*VLSI DESIGN*
1996, Vol. 4, No. 2, pp. 91–105
Reprints available directly from the publisher
Photocopying permitted by license only



# An Evaluation of Parallel Synchronous and Conservative Asynchronous Logic-Level Simulations

AUSIF MAHMOOD[a] and WILLIAM I. BAKER[b]

[a]*Computer Science and Engineering, University of Bridgeport, CT 06601;* [b]*Washington State University at Tri-Cities, Richland, WA 99352*

A recent paper by Bailey [1] contains a theorem stating that the idealized execution times of unit-delay, synchronous and conservative asynchronous simulations are equal under the conditions that unlimited number of processors are available and the evaluation time of each logic element is equal. Further it is shown that the above conditions result in a lower bound on the execution times of both synchronous and conservative asynchronous simulations. Bailey's above important conclusions are derived under a strict assumption that the inputs to a circuit remain fixed during the entire simulation. We remove this limitation and, by extending the analyses to multi-input, multi-output circuits with an arbitrary number of input events, show that *the conservative asynchronous simulation extracts more parallelism and executes faster than synchronous simulation in general.* Our conclusions are supported by a comparison of the idealized execution times of synchronous and conservative asynchronous algorithms on ISCAS combinational and sequential benchmark circuits.

*Keywords:* Parallel logic simulation, Distributed simulation, Conservative asynchronous simulation, Synchronous simulation.

## 1. INTRODUCTION

Reliable design of digital VLSI systems requires extensive logic simulations consuming enormous amounts of CPU time. Parallel processing offers a viable way to improve upon this time. Two main classes of algorithms exist for parallel logic simulation known as the synchronous and asynchronous algorithms. In synchronous simulation (sometimes referred to as centralized-time simulation), a centralized clock for the simulation time is maintained. All logic elements experiencing input events at the current simulation time are processed and then the clock is advanced by one time unit to the next simulation time. In contrast, the asynchronous simulation (also called distributed simulation) does not require any centralized clock to coordinate its execution. Instead, all events carry the simulation time information (timestamp) themselves. In conservative asynchronous simulation, a logic element is ready for evaluation as soon as all of its inputs have received a token (a logical value and its timestamp). When a logic element evaluates, it produces an output based on the logical value of the input tokens and consumes the input token(s) with the lowest timestamp. The output has a timestamp equal to the timestamp of the consumed input token(s) plus the delay of the logic element itself. In the "conservative" form of asynchronous simulation, the time order of tokens is always

guaranteed and only "safe" evaluations are allowed i.e., an evaluation guaranteeing a correct result.

In implementing the event driven principle (i.e., sending an output token to the fanout elements only if there is a change in its logical value), the conservative asynchronous simulation can deadlock. A deadlock is a situation where no element can evaluate because at least one of its inputs is missing a token. This occurs frequently in the simulation of circuits with feedback because if the output that is feeding back did not change, no token will be sent to that input, causing a deadlock.

There are two ways to handle deadlocks (proposed by Chandy and Misra [2–3]); one is deadlock avoidance by the use of NULL or redundant messages, the other is deadlock detection and recovery. Bailey [1] develops the execution time of asynchronous simulation without considering the overhead due to handling of deadlocks. We do consider this overhead in the execution times of ISCAS-85 [4] and ISCAS-89 [5] benchmark circuits.

In the development of execution times of synchronous and conservative asynchronous simulation, Bailey first describes the circuit to be simulated in terms of a simulation dependency graph, $\mathcal{D}$, which is a directed graph of events with each vertex representing an event in the circuit. The vertices in the graph are labeled with events and the edges are labeled with delays in the circuit. Both the events and the delays have positive integer values. If a parent event causes a child event, then there is an edge in $\mathcal{D}$ from the parent event vertex to the child event vertex with a delay of the logic element corresponding to the child event. The execution times of synchronous and conservative asynchronous simulation are developed in terms of this graph. In Bailey's analyses, a fixed execution sequence is assumed, the evaluation time of each vertex in the graph is equal, an unlimited number of processors are available and the inputs to a circuit remain fixed during a simulation. Under the above assumptions, it is then proved that the unit-delay simulation is a lower bound on the execution times of both synchronous and conservative asynchronous simulations and that these execution times are equal.

