchers have performed a preliminary simulation to detect circuit behavior, providing more information for partitioning [Briner 1990; Chamberlain and Henderson 1994; Maanjikian and Loucks 1993]. Others have investigated the feasibility of dynamically adjusting the partition, allowing the simulator to adjust to circuit activity [Kravitz and Ackland 1988; Nicol and Reynolds 1985].

Another goal in placement is to reduce communication, which can represent a major performance bottleneck. Channels may become congested. Communication requires message-handling time and additional event-scheduling time. It also stresses the synchronization algorithm;

as a signal crosses processor boundaries, the synchronization mechanism must ensure that the signal is properly handled. Per message synchronization costs are low for synchronous simulation but can be high for the asynchronous techniques. In optimistic algorithms, the probability of a roll back is proportional to the probability of a message being received [Briner 1990]. In conservative algorithms, with more communication channels the likelihood of deadlock is higher, or, in deadlock avoidance algorithms, additional null messages must be sent [Soule and Gupta 1992].

Finally, mapping is related to communication. Mapping allocates partitions to processors. On machines with different interprocessor communication times, it is best to place frequently communicating partitions closer to reduce message latency and congestion. This problem has not been thoroughly investigated but has received some attention [Davoren 1989; Nandy and Loucks 1992].

7.1 Partitioning to Reduce Communication and Synchronization Costs

The most actively pursued area in partitioning has focused on reducing communication and synchronization overhead. In order to account for load balance, most partitioning research that focuses on communication ensures an equal number of gates are assigned to each processor. We illustrate a number of the more common algorithms using the example circuit of Figure 7, a two-bit full adder.

Levendel et al. [1982] present a partitioning method based on strings. The algorithm follows a primary input to a fanout gate and selects one of the fanout gates to add to the string. The process continues to a primary output. The gates on the string are placed on the same processor. If nodes remained unassigned, one is randomly selected to start a new string until no more nodes remain. In the example of Figure 7, a possible string associated with input *a* includes gates 1, 4, 5, and 7. The string associated with input *b* then includes only gate 3, since

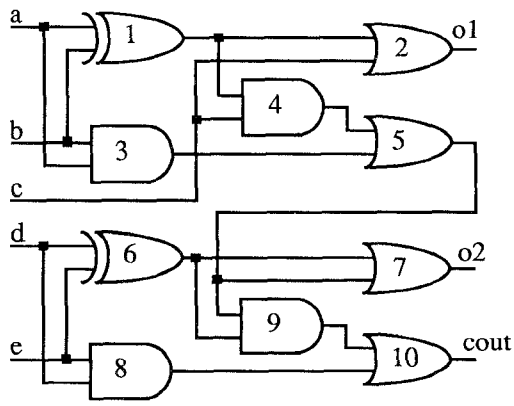

Figure 7. Partitioning example circuit.

the fanout of gate 3 is already assigned to another string. The string of input *c* is gate 2; the string of input *d* is gates 6, 9, and 10; and the string of input *e* is gate 8. For a two-processor simulation, assigning strings *a* and *b* to processor 1 leaves half of the gates for processor 2. The resulting partitioning is illustrated in Figure 8(a). Note that the resulting partitioning is highly dependent upon the choices of fanout gates used to build the strings and the ordering in which the strings are constructed. The algorithm is fast and ensures that at least one fanout gate will be on the current processor. However, the algorithm fails to reduce communication of closely related components significantly. Agrawal [1986] extends this algorithm to account for timing delays of gates in the circuit so that more concurrency may be exploited.

Smith et al. [1987] introduce fanin and fanout cones to improve the problem of communication. A cone of gates is generated by processing the gates in rank order. Each gate has a cone consisting of the set of gates which are affected by the output of the gate. Once the cones are built for all gates, the gates driven by primary inputs are evenly assigned to processors. After the primary inputs have been assigned, a gate is randomly selected. The gate's cone set and the union of cone sets associated with all gates already placed on each processor are com-

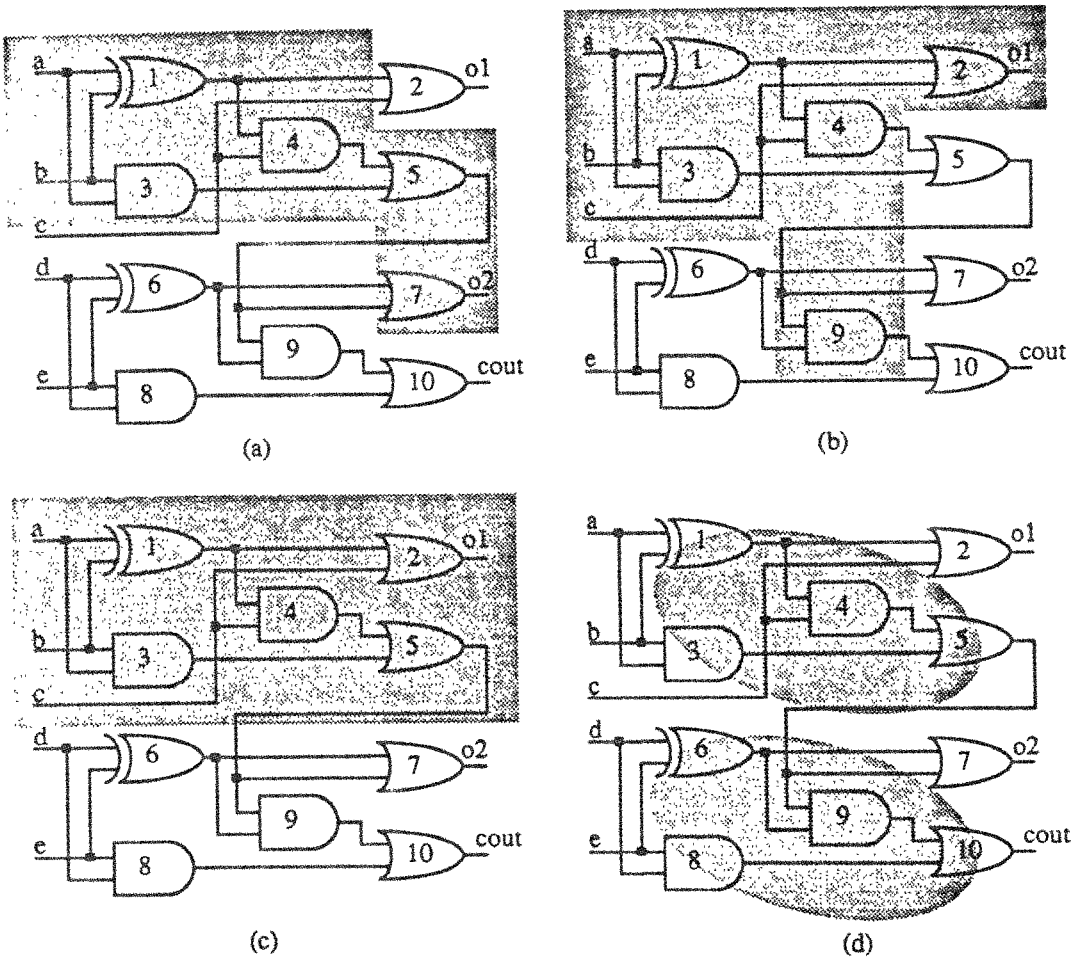

Figure 8. Example partitioning results.

pared. The processor which has the largest set in common is selected for the gate. After a processor is full, it is no longer considered for assignment. They report that this is fairly fast and reduces communication greatly when compared to a simple organization which places gates on the same processor if they are of the same rank.

We illustrate this algorithm using fanin cones, starting from the primary outputs and working back to the primary inputs. Table 6 shows the fanin cones for each of the gates in the example circuit. Starting from the primary outputs, we arbitrarily assign gate 2 to processor 1 and gates 7 and 10 to processor 2. Choosing gate 5 at random, we note that it has more overlap with the cones of gates 7 and 10, so it is assigned to processor 2.

Table 6. Fanin Cones for Example Circuit

Gate	Gates in Fanin Code	Gate	Gates in Fanin Code
1	1	6	6
2	1, 2	7	1, 3, 4, 5, 6, 7
3	3	8	8
4	1, 4	9	1, 3, 4, 5, 6, 9
5	1, 3, 4, 5	10	1, 3, 4, 5, 6, 8, 9, 10

Choosing gates 6 and 8 at random results in the same conclusion, assignment to processor 2. Since half of the gates are now on processor 2, the remaining gates are assigned to processor 1. This results in the partitioning illustrated in Figure 8(b).

Mueller-Thuns et al. [1993] believe that the cost of communication in a parallel and distributed environment is the major factor in obtaining speedups in parallel simulation. To reduce communication, they place entire cones on a processor without regard to whether the gates within the cone have already been placed on another processor (a gate may be in more than one cone). This leads to redundant evaluation of gates. The partitioning problem then becomes which cones to place on which processor rather than which gates to place on which processor. To reduce communication between cones, they use a depth-first search on the inputs to cones, forming a tree. Thus, leaf cones of the tree are likely to be on the same processor and have a parent cone on the same processor.

One common variation on cone partitioning is to form the partitions starting from latches in addition to primary outputs. This limits interprocessor communication to clock events, decreasing synchronization overhead. This technique is particularly attractive for zero-delay, rank-order simulation and oblivious algorithms.

Bisection and multiway partitioning are graph-partitioning algorithms which have been used extensively in placement and routing problems [Fiduccia and Mattheyses 1982; Kernighan and Lin 1970]. In both, the components are treated as nodes, and signals are treated as arcs in a graph. The goal is to divide the graph recursively into partitions to minimize arcs between partitions. A bisection of the example circuit is illustrated in Figure 8(c), with only a single arc connecting the two partitions. Briner [1990] shows that for optimistic time synchronization, bisection improves gate-level simulation greatly over random methods where the cost for functional evaluations is similar to communications costs. However, manual partitioning can be far superior when available. Figure 9 shows that for a transistor-level simulation using lazy cancellation, a good random partitioning is better than a bisection partitioning. This is true even when various sizes of moving time windows are used. Random partitioning performs better than the bisection partitioning because, for transistor-level simulations, model evaluations dominate the computation, and the randomness of the data provides better load balance. Load balance is achieved at the cost of communication and synchronization overhead (roll backs). The side effects of roll backs, repeated model evaluations, are diminished by the use of lazy cancellation.

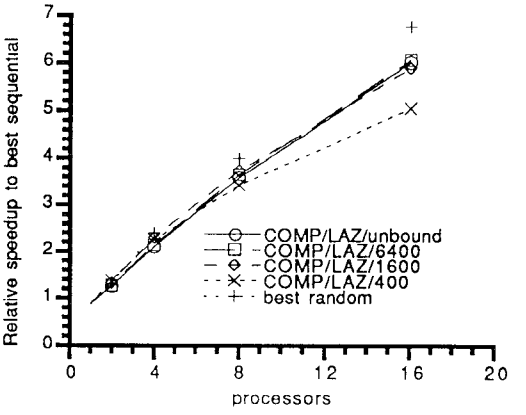

Figure 9. Random vs. bisection partitioning of a transistor network.

Sporrer and Bauer [1993] have performed a number of experiments on partitioning circuits. Using the rank-order techniques of Smith et al. [1987] (placing cuts at elements of the same rank), Sporrer and Bauer achieve good load balance, but nearly 30% of all signals must cross between processors. They also implemented a bisection technique based on Fidducia and Mattheyses [1982] which reduces the number of boundary signals to 10–20%. They present a modified clustering technique which goes through two phases: fine-grained clustering and

course-grained clustering. In the first phase, they use either a flip flop clustering algorithm or the corolla-partitioning technique of Dey et al. [1990] to form small clusters. Flip flop clustering places gates in small clusters near flip flops. Corolla partitioning detects reconvergent signals, creating what is known as a petal. Figure 8(d) shows two petals in the example circuit. Overlapping petals are then grouped into disjoint sets called corollas. In the second phase of the partitioning, the clusters are grouped together into larger clusters while minimizing the number of interconnections. The flip flop clustering technique reduces the number of signal crossings to around 4%, and corolla partitioning reduces signal crossings to around 1%.

Simulated annealing is a common method for reducing interconnections in physical design. Thus, it seems appropriate to consider it for reducing communication in logic simulation. Frank [1985] and Chamberlain and Franklin [1990] have both used simulated annealing to partition circuits prior to simulation. However, this work has been hindered by two factors. First, the time required to perform the simulated-annealing task is long relative to the serial execution time of the simulation. Second, the lack of information about circuit activity prior to simulation limits the ability to formulate an effective cost function to drive the simulated-annealing algorithm. Some performance predictions for circuits partitioned with simulated annealing are presented in Section 8.

7.2 Partitioning to Improve Load Balance

The easiest and fastest partitioning technique is random partitioning, in which elements are randomly assigned processors [Chamberlain and Franklin 1990; Frank 1985; Kravitz and Ackland 1988; Smith et al. 1987]. This ensures good load balance. If a portion of the circuit (e.g., an ALU in a CPU) is active, that portion of the circuit is distributed (e.g., bit-slices of the ALU) across the processors for simulation rather than concentrated on a single processor (which may be the result in a partitioning algorithm that stresses a reduction in communication). Smith et al. [1988] show that if model evaluations take significantly more time than communication, random partitioning does a much better job of ensuring concurrency than cone partitioning. However, if the cost of communication is of the same order as functional evaluation, closely related elements need to be on the same processor to avoid communication and synchronization overhead.

Wong and Franklin present an algorithm which attempts to minimize communication while maintaining processor balance [1987b]. The basic method is to use an undirected graph with vertices to represent gates and edges to represent interconnections between gates. A vertex is selected for each processor such that all selected vertices are at least some distance D away from each other. Subsequent vertices are added to each processor in a breadth-first, round-robin fashion. Partitioning the circuit of Figure 7, we start with gates 3 and 8 initially on processors 1 and 2, respectively. In round 2, gate 5 is added to processor 1 (since it is adjacent to gate 3), and gate 10 is added to processor 2 (being adjacent to gate 8). In round 3, gate 1 is added to processor 1, and gate 6 is added to processor 2. In round 4, gate 7 is added to processor 1, and gate 9 is added to processor 2. Finally, in round 5, gate 4 is added to processor 1, and gate 2 is left for processor 2. The resulting partitioning is the same (in this case) as the strings algorithm and is illustrated in Figure 8(a). Analytic comparisons with random partitioning show that this heuristic partitioning is up to twice as effective for several circuits on a bus-based architecture. However, later related work shows that for a hypercube executing a synchronous algorithm, random partitioning outperforms frequently the heuristic partitioning, while the heuristic shows better potential for use with the optimistic asynchronous algorithm [Chamberlain and Franklin 1990; 1991].

The above algorithms all make the implicit assumption that the computational load associated with individual gates is consistent across the circuit. This assumption is not valid, however, since the frequency of evaluation of individual gates can be significantly different. To address this issue, Manjikian and Loucks [1993] focus on redundant computation in cone partitioning by extending the process to do presimulation to get better information about load balance. Rather than counting the number of components in a cone for maintaining load balance, they measure the activity of each cone and any overlapping subcones from presimulations. With this information, they then perform iterative improvement [Sanchis 1989] that attempts to minimize the difference between the total activity per processor (including repeated activity) and the total activity in a uniprocessor simulation. To avoid putting all cones on the same processor, we may only move cones from larger cone sets to smaller ones, and once moved, they cannot be moved again.

The use of presimulation to characterize gate activity is supported in a study by Chamberlain and Henderson [1994], who measured the predictive power of presimulation on a number of circuits. They showed that, at least for sufficient numbers of random input vectors, presimulation is an excellent predictor of subsequent gate activity.

Another technique for handling load balancing is to adjust the partitions dynamically. Nicol and Reynolds [1985] use histories to determine when to repartition. Using Bayesian decision processes, they develop a model to decide when the cost of repartitioning is worth the expected improvement in concurrency. For repartitioning they suggest using a min-cut (bisection) with clustering algorithm.

Kravitz and Ackland [1988] performed studies of the simulations of two circuits and found that dynamic partitioning of a transistor-level simulation on a distributed-memory machine was not effective. Theoretically, dynamic partitioning could be more effective than static partitioning, but their results showed marginal gain even if partitioning was decided with an omniscient scheduler. However, the measurements were performed using relatively small circuits (about 10,000 transistors), and it is unclear whether their conclusions hold for larger circuits.

7.3 Mapping to Reduce Communication Latency and Congestion

A related problem is mapping partitions to processors. On machines with uniform memory access, mapping is not of concern. However, for machines which have nonuniform memory access, the proximity of partitions can affect message delays and congestion. Few investigations have been attempted in this area.

Using the hierarchical structure of a design, Davoren [1989] defines locality trees as an approximation technique for reducing communication between processors. A locality tree uses a circuit's hierarchical definition to define a tree of components. These components are then arranged within the tree so that those components which communicate are mapped closer together. One should note that physically connected components may not necessarily communicate with each other but just pass information through them; thus, proximity alone is insufficient and may not necessarily be available early in the design.

Returning to the example two-bit adder of Figure 7, a hierarchical decomposition might place one bit of the adder on processor 1 and the other bit of the adder on processor 2. This results in the partitioning illustrated in Figure 8(c). Note, it is not surprising that this is the same result as bisection, since the underlying strategy of the two algorithms is similar (i.e., group-connected components together). Davoren's results show performance increases of 208% when scaling from a network of 8 to 64 transputers for a circuit of 1060 gates using a conservative algorithm.

Nandy and Loucks [1992] study a number of factors including load balance,

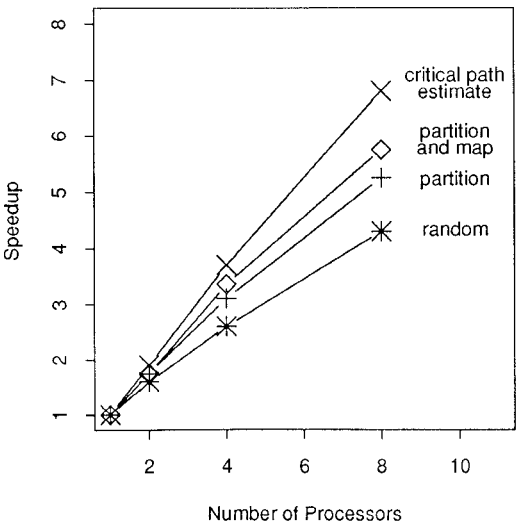

Figure 10. Random, iterative improvement, mapping, and optimal partitioning of a gate-level simulation.

partitioning and mapping. They use an iterative improvement technique similar to Manjikian and Loucks [1993] but instead focus on the cost of nets crossing partitions rather than the cost of redundant computations. On a network of transputers using a conservative asynchronous paradigm, they achieve speedups on the average of 4.5 on very small circuits. By adding weights to the communication arcs, they are able to map the partitions on the transputer to account for its nonuniform message rate. Figure 10 displays the theoretical optimal performance of the simulator using a technique similar to Briner [1988] versus iterative improvement partitioning, iterative improvement partitioning with mapping, and random partitioning. Of particular note is that performance is increased an additional 10–20% when the target architecture's communication limitations are considered.