We continue a similar development here but relax the assumption that the inputs to a circuit are to remain fixed during a simulation. Since most practical simulations require testing the circuit with a large set of different inputs, it is more meaningful to analyze the parallelism and execution time of synchronous and asynchronous simulations under varying input conditions. As will be shown later in this paper, the presence of multiple input events allows the conservative asynchronous simulation to extract more parallelism due to its capability to process events belonging to different simulation times and thus quite different conclusions are obtained as compared to [1]. In analyzing parallelism in the execution of a simulation, we examine both pipelining and concurrency in the processing of events. Pipelining corresponds to processing a stream of events on a line in a circuit. It is a measure of how quickly the next consecutive event on an input line of a logic element can be processed after the previous event has been consumed. Concurrency refers to parallel evaluation of different logic elements at a given time, in response to events on their inputs.

In the remaining organization of this paper, we analyze the synchronous and asynchronous simulations individually and develop bounds on the execution times for general multi-input, multi-output circuits experiencing an arbitrary number of input events. A relative comparison of the synchronous and conservative asynchronous simulation execution times is then presented to show that the conservative asynchronous simulation may execute faster. It is also shown that the conservative asynchronous simulation can further improve upon its execution time by employing safe lookaheads i.e., an evaluation based on the controlling input being present on a logic element's input. Finally, a comparison of the idealized execution times of the synchronous and conservative asynchronous simulation algorithms on ISCAS combinational and sequential benchmark circuits is presented to support our conclusions. It is shown that the conservative asynchronous simulation implementing the deadlock avoidance scheme exploits better pipelining and concurrency in element evaluations and even with the overhead of NULL messages, executes

faster than both the synchronous simulation and the conservative asynchronous simulation implementing the deadlock detection and recovery scheme. Except for allowing inputs to change during a simulation, the remaining assumptions throughout this paper are similar to Bailey [1] i.e., all logic elements have a unit-delay, an unlimited number of processors are available, and each processor evaluates a logic element in $E$ time units.

An initial version of this paper was presented at the *6th IEEE Symposium on Parallel and Distributed Processing*, October 1994.

## 2. EXECUTION TIME OF SYNCHRONOUS SIMULATION

Bailey shows the execution time of unit-delay synchronous simulation, $\tau_{s,u}$, to be

$$\tau_{s,u} = E^* (\text{depth}(\mathcal{D}) + 1) \quad (1)$$

The limitation of (1) is that it is only valid for a single input circuit with a single input event or for multiple input circuits such that an event occurs at exactly the same simulation time on different inputs. To allow for multiple input events, equation (1) needs to be modified to take into account the pipelining effect taking place due to a sequence of events on an input, and the possible concurrency due to events on different inputs. As an example of pipelining, if two events separated by one time unit are received on the input of a single input circuit, then the execution time is, $\tau_{s,u} = E^* ((\text{depth}(\mathcal{D}) + 1) + 1)$ i.e., it increases by only one evaluation time. This is because while a logic element at depth $i$ in $\mathcal{D}$ is executing the first event, the element at depth $i$-1 is executing the second event. On the other hand, if the two consecutive input events on a line are separated in simulation time by $t_{sep}$ time units and $t_{sep} \geq \text{depth}(\mathcal{D})$, then $\tau_{s,u} = E^* (\text{depth}(\mathcal{D}) + 1)$ *2 i.e., the execution time doubles to that of a single event due to lack of pipelining. In general for a single input circuit with a sequence of $e$ input events, the execution time of unit-delay synchronous simulation is bounded by

$$E^* ((\text{depth}(\mathcal{D}) + 1) + e - 1) \leq \tau_{s,u} \leq E^* ((\text{depth}(\mathcal{D}) + 1)^* e.$$

For a completely general circuit, we must allow for an arbitrary number of external inputs, with each input experiencing different number of events at different simulation times. The calculation of execution time then requires that the simulation dependency graph be identified for each external input. We denote $\mathcal{D}_i$ as the section of $\mathcal{D}$ corresponding to an input $i$. Let $n$ be the number of external inputs and $e_i$ be the number of input events on an input $i$. Then the best-case execution time for the unit-delay synchronous simulation is given by (2). It occurs when all input events on a line are separated by one time unit to extract maximum pipelining, and different inputs receive events at the same simulation time to achieve maximum concurrency.

$$\tau_{s,u} = \underset{i = 0 \text{ to } n-1}{Max} (E * ((\text{depth}(\mathcal{D}_i) + 1) + e_i - 1)) \quad (2)$$

The worst-case execution time is given by (3) and occurs when all input events are separated in simulation time by an interval greater than or equal to the depth of the simulation dependency graph, such that there is no pipelining or concurrency (between different external input events).

$$\tau_{s,u} = \sum_{i=0}^{n-1} E *(\text{depth}(\mathcal{D}_i) + 1)^* e_i \quad (3)$$

We illustrate the best- and worst-case execution times using an example. An exclusive-OR circuit is shown as an interconnection graph in Figure 1. Figure 2 shows the simulation dependency graph $\mathcal{D}_i$ for each input. The vertices in the graph are labeled with events and the edges are labeled with the delays in the circuit.