7.4 Discussion

Research clearly shows that effective partitioning and mapping depends on a number of factors: the computation/communication ratio, the circuit activity (especially variation in location of activity), and the target architecture's communication capabilities. In static partitioning, a random partitioning works well for both conservative and optimistic synchronization algorithms when functional-evaluation time dominates communication time. However, when functional evaluations are not as costly, bisection, corolla, clustering, and Wong and Franklin's [1987b] heuristic outperform random partitioning for optimistic synchronization techniques because communication and synchronization overheads are reduced. Because different circuits and simulation models were used, it is difficult to compare the different techniques. While many authors report the cut size, cut size alone does not guarantee better performance because circuit activity may vary, leading to load imbalance.

For synchronous simulation, cone-partitioning methods have lead to good performance. Iterative improvement on cone partitioning with presimulation data has reduced the number of redundant computations while cone levelization reduces communication. When available, manual partitioning has performed extremely well. To date, simulated annealing has not proven effective both due to a lack of a good cost function and excessive execution requirements. One possible improvement is to use presimulation data in the cost function formulation.

The problems of mapping to nonuniform memory access machines and of dynamic and incremental partitioning have not been thoroughly investigated. Mapping has improved the performance of transputer-based parallel simulation. Techniques for detecting when to repartition have not been well defined, and the overhead costs associated with repartitioning may overshadow any gain in performance. However, this mechanism may be the only way to handle a circuit's dynamic behavior. Research is needed to understand the tradeoffs in these areas better.

Finally, it should be noted that most results in the literature have ignored the

cost of simulation preparation: if it takes longer to partition a circuit than simulate it, what use is there in accelerating the simulation? Most of the partitioning techniques discussed are linear in the size of the circuit. However, is this really fast enough? For regression testing, when large numbers of input vectors are used, partitioning costs are not as significant as they are early in the design phase when only a few vectors may be needed. To speed partitioning, it is also possible to use parallel processing [Nandy and Loucks 1993].

8. PERFORMANCE MODELS

Now that the factors that impact the performance of parallel logic simulation have been described, we present methods to assess the likelihood of achieving acceptable performance in realistic scenarios (i.e., ones that include the impact of multiple factors). There have been a number of performance models developed to investigate parallel simulation, including both analytic and trace-driven modeling. Even though the final say in performance is the measurement of a real implementation, performance models have distinct advantages: they are considerably faster to develop than full-fledged implementations, and they can characterize performance over a larger range of parameters than is practical (or even possible) with implementations alone.

8.1 Analytic Modeling

An analytic performance model of the synchronous simulation algorithm was originally developed by Wong and Franklin [1987a], who considered a special-purpose architecture dedicated to logic simulation. It was extended by Chamberlain and Franklin [1988] to include hierarchical component modeling on a general-purpose, distributed-memory machine. The performance model assumes a synchronous algorithm executing on a hypercube architecture. Circuit components are statically allocated to processors and do not migrate during execution. If the simulation runs for B *busy ticks* (simulation time points that have one or more events that need to be evaluated), the execution time for P processors is expressed as

$$R_P = B \cdot [\max(t_{CPU}, t_{COMM}) + t_{SYNC}]$$

where t_{CPU} is the average processor execution time per busy tick; t_{COMM} is the average communications time per busy tick; and t_{SYNC} is the time required to synchronize at the end of a busy tick. The processor and communications times are combined using a maximum operator to reflect the fact that computation and communications can occur simultaneously (provided the memory bandwidth is not saturated). The synchronization time must be added to the maximum since its execution does not overlap with either event processing or data communications.

The variables in the above expression can be expressed as a function of simulation properties (e.g., event counts, functional-evaluation counts, etc.) and architectural properties (e.g., functional-evaluation time, message formulation time, message delivery time, etc.). A complete derivation and explanation for the model is given in Chamberlain and Franklin [1990]. Using technology parameters typical of an nCUBE 2 machine and input parameters measured from serial simulations of three benchmark circuits, the model provides predicted performance for a number of distinct partitioning strategies [Chamberlain and Franklin 1986; 1990].

Figure 11 shows the predicted speedup over a single-processor implementation, R_1/R_P, for one of the benchmark circuits. Curves are presented for three different circuit partitionings: random (R), Wong and Franklin's heuristic (H), and simulated annealing (S). Random partitioning performs the best, because performance is limited by load balance rather than communications requirements. The heuristic and simulated-annealing partitioning algorithms attempt

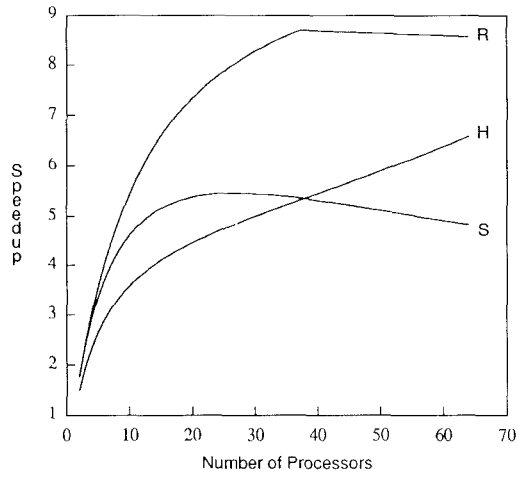

Figure 11. Predicted synchronous speedup.

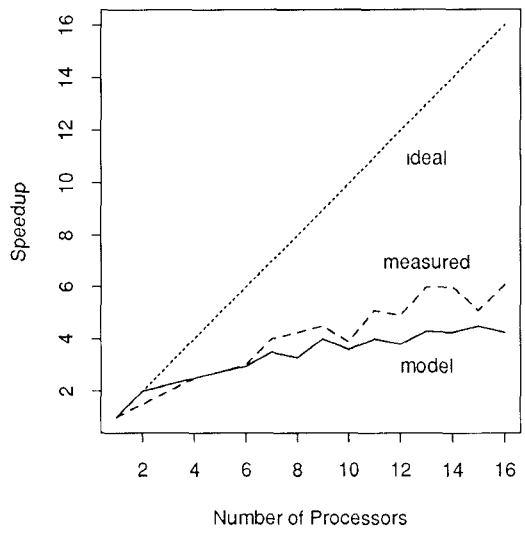

Figure 12. Synchronous model evaluation.

to decrease communications at the expense of poorer load balance.

Agrawal and Chakradhar [1992] present a statistical model of the processor workload in synchronous simulation. The model is based on circuit activity, and can be viewed as a refinement of the t_{CPU} term of the Chamberlain and Franklin model. Agrawal and Chakradhar define circuit activity as the mean number of logic elements that must be evaluated per busy tick. The number of functional evaluations on a processor is modeled as a binomially distributed random variable. The average processing time per busy tick (t_{CPU}) is then derived as the expectation of the maximum of P samples from this distribution.

They evaluate the model by comparing the predicted performance with observed performance on several production VLSI circuits. The results of one of these comparisons is shown in Figure 12. The dotted line represents ideal speedup; the dashed line shows observed performance with up to 16 processors [Agrawal 1986]; and the solid line shows the predicted performance using the model. The modeled results match the observed performance closely for all of the circuits they investigated.

The above models address synchronous algorithms, but are not useful for asynchronous techniques. The next section describes models that can be used to predict the performance of asynchronous algorithms as well.

8.2 Trace-Driven Modeling

There have been two efforts at trace-driven simulation modeling to predict the potential performance of parallel logic simulation. Briner's [1988] spanning metric determines performance of a near-optimal parallel simulator running on shared-memory architectures with uniform and nonuniform memory access times. Chamberlain and Franklin's [1991] architectural simulation model predicts the execution time of several optimistic algorithms executing on a hypercube machine.

8.2.1 Spanning-Metric Model

Briner [1988] extends the spanning metric (Section 5) for measuring potential parallelism on an infinite number of processors to the more realistic case of a finite number of processors. The first extension considers uniform-access, shared-memory machines. The second considers nonuniform memory access and partitioning. After capturing the causality of a sequential RNL simulation, a

trace-driven simulation is performed on the data. The trace-driven simulation uses techniques from the task assignment literature [Gonzalez 1977] to account for scheduling on a finite number of processors. Measurements of three small circuits show that near-perfect speedups are possible up to the maximum speedup of the unlimited processor case.

Briner extends the model of task assignment to consider the problem of data accesses in a machine with a nonuniform memory architecture. His results show that data assignment is a difficult problem and that poor assignment reduces potential parallelism by at least a factor of two below the uniform memory access model.

8.2.2 Architectural Simulation Model

Chamberlain and Franklin [1991] developed an architectural simulation of several parallel simulation algorithms executing on a hypercube architecture. First, a circuit description and random set of input vectors are simulated on a standard uniprocessor, and trace data is collected. Second, the circuit is statically partitioned using the heuristic of Wong and Franklin [1987b]. Third, the trace data, partitioned circuit description, architecture description, and parallel-algorithm description are input into an architectural simulator that performs a trace-driven simulation.

Data is collected for three different time synchronization strategies. The first is an optimal algorithm that is not attainable in practice but bounds the potential performance. The second two are both optimistic asynchronous algorithms using aggressive cancellation: the basic time warp (TW) algorithm and a moving time window (MTW) variant. The optimal bound is found by determining the most heavily loaded architectural resource (either processor or communications link) and observing that this resource must be on the execution's critical path. The execution time is then modeled as the workload performed by this resource. Note that this execution time is optimal only for a *given* partitioning. A different allocation of circuit components to processors will yield a different optimal performance.

Figure 13 compares the speedup predicted for two different benchmark circuits. The most notable conclusion from these curves is the extreme variation in performance from one circuit to the next. Using TW, benchmark 1 (Figure 13(a)) has speedup less than unity (i.e., the parallel execution is slower than the serial execution) for 32 processors, while benchmark 2 (Figure 13(b)) has near-optimal performance. One also notices a phenomenon in which the speedup curves have more than one local maxima. This phenomenon has also been observed in TW simulations for other application domains [Lin and Lazowska 1991].

Note that the MTW algorithm will occasionally perform noticeably better than the TW algorithm, and rarely perform appreciably worse. To some extent, it is successful in decreasing the frequency (and resulting performance degradation) of roll back.

8.3 Discussion

Both analytic and trace-driven performance models allow us to draw a number of conclusions. First, parallel logic simulations can have significant performance advantages over serial simulation. Thus, there is the potential for parallel logic simulation to have a real economic impact in the design automation community. Second, the interrelationships between the factors and the combination of factors that would ensure good performance are still not well understood. For example, widely varying performance is predicted for optimistic algorithms for different processor populations, different circuits, and different partitionings. Third, a good partitioning algorithm is essential, as poor partitionings degrade performance seriously.

What remains to be accomplished is to understand better how individual factors (or combinations of factors) impact per-

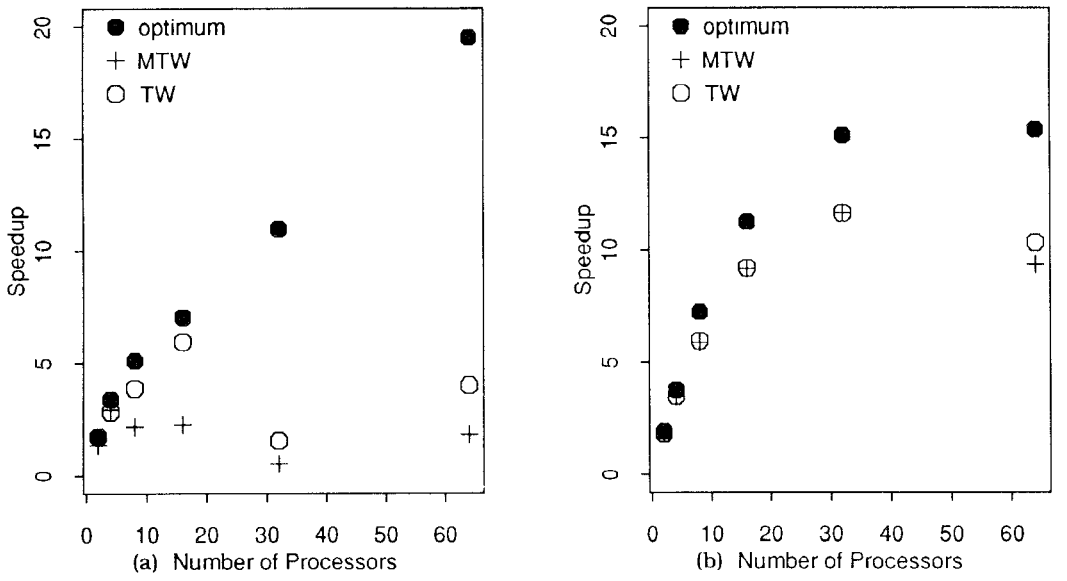

Figure 13. Architectural simulation.

formance. The analytic models need to be refined to rely less on empirical input data. For example, the circuit activity in Agrawal and Chakradhar's model must currently be measured from simulation executions. If it could be reasonably predicted from the circuit structure and/or input vectors, conclusions could be drawn about the appropriateness of the synchronous algorithm for a particular instance without having previously run the simulation. The trace-driven models have, to date, only been used to predict the execution time for a small number of data points, and several factors impacting performance differ from one data point to the next. Controlled experiments using trace-driven models need to vary only a single factor at a time to help quantify the impact each factor has on performance.

9. IMPLEMENTATIONS

In addition to formal studies and performance modeling, a number of parallel simulators have been implemented using a variety of synchronization strategies. Implementations give the most accurate feedback on actual performance. Unfortunately, different implementations are often difficult to compare, since they tend to vary across multiple factors. We present the implementations via their synchronization strategy and compare them at the end of the section.

9.1 Oblivious Simulators

As well as being used in hardware accelerators, the oblivious approach has been considered for use on general-purpose, parallel computers. Kravitz et al. [1991] addressed the feasibility of mapping the unit-delay simulator COSMOS [Bryant et al. 1987] onto SIMD computers. This implementation was able to take advantage of both course-grained parallelism from many simultaneous element evaluations and fine-grained parallelism used to evaluate the Boolean equations representing the behavior of each element. Because of static scheduling, the equations can be rank ordered, and the evaluation of each rank comprises an atomic operation. A simple model of processors and interconnection structures is used in the parallel COSMOS implementation, placing significant scheduling demands on the compiler.

Two relatively small circuits (of 20,000 and 43,000 transistors) were used to evaluate the performance of parallel COSMOS. The *effective parallelism*, the average number of Boolean operators which can be evaluated concurrently, provides an upper bound on the speedup that can be obtained using the oblivious strategy and includes the redundant evaluations of elements whose inputs have not changed. The effective parallelism for one circuit achieved a maximum value around 2500, while the other circuit reached a lower maximum near 400. These high speedups are not generally attainable in practice, due to communications and scheduling constraints. To measure actual speedups, a prototype parallel COSMOS simulator was implemented on the Thinking Machines CM-2 [Hillis 1986]. This simulator used a simple SIMD model, and thus did not exploit the full power of the Connection Machine's instruction set. Using this parallel simulator, one circuit simulation ran at twice the speed of sequential COSMOS, while the other only ran at half that of sequential COSMOS. These circuits are relatively small, and the authors expect better performance with larger circuits.

Jun et al. [1990] proposed a variant of the oblivious strategy for use in gate-level simulation, where some care was taken to minimize memory usage and time spent in simulating feed-back loops. It uses a segmented waveform relaxation method in conjunction with bit-wise operations. The segmented waveform relaxation method is used to divide the simulation interval into a number of subintervals, each of which can be represented by a single word. Each element's state during a subinterval is represented by a single word, and bit-wise operations can be performed on the entire word to propagate behavior throughout the circuit. Logical shifts are used to represent gate delays. Parallel computation is used in two ways: (1) to evaluate gates at the same rank order and (2) to pipeline the subintervals in the waveform evaluations. While the authors report good performance on the ISCAS combinational benchmarks [Brglez and Fujiwara 1985] using a single processor, no results were available for the parallel implementation.

9.2 Synchronous Simulators

Several parallel logic simulators use synchronous algorithms. Soule and Blank [1988] implemented a synchronous simulator on a 16-processor Encore machine. Speedups ranging from 4 to 9 were obtained on circuits ranging in size from approximately 500 to 5000 using 15 processors.

Mueller-Thuns et al. [1990] have synchronous implementations for two different abstraction levels of logic circuits: switch level and gate level. For switch-level simulation, they preprocess the original "flat" transistor netlist to create an acyclic set of strongly connected components. These components are then used as tasks and are available for scheduling on a parallel processor. As in the work of Jun et al., parallelism is obtained by evaluating strongly connected components in parallel during the same time unit if they have the same rank order, and by evaluating components in different ranks in parallel if the time for the component with lower rank is greater than that for the component with higher rank. Using these techniques, they obtain speedups ranging from 2.21 to 7.56 on circuits having approximately 10,000 to 34,000 transistors using 8 processors of an Encore multiprocessor.

For their gate-level circuits, clocks at latches are used as the synchronization mechanism [Mueller-Thuns et al. 1993]. The circuit is partitioned according to the input cones of latches (or primary outputs). All gates on a path between primary inputs (or outputs of other latches) and primary outputs (or inputs to latches) are on the same processor, creating a simple combinational circuit. Once all primary inputs and outputs of latches are known, a zero-delay simulation is performed which allows updating of the primary outputs and latches. After these

have stabilized, signals are propagated to other processors, beginning another cycle. This algorithm has been implemented on both shared-memory and distributed-memory parallel processors. Using 8 processors, they obtained speedups ranging from 4.6 to 4.9 on the distributed-memory machine and 4.8 to 5.0 on the shared-memory machine using three example circuits from the ISCAS-89 benchmarks [Brglez et al. 1989]. The circuits ranged in size from just under 18,000 to almost 24,000 gates.

Bataineh et al. [1992] implemented a synchronous event-driven algorithm for a gate-level simulation on the Cray Y-MP. They used both vectorization and parallel processing to process all events in a given time step in parallel. For this technique to perform well, they report that there needs to be 32 events available for each of the 8 processors at each time step. Using two example circuits, a combinational circuit from the ISCAS-85 benchmarks and a linear-feedback shift register, they found that there were enough events in each time step and obtained speedups of 36 over a scalar version running on the Cray for the ISCAS circuit (2406 gates), and 52 for the shift register (size not reported). Thus the combination of vector and parallel processing resulted in good speedups.

9.3 Conservative Asynchronous Simulators

As a part of Soule and Blank's [1988] parallel implementation of a gate-level and register-transfer-level simulator, they implemented a conservative asynchronous algorithm using deadlock detection and recovery. They obtained speedups of 7 to 11 for circuits of size 100 to 5000 elements on a 16 processor Encore. They found that this mechanism worked more efficiently than the synchronous algorithm when more than 10 processors were utilized.

Despite the early promise of conservative algorithms, Soule and Gupta [1992] show that large circuits cause frequent deadlock and that deadlock resolution requires 50–80% of the total execution time. With deadlock avoidance, they estimate that 20 to 100 null messages would have to be sent for each real message. Thus, they conclude that deadlock detection and recovery are more likely to succeed. In a study to detect the sources of deadlock in the conservative algorithm specific to digital simulation, Soule and Gupta found four sources of deadlock: feedback to registers, multiple paths with different delays, the simulation algorithm's order of updating nodes, and desensitized nodes. They are able to perform a number of optimizations with this information. By exploiting element behavior, deadlock can be avoided. For example, since a register's output only changes on a clock, the time of the next output can be predicted, taking advantage of lookahead. Combinational circuits that are desensitized (e.g., an AND gate with a zero input) produced no events, allowing signals to other inputs to be ignored until the critical signal changes. Passing information about the lookahead of the sensitized signal directly to affected nodes avoids the overhead of passing a message through the desensitized combinational circuit. Grouping elements together into a single element reduces both event handling and the probability of a deadlock. Speedups of 16–32 on an ideal multiprocessor with 64 processors are predicted for circuits of size 5000–25,000. However, these results are self-relative. Even with these optimizations, they conclude that for most circuits the conservative algorithm will be 2.8 to 3.5 times slower than a synchronous algorithm.

Su and Seitz [1989] considered a number of variants of the conservative asynchronous strategy on distributed-memory MIMD machines. All variants used deadlock avoidance; they differ in minimizing the number of null messages by sending them in a "lazy" fashion. They used Intel iPSC machines to measure the speedup of a 1376-gate multiplier network. Using a 128-node iPSC/1 they obtain speedups of approximately 10, and on a 16-node iPSC/2 they obtain speedups of 2 for the best variant. They conclude that if the

time to process null messages is comparable to the time to process actual messages, then the conservative asynchronous algorithm will not perform well when the number of processors is small.

To reduce the overhead of null messages, they study a conservative asynchronous algorithm which groups multiple circuit elements together into a single element. Elements in a single group are simulated via a sequential algorithm, while different groups use the conservative asynchronous strategy. They use a 1067-gate self-timed FIFO circuit to evaluate this algorithm, and achieve much more promising results using a small numbers of processors, especially if the circuit is manually partitioned.

Ackland et al. [1985] and Kravitz and Ackland [1988] have implemented a MOS timing simulator on a message-based multiprocessor using a conservative asynchronous algorithm. While their timing simulator provides continuous waveforms, it uses discrete-event simulation techniques to communicate between subcircuits. Speedups from 7 to 20 are reported on a 60-processor system. One thing to note with these results is that the timing simulator requires much more computation per element evaluation than a logic-level simulator, and thus communication overhead is less critical.

Subramanian and Zargham [1990] present a parallel, demand-driven simulator for a Sequent Balance. In demand-driven simulation, results are requested at the output nodes, and requests propagate back to the input nodes. Extensive memory is required as events propagate to the inputs. The parallel version takes advantage of global memory by allowing processors to use a central task queue which supports better load balance. Using 5 processors, randomly devised combinational circuits of 500 gates show speedups of 2 to 3 times a sequential implementation. Larger simulations on larger parallel processors are needed to make a fair comparison with the Chandy-Misra-Bryant approach.

Chung and Chung [1990] implemented three conservative algorithms for gate-level simulation on the Connection Machine CM-2, an SIMD architecture. The first is a synchronous algorithm, using a global clock. The second is a null-message implementation of the conservative asynchronous algorithm with no lookahead. The third algorithm adds lookahead to the asynchronous algorithm. Two circuits were used to measure the performance of these algorithms; a 16-bit combinational multiplier from the ISCAS-85 benchmarks (2406 gates) and a 32-bit array multiplier (8000 gates). All three of these algorithms outperform Time Warp on the Connection Machine, in part due to the complexity of the Time Warp implementation. Among the three algorithms, the algorithm using lookahead outperforms the other two in most cases, although when the number of inputs is small, the synchronous algorithm performed best for the smaller multiplier. Due to memory limitations in the Connection Machine, large circuits may require the synchronous algorithm, since its memory requirements are less than those for the conservative algorithms.

Arvind and Smart [1991] implemented ELSA, a framework for event-driven logic simulation in a nonuniform memory environment. This framework supports both conservative and optimistic synchronization strategies, as well as a hybrid strategy. The conservative synchronization strategy uses deadlock avoidance with null messages. To lessen the cost of sending and processing null messages, their number is minimized by eliminating redundant ones. ELSA is implemented in Occam2 on a transputer-based multiprocessor. Three multiplier circuits are studied. Speedups over a uniprocessor are not reported, because the minimum number of transputers used is 8 or 16, depending on the circuit. However, the execution gains appear fairly dramatic—one circuit executes 3.6 times faster on 40 transputers than on 16.

9.4 Optimistic Simulators

There have been several VLSI system simulators implemented using optimistic

techniques. The earliest simulator was built by Arnold, who reports on a LAN implementation [Arnold 1985; Arnold and Terman 1985]. The internals of RSIM [Terman 1983] were changed for parallel simulation and modified for a fixed-point implementation because there was no floating support on the machines used. One processor acted as a master, keeping a database and user interface. The other 6 nodes worked as slaves, performing the simulation. Intermittently during simulation, each processor checkpoints its internal state consisting of the event queue, all nodes, all transistors, and simulated time. To avoid excessive memory usage and CPU overhead, checkpoints are not taken on each event. Thus roll backs may have to go back further than the timestamp of a late-arriving event, leading to repeated computations. Arnold's simulator performs no fossil collection and is unable to simulate large systems. In the simulation of a 64-bit adder with seven processors, a speedup of 4.2 is obtained.

Chung and Chung [1989] implemented a Time Warp simulator on the CM-2. Processors on the machine represent either an event or a gate. For the gate processors, a table lookup alleviates the problem of evaluating different types of gates on a SIMD processor. However, table lookup allows only a limited set of models with few inputs. While it is reasonable to break some large gates into smaller gates, some gates cannot be easily broken into simple AND, OR, and flip flop gates. All event processors can perform the same operations. Because some processors are dedicated to event handling and others to modeling, the Connection Machine is effectively losing 50% of the processors in each step of the simulation.

Their general algorithm computes the local time for each gate, evaluates those events at this time, evaluates active gates, propagates any events to event processors, calculates global virtual time, and performs garbage collection of past events. They take advantage of primitive operations of the machine to optimize the selection of the next event. After all the events have been evaluated, new events can be added to the segment of the appropriate gate using the enumerate feature of the parallel machine. All of the operations take $O(\log P)$ instructions on each simulation cycle, where P is the number of processors.

They conclude that they have maximized the data parallelism of the Connection Machine by using an asynchronous algorithm for a synchronous machine. The largest shortcoming of the work is the lack of utilization of the event processors. To increase event processor utilization, they describe a technique called Lower Bound of Rollback, which reduces the number of past events required by the usual GVT calculations. No absolute results are presented which compare performance of a sequential simulation and the parallel algorithm, although the later results presented in the previous section show that the conservative strategies outperform this optimistic version.

Briner et al. [1991] present another version of the Time Warp algorithm on a BBN GP1000, a nonuniform access, shared-memory machine. The costs of distributing events, saving state, and rolling back are kept proportional to the sequential simulation's event overhead by using incremental state saving. Unlike Arnold's implementation, the state of the queue, gates, and other data structures are saved incrementally on each event. While the cost of processing an event increases with the parallel implementation, there is no overhead beyond the effects of the event, unlike Arnold's implementation, where the cost of saving state is proportional to the size of the circuit. If large portions of the circuit are inactive, Arnold's simulator wastes excessive time saving unchanged information, and as seen in Section 5, much of a circuit is inactive.

Briner et al. find that lazy cancellation removes some of the interdependence of events and improves performance. However, partitioning is still critical because an event from another processor may roll back all the gates on a proces-

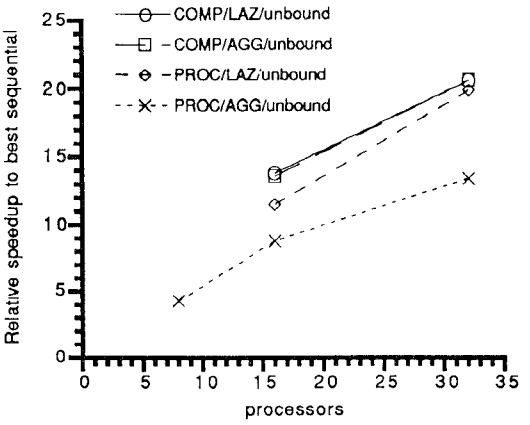

Figure 14. Single vs. multiple partitions per processor.

sor. These effects are mitigated if gates on a processor are synchronized independently rather than aggregately. With independent gates, a roll back of one gate will not cause unrelated gates to roll back. However, if independent gates are kept on the same processor, additional overhead is incurred because the same node may be replicated on the same processor. Figure 14 shows speedup curves for component (COMP) and processor (PROC) synchronization using both aggressive (AGG) and lazy (LAZ) cancellation techniques. It appears more important to reduce synchronization than to reduce repeated operations. Interestingly, what may reduce synchronization costs for one synchronization algorithm may increase it for another. Su and Seitz found that for conservative algorithms it is necessary to group elements to minimize the overhead of null messages.

Briner [1990] also considers some other techniques to speed up the simulation. In order to reduce communication latency for high-fanout nodes, he uses a repeated-message structure which allows high-fanout nodes to be sent in a parallel, rather than sequential, fashion. This technique reduces the synchronization effect large fanout nodes (e.g., the clock) have on the simulation. Another technique used to improve performance is to use a variant of moving time windows. By reducing the time skew between processors, the probability of roll back is lessened. Absolute speedups ranging from 7 to 23 are obtained on 32 processors using circuits ranging from 666 to 31,680 elements.

As discussed in the previous section, the ELSA framework includes optimistic as well as conservative synchronization strategies on a transputer system [Arvind and Smart 1991]. Its optimistic strategy uses lazy cancellation. When compared to the conservative strategy, the optimistic strategy performed at least as well in all instances except one, and generally performed better, especially on larger circuits. No application-specific optimizations have been included for these results. ELSA also has a hybrid conservative/optimistic mode, where the coarser parts of the simulation use the conservative strategy, and the finer-grained ones use the optimistic strategy; no results are presented for this model.

Manjikian and Loucks [1993] extend the synchronous algorithm of Mueller-Thuns et al. to allow optimistic simulation before all inputs to a processor are known. However, they do not propagate the output results until the input values are actually known. Thus, no interprocessor rollback is necessary. Given that, for the circuits studied, between 2 and 20% of the latches change, significant speedups are possible. Because zero-delay simulation is used, the states of internal nodes do not have to be preserved. Further, portions of the circuit simulated with earlier-arriving inputs that do not interact with later-arriving inputs will not be resimulated. In reporting the results of their measurements, Manjikian and Loucks compare the performance of the parallel simulator running on a network of 7 Sun Sparcstation IPCs against a pure sequential algorithm. They use circuits from the ISCAS-89 benchmarks, and obtain speedups of 2.5 to 4.1. Thus, these results reflect accurately the impact of parallel simulation and are impressive because they have obtained approximately 50% effi-

ciency in an environment with a large communications overhead.

Bauer et al. [1991] and Sporrer and Bauer [1993] have implemented a logic simulator using the Time Warp synchronization strategy on both shared and distributed machines. As in Briner et al. [1991], they also used incremental state savings to reduce the overhead of the simulation. The simulator was first prototyped on a Sequent, but used a message-based communication structure since the final target machine did not have shared memory. Subsequently, they implemented the simulator on a network of Sun Sparcstation 2's. Measurements for both the Sequent and Sun implementations compare the performance of the parallel simulator running on a single node with the same version running on multiple nodes. With this comparison, they obtain speedups from approximately 2 to 3 on a subset of the ISCAS-89 benchmarks using a 5-processor Sequent, and 5.2 to 5.7 using a network of 8 Sparcstations, with some circuits improving with additional nodes. In order to compare this work with Manjikian and Loucks, we note that Bauer and Sporrer [1993] report that the parallel version of their simulator required 70% more overhead than the sequential version. If we factor this into the speedups reported above, we find that the speedups using a network of 8 Sparcstations now ranges from 3.1 to 3.4, much closer to those results reported by Manjikian and Loucks [1993].

Kim and Chung [1994] implemented on optimistic algorithm using a token-passing mechanism for GVT maintenance. They report varying performance, with speedups ranging from 2 to 15 on an 80-processor BBN Butterfly.

9.5 Discussion

There have been a number of implementations spanning all of the major synchronization algorithms. What synchronization strategy is preferred? This is somewhat difficult to answer, since there are so many differences in the implementations, even within the same

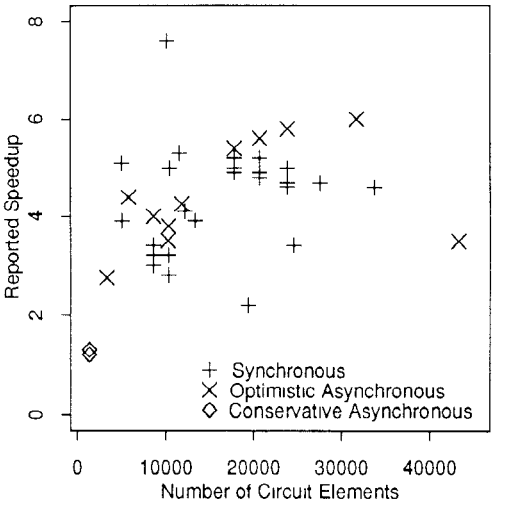

Figure 15. Comparing reported speedups from different synchronization strategies (8 processors).

synchronization algorithm. It is hard to determine accurately to what extent the synchronization algorithm is impacting performance and to what extent other factors are influencing the outcome.

If the oblivious strategy is used, speedups appear quite promising. However, the speedups show comparisons of the cost of the oblivious strategy for multiple processors over the oblivious strategy for a single processor. Considering the low values obtained in circuit parallelism measurements (Section 5), most of this computation is redundant, although the low cost of element evaluation may mitigate this somewhat.

Now we consider ways to compare different synchronization strategies for event-driven simulation. First, Figure 15 shows a number of speedups that span the synchronization strategies. Data for eight processors were used, since this was a common number of processors reported in the literature. These results were obtained from different researchers, using different abstraction levels, different timing models, and different example circuits. Additionally, some researchers compute speedups by comparing the parallel implementation with a good sequential one, while others compare it with a

parallel implementation running on a single processor. Thus only general comparisons can be made. However, no conservative asynchronous implementation resulted in good speedup, while both optimistic and synchronous implementations performed quite well. Fine timing granularities were used in some of the optimistic implementations that obtained good speedup, while all of the synchronous implementations used coarse timing. From these data points, it appears that a synchronous implementation is sufficient if coarse timing models are used; if fine timing is used, an optimistic implementation should strongly be considered. As the number of processors increases, synchronous implementations may suffer; optimistic implementations are likely to scale better.

More precise comparisons between synchronization strategies cannot be made from Figure 15; to do this we consider small numbers of similar implementations. Several researchers have implementations using two synchronization strategies. Chung and Chung [1989; 1990] used SIMD machines as a platform for all three strategies, and found that the conservative asynchronous implementation with lookahead is generally preferred on SIMD machines. ELSA is implemented for both conservative and optimistic asynchronous strategies, and the optimistic strategy performed better generally, especially on large circuits. Soule and Gupta [1992] implemented both synchronous and conservative asynchronous strategies, and found that the synchronous strategy was preferred. This may be simply due to the relative coarseness of their timing granularity, or may be a more general result.

Finally, several researchers have used a subset of the ISCAS-89 benchmarks to report their findings. We have taken measurements reported by Mueller-Thuns et al., Manjikian and Loucks, and Sporrer and Bauer [1993] to compare their results on one of these circuits, s38584. The speedup curves are shown in Figure 16. Here we show speedup versus the number of processors to provide a

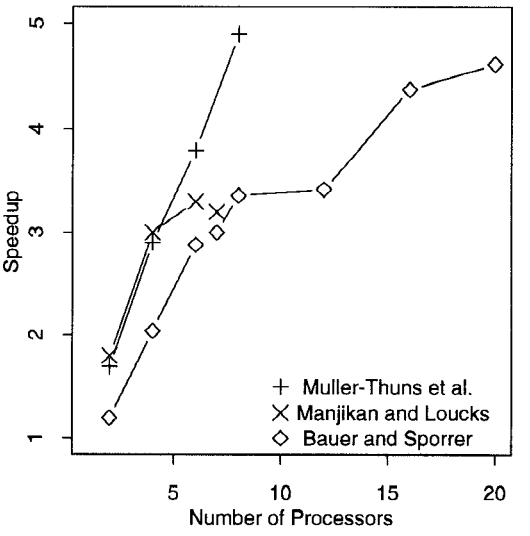

Figure 16. Speedups for the ISCAS-89 s38584 Benchmark.

feeling for the way speedups change as the number of processors increase. Additionally, we have factored the Bauer and Sporrer results to reflect the fact that their single-processor comparison uses 70% more overhead than a true sequential implementation, as in Section 9.4. The Mueller-Thuns et al. results appear more promising, although this may be due in large part to the fact that they use a shared-memory implementation, and the other two groups use workstation networks. Of the two optimistic approaches, Manjikian and Loucks achieve better performance than Bauer and Sporrer for the data they report, but Bauer and Sporrer achieve even better performance by using additional workstations. It is not clear whether the decrease in performance from 6 to 7 processors will continue in the Manjikian and Loucks results, or whether it is simply a local minimum.

In conclusion, it is very hard to compare data from different implementations. However, we have seen some interesting trends. If coarse timing is used, synchronous algorithms perform well on small numbers of processors, and vector machines can also provide good speedups.

If fine timing is used, the optimistic strategy appears to be most promising. Both Briner et al. and Bauer and Sporrer have found that incremental state saving is critical in minimizing overhead due to state saving. Briner [1990] also finds it important to reduce the synchronizing effect of large fanout nodes and to reduce the granularity of synchronization by having smaller but more partitions per processor. Moving time windows is necessary to keep processors from excessive roll backs.

On SIMD architectures, conservative algorithms outperform optimistic ones. On MIMD machines, if there is increased computation per element evaluation, the conservative strategy can perform well, as seen with the MOS timing simulator. As with general discrete-event simulation applications, good lookahead is critical to performance of conservative logic simulation.

10. CONCLUSIONS

Parallel simulation of VLSI systems is feasible if implementations can account for the five factors which impact parallel simulation: synchronization algorithm, circuit structure, timing granularity, target architecture, and partitioning and mapping. A number of techniques for studying the five factors have been presented: formal models, performance models, empirical studies, and implementations. Formal models are useful in obtaining general results, although simplifying assumptions are generally necessary to make the analysis tractable. Performance models are usually more constrained than formal models, but they can include more details, thus making the results more realistic. Empirical studies can focus both types of models as well as help guide choices in implementations. Implementations, in the form of prototypes, provide the most accurate feedback on performance, but it is extremely difficult to draw general conclusions from an implementation because of the large number of design decisions which impact performance.

By reviewing the results of all of the analysis techniques, we are able to draw some general conclusions concerning the five factors. Circuit structure has clearly a dramatic influence on the performance of parallel simulations. This has been observed both through studies on circuit activity as well as in actual implementations. The exact relationship between circuit structure and simulation performance is not well understood. We cannot accurately predict which types of circuits will perform well using parallel simulators and which will perform poorly.

Timing granularity and synchronization algorithms also affect performance. Clearly, coarser timing increases circuit parallelism. Thus for synchronous simulations, coarser timing granularities are the most promising, assuming event evaluation times are relatively constant. It is unclear whether parallel simulation using finer timing will have acceptable performance. Asynchronous algorithms have the potential of allowing these simulations to approach or exceed the performance obtained using coarser granularities, but the overheads needed to run these algorithms may be too costly. At this time the optimistic algorithms seem to perform better than the conservative asynchronous algorithms, both in formal models and in implementations. The oblivious strategy is another alternative. With its redundant computations, it is unclear whether this strategy can outperform the discrete-event simulation techniques which better adjust to circuit activity.

Very little data is available for understanding the relationship between target architecture and simulation performance, although architecture affects the performance of all parallel programs clearly. Good results with nearly 50% efficiency have been obtained over relatively slow networks, while some implementations on tightly coupled systems perform poorly.

Finally, partitioning and mapping are very important. To date, static partitioning has received the most attention. The best automatic partitioning algorithm

will depend on the relative importance of load balancing, communication and synchronization costs, and the target architecture. When the communication/computation ratio is small, random partitioning performs well; when it is large, more formal techniques which reduce communication and synchronization costs are necessary. Less is known about the impact of dynamic partitioning, and whether the gain in performance is worth the overhead associated with repartitioning.

Where do we go from here? Obviously, obtaining good performance from parallel logic simulation is nontrivial. While we have seen some performance gains over sequential simulation, no implementation has emerged as the clear winner. Progress has been made in understanding some of the relationships between the factors affecting performance, but this has not yet led to breakthroughs. Additional research is needed to make real progress. New or hybrid synchronization algorithms may be needed. If we hope to exploit the full potential of parallel logic simulation, we must understand the relationships among all five factors and how they impact performance. We need to find the function f, such that

$$\begin{aligned} &\textit{performance} \\ &\quad = f(\textit{synchronization}, \textit{structure}, \\ &\qquad \textit{timing}, \textit{architecture}, \\ &\qquad \textit{partitioning}). \end{aligned}$$

This function can be used to determine the circumstances under which parallel logic simulation is viable, and to guide the design, implementation, tuning, and use of parallel logic simulators.

Since this function f is clearly complex, it is important initially to isolate as many of its variables (the five factors) as possible. Trace-driven models can help here. While to date, trace-driven models have only been used to predict the execution time for particular sets of circumstances, they can be used to isolate these variables and begin to understand their impact. Later, they can also be used to explore interactions among different factors.

Another step in better understanding the effectiveness of parallel simulations is to encourage more cooperative research. There have been many implementations of parallel logic simulators. However, few if any of these implementations are available to users for experimentation. Moreover, the set of circuits used to characterize the efficiency of the implementations is quite diverse. The ISCAS benchmarks have been used by a number of researchers, but these circuits are relatively small. Larger circuits are needed for testing; parallel simulators are not generally necessary for small circuits, and results may be skewed by considering only small circuits. Furthermore, there are a large number of variables in circuits which are not fully characterized in the ISCAS benchmarks: timing granularity, model abstraction, and input vectors are three important ones. A well-chosen set of benchmarks, synthetic or real, is critical for both implementations and trace-driven modeling; otherwise erroneous conclusions may result.

Real progress has been made in the area of logic simulation, but there is much still to be done. With cooperation of the logic simulation research community, progress can continue, and fast, efficient parallel logic simulators running on general-purpose machines can become a reality, allowing a significant reduction in the design cycle while allowing more thorough system design testing.

REFERENCES

ACKLAND, B. D., AHUJA, S. R., LINDSTROM, T. L., AND ROMERO, D. J. 1985. CEMU—A concurrent timing simulator. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. IEEE, New York, 122–124.

AGRAWAL, P. 1986. Concurrency and communication in hardware simulators. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. CAD-5*, 4 (Oct.), 617–623.

AGRAWAL, V. D. AND CHAKRAADHAR, S. T. 1992. Performance analysis of synchronized iterative algorithm on multiprocessor systems. *IEEE Trans. Parall. Distrib. Syst. 3*, 6 (Nov.), 739–745.

AGRAWAL, P. AND DALLY, W. J. 1990. A hardware logic simulation system. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 9*, 1 (Jan.), 19–29.

ARNOLD, J. 1985. Parallel simulation of digital circuits. MS thesis, Massachusetts Institute of Technology, Cambridge, Mass.

ARNOLD, J. AND TERMAN, C. 1985. A multiprocessor implementation of a logic-level timing simulator. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. IEEE, New York, 116–118.

ARVIND, D. K. AND SMART, C. R. 1991. A unified framework for parallel event-driven logic simulation In *Proceedings of the SCS Summer Simulation Conference*. SCS, San Diego, Calif., 92–97.

BAILEY, M. L. 1993. A delay-based model for circuit parallelism. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 12*, 12 (Dec.), 1903–1912.

BAILEY, M. L. 1992a. How circuit size affects parallelism *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 11*, 2 (Feb), 208–215.

BAILEY, M. L. 1992b. A time-based model for investigating parallel logic-level simulation. *IEEE Trans. Comput. Aided Des. Integr Circ. Syst. 11*, 7 (July), 816–824.

BAILEY, M. L. AND SNYDER, L. 1988. An empirical study of on-chip parallelism. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*. New York, 160–165.

BAILEY, M. L. AND LIN, Y.-B. 1993. Synchronization strategies for parallel logic-level simulation. *Int. J. Comput. Simul. 3*, 3, 211–230.

BATAINEH, A., ÖZGÜNER, F., AND SZAUTR, I. 1992. Parallel logic and fault simulation algorithms for shared memory vector machines. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. IEEE, New York, 369–372

BAUER, H. AND SPORRER, C. 1993. Reducing rollback overhead in Time-Warp based distributed simulation with optimized incremental state saving. In *Proceedings of the 26th Annual Simulation Symposium*. IEEE Computer Society Press

BAUER, H., SPORRER, C., AND KRODEL, T. H. 1991. On distributed logic simulation using Time Warp. In *Proceedings of the International Conference on Very Large Scale Integration VLSI 91*. North-Holland, Amsterdam, 127–136.

BEECE, D. E , DEILBERT, G., PAPP, G., AND VILLANTE, F 1988. The IBM engineering verification engine. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*. IEEE, New York, 218–224.

BILLOWITCH, W. D. 1993. IEEE 1164: Helping designers share VHDL models. *IEEE Spectr. 30*, 6 (June), 37.

BLANK, T. 1984. A survey of hardware architectures used in computer-aided design. *IEEE Des Test Comput. 1*, 4, 21–39

BRGLEZ, F. AND FUJIWARA, H. 1985. A neutral netlist of 10 combinational benchmark circuits and target translator in Fortran. *IEEE International Symposium on Circuits and Systems*. IEEE, New York.

BRGLEZ, F., BRYAN, D., AND KOZMINSKI, K. 1989. Combinational profiles of sequential benchmark circuits. In *Proceedings of the 1989 IEEE International Symposium on Circuits and Systems* IEEE, New York.

BRINER, J. V., JR. 1990. Parallel mixed-level simulation of digital circuits using virtual time. Ph.D. thesis, Duke Univ., Durham, N.C.

BRINER, J. V., JR. 1988. A framework for analyzing parallel discrete event simulation. In *Proceedings of the Computer Measurement Group*. CMG, Dallas, Tex., 180–185.

BRINER, J. V., JR., ELLIS, J. L., AND KEDEM, G. 1988 Taking advantage of optimal on-chip parallelism for parallel discrete event simulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. IEEE, New York, 312–315.

BRINER, J. V., JR., ELLIS, J. L., AND KEDEM, G. 1991. Breaking the barrier of parallel simulation of digital systems. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. IEEE, New York, 223–226.

BRYANT, R E 1977. Simulation of packet communication architecture computer systems. Tech Rep MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge, Mass

BRYANT, R. E., BEATTY, D , BRACE, K., CHO, K., AND SHEFFLER, T 1987. Cosmos: A compiled simulator for MOS circuits. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*. ACM, New York, 9–16.

CHAMBERLAIN, R. D. AND FRANKLIN, M. A. 1991. Analysis of parallel mixed-mode simulation algorithms. In *Proceedings of the 5th International Parallel Processing Symposium*. IEEE, New York, 155–160.

CHAMBERLAIN, R. D AND FRANKLIN, M. A. 1990 Hierarchical discrete-event simulation on hypercube architectures *IEEE Micro 10*, 4 (Aug.), 10–20.

CHAMBERLAIN, R. D. AND FRANKLIN, M. A. 1988. Discrete-event simulation on hypercube architectures. In *Proceedings of the 1988 IEEE International Conference on Computer-Aided Design*. IEEE, New York, 272–275.

CHAMBERLAIN, R. D. AND FRANKLIN, M. A. 1986. Collecting data about logic simulation. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. CAD-5*, 3 (July), 405–412.

CHAMBERLAIN, R. D. AND HENDERSON, C. 1994. Evaluating the use of pre-simulation in VLSI circuit partitioning. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*. SCS, 139–146.

CHANDY, K. M AND MISRA, J. 1981. Asynchronous distributed simulation via a sequence

of parallel computations. *Commun. ACM 24*, 11 (Apr.), 198–206.

CHUNG, M. J. AND CHUNG, Y. 1990. Efficient parallel logic simulation techniques for the Connection Machine. In *Supercomputing '90*. IEEE Computer Society, Washington, D.C., 606–614.

CHUNG, M. J. AND CHUNG, Y. 1989. Data parallel simulation using Time-Warp on the Connection Machine. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*. IEEE, New York, 98–103.

DAVOREN, M. 1989. A structural mapping for parallel digital logic simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*. SCS, San Diego, Calif., 179–182.

DEY, S., BRGLEZ, F., AND KEDEM, G. 1990. Corolla based circuit partitioning and application to logic synthesis. Tech. Rep. TR90-40, MCNC Research Triangle Park, N.C.

DENNEAU, M., KRONSTADT, E., AND PFISTER, G. 1983. Design and implementation of a software simulation engine. *Comput. Aided Des. 15*, 3 (May), 123–130.

FIDUCCIA, C. M. AND MATTHEYSES, R. M. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*. ACM, New York, 175–181.

FLYNN, M. J. 1966. Very high-speed computing systems. *Proc. IEEE 54*, 1901–1909.

FRANK, E. H. 1986. Exploiting parallelism in a switch-level simulation machine. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. IEEE, New York, 20–26.

FRANK. E. H. 1985. A data-driven multiprocessor for switch-level simulation of VLSI circuits. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, Pa.

FUJIMOTO, R. M. 1990. Parallel discrete event simulation. *Commun. ACM 33*, 10 (Oct.), 30–53.

FUJIMOTO, R. M. 1989. Performance measurements of distributed simulation strategies. *Trans. Soc. Comput. Simul. 6*, 3 (July), 211–239.

GAFNI, A. 1988. Rollback mechanisms for optimistic distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*. SCS, San Diego, Calif., 61–67.

GOERING, R. 1988. Simulation accelerators used in CAD. *Comput. Des.* (Mar. 15).

GONZALEZ, M. J, JR.. 1987. Deterministic processor scheduling. *ACM Comput. Surv. 9*, 3, 171–204.

HAHN, W. 1989. The Munich simulation computer: Design principles and performance prediction. In *Hardware Accelerators for Electrical CAD*, T. Ambler, P. Agrawal, and W. Moore, Eds. Adam Hilger, Bristol, U.K.

HILLIS, W. D. 1986. *The Connection Machine*. MIT Press, Cambridge, Mass.

JEFFERSON, D. R. 1985. Virtual Time. *ACM Trans. Program. Lang. Syst. 7*, 3, 404–425.

JUN, Y.-H., HAJJ, I. N., LEE, S.-H., AND PARK, S.-B. 1990. High speed VLSI logic simulation using bitwise operations and parallel processing. In *Proceedings of the IEEE International Conference on Computer Design*. IEEE, New York, 171–174.

KERNIGHAN, B. W. AND LIN, S. 1970 An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J. 49*, 2, 291–307.

KIM, H. K. AND CHUNG, S. M. 1994. Parallel logic simulation using Time Warp on shared-memory multiprocessors In *Proceedings of the 8th International Parallel Processing Symposium*. IEEE Computer Society Press, 942–948

KRAVITZ, S. A. AND ACKLAND, B. D. 1988. Static vs. dynamic partitioning of circuits for a MOS timing simulator on a message-based processor. In *Proceedings of the SCS Multiconference on Distributed Simulation*. SCS, San Diego, Calif., 136–140.

KRAVITZ, S. A., BRYANT, R. E., AND RUTENBAR, R. A. 1991. Massively parallel switch-level simulation: A feasibility study. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 10*, 7, 871–894.

LEVENDEL, Y. H., MENON, P. R., AND PATEL, S. H. 1982. Special-purpose computer for logic simulation using distributed processing. *Bell Syst. Tech. J. 61*, 10, 2873–2909.

LEWIS, D. M. 1991. A hierarchical compiled code event-driven simulator. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 10*, 6 (June), 726–737.

LIN, Y.-B. AND LAZOWSKA, E. D. 1991. Processor scheduling for Time Warp parallel simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*. SCS, San Diego, Calif., 11–14.

LUBACHEVSKY, B. D. 1989. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM 32*, 1 (Jan.), 111–123.

MAHMOOD, A., BAKER, W. I., HERATH, J., AND JAYASUMANA, A. 1992. A logic simulation engine based on a modified data flow architecture. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. IEEE, New York, 377–380.

MAANJIKIAN, N. AND LOUCKS, W. M. 1993. High performance parallel logic simulation on a network of workstations. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*. SCS, 76–84.

MAURER, P. M. AND LEE, Y. S. 1994. Gateways: A technique for adding event-driven behavior to compiled simulations. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 13*, 3 (Mar.), 338–352.

MISRA, J. 1986. Distributed discrete-event simulation. *ACM Comput. Surv. 18*, 1, 39–65.

MUELLER-THUNS, R. B., SAAB, D. G., AND ABRAHAM, J. A. 1990. Design of a scalable parallel switch-level simulator for VLSI. In *Supercomputing '90*. IEEE Computer Society, Washington, D C., 615–624.

MUELLER-THUNS, R. B., SAAB, D. G., DAMIANO, R. F., AND ABRAHAM, J. A. 1993. VLSI logic and fault simulation on general-purpose parallel computers. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 12*, 3 (Mar.), 446–460.

NANDY, B. AND LOUCKS, W. M 1993. On a parallel partitioning technique for use with conservative parallel simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*. SCS, 43–51.

NANDY, B. AND LOUCKS, W. M. 1992. An algorithm for partitioning and mapping conservative parallel simulation onto multicomputers. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*. SCS, 139–146.

NICOL, D. M AND REYNOLDS, P. R., JR. 1985 A statistical approach to dynamic partitioning In *Proceedings of the SCS Multiconference on Distributed Simulation* SCS, San Diego, Calif, 53–56

PFISTER, G. F 1986. The IBM Yorktown simulation engine. *Proc. IEEE 74*, 6 (June), 850–860.

SAITOH, M. 1988. Logic simulation system using simulation processor (SP). In *Proceedings of the 25th ACM/IEEE Design Automation Conference*. IEEE, New York, 225–230.

SANCHIS, L. 1989. Multiple-way network partitioning. *IEEE Trans. Comput. 38*, 1 (Jan.), 62–81.

SHRIVER, E. J. AND SAKALLAH, K. A. 1992. Ravel: Assigned-delay compiled-code logic simulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. IEEE, New York, 364–368.

SMITH, R. J., II. 1986. Fundamentals of parallel logic simulation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. IEEE, New York, 2–12.

SMITH, S. P., UNDERWOOD, B., AND MERCER, M. R. 1987. An analysis of several approaches to circuit partitioning for parallel logic simulation. In *Proceedings of the 1987 International Conference on Computer Design*. IEEE, New York, 664–667.

SMITH, S. P., UNDERWOOD, B., AND NEWMAN, J. 1988. An analysis of parallel logic simulation on several architectures. In *Proceedings of the 1988 International Conference on Parallel Processing*. Penn State University Press, University Park, Pa., 65–68.

SOKOL, L. M., BRISCOE, D. P , AND WIELAND, A. P 1988. MTW: A strategy for scheduling discrete simulation events for concurrent execution. In *Proceedings of the SCS Multiconference on Distributed Simulation* SCS, San Diego, Calif., 34–42.

SOULE, L. P. 1992 Parallel logic simulation· An evaluation of centralized-time and distributed algorithms. Ph D. thesis, Stanford Univ , Stanford, Calif.

SOULE, L. AND BLANK, T. 1988 Parallel logic simulation on general-purpose machines In *Proceedings of the 25th ACM/IEEE Design Automation Conference* ACM, New York, 166–171

SOULE, L. AND BLANK, T 1987. Statistics for parallelism and abstraction levels in digital simulation In *Proceedings of the 24th ACM/IEEE Design Automation Conference* ACM, New York, 588–591.

SOULE, L. AND GUPTA, A. 1992 An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*. SCS, 129–138.

SPORRER, C. AND BAUER, H. 1993. Corolla partitioning for distributed logic simulation of VLSI-circuits. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*. SCS, 85–92.

SU, W -K. AND SEITZ, C. L. 1989. Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*. SCS, San Diego, Calif., 38–43.

SUBRAMANIAN, K. AND ZARGHAM, M. R. 1990. Parallel logic simulation on general-purpose machines. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. ACM, New York, 485–490.

TAKASAKI, S., SASAKI, T., NOMIZU, N , ISHIKURA, H , AND KOIKE, N. 1986. HAL II: A mixed level hardware logic simulation system. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. IEEE, New York, 581–587

TERMAN, C. J. 1983 Simulation tools for digital LSI design. Ph.D thesis, Massachusetts Institute of Technology, Cambridge, Mass

WONG, K. AND FRANKLIN, M. A 1987a Performance analysis and design of a logic simulation machine In *Proceedings of the 14th Annual International Symposium on Computer Architecture* IEEE. New York, 49–55.

WONG, K. AND FRANKLIN, M A. 1987b. Load and communications balancing on multiprocessor logic simulation engines In *Hardware Accelerators for Electrical CAD*, T. Ambler and W. Moore, Eds. Adam Hilger, Bristol, UK.

WONG, K. F , FRANKLIN, M. A., CHAMBERLAIN, R. D., AND SHING, B. L. 1986. Statistics on logic simulation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. IEEE, New York, 13–19.

Received August 1993, final revision accepted June 1994