If the inputs A and B in Figure 2 experience two events at times 5 and 6, then from (2) the best-case execution time is, $\tau_{s,u} = E * ((3 + 1) + 2 - 1) = E$

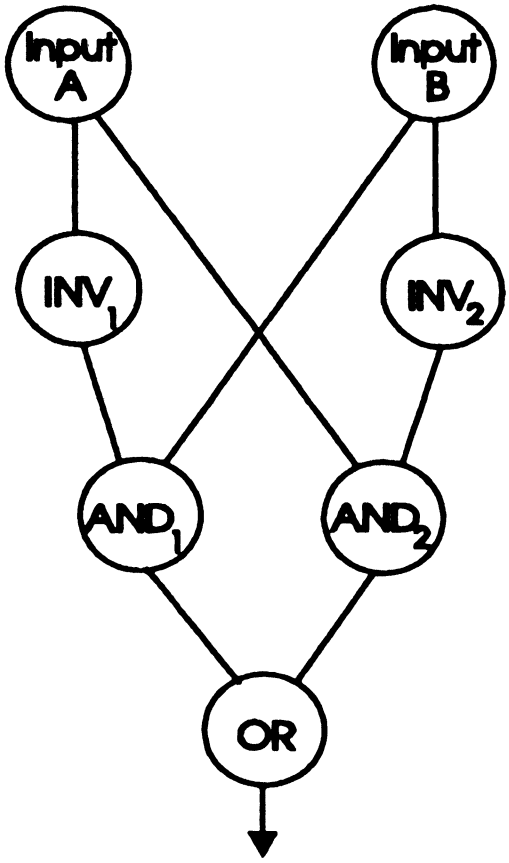


FIGURE 1 Circuit Interconnection Graph for an Exclusive-OR Circuit

* 5. The pipelining and concurrency in execution are shown in Figure 3 by mapping $\mathcal{D}$ to the execution time (assuming $E = 1$). To show the worst-case execution time, if input A experiences two events at times 5 and 15, and input B experiences events at times 10 and 20, then from (3), $\tau_{s,u} = E * (3 + 1) * 2 + E * (3 + 1) * 2 = E * 16$. Figure 4 shows this by mapping $\mathcal{D}$ to the execution time for $E = 1$. Note that there is no pipelining or concurrency between different input events in this case because of the wide separation of events in simulation time in relation to the circuit depth.

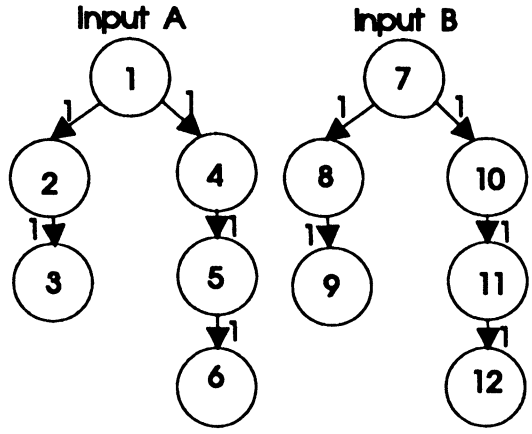


FIGURE 2 Simulation Dependency Graph for the Exclusive-OR Circuit

## 3. EXECUTION TIME OF CONSERVATIVE ASYNCHRONOUS SIMULATION

For unit-delay conservative asynchronous simulation, assuming an unlimited number of processors, Bailey shows the execution time, ($\tau_{c,u}$) to be,

$$\tau_{c,u} = E * (\text{depth}(\mathcal{D}) + 1) \tag{4}$$

It can be easily verified that (4) is not valid for any simulation other than a single input circuit receiving a single event. Taking into account the effect of multiple input events, we develop the expressions for the best- and worst-case execution times of conservative asynchronous simulation for general multi-input, multi-output circuits.

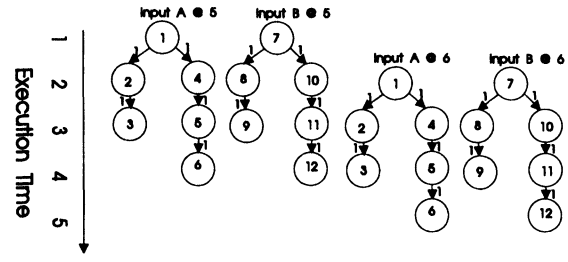


FIGURE 3 Execution Time for the Exclusive-OR Circuit Showing Pipelining and Concurrency in Synchronous Simulation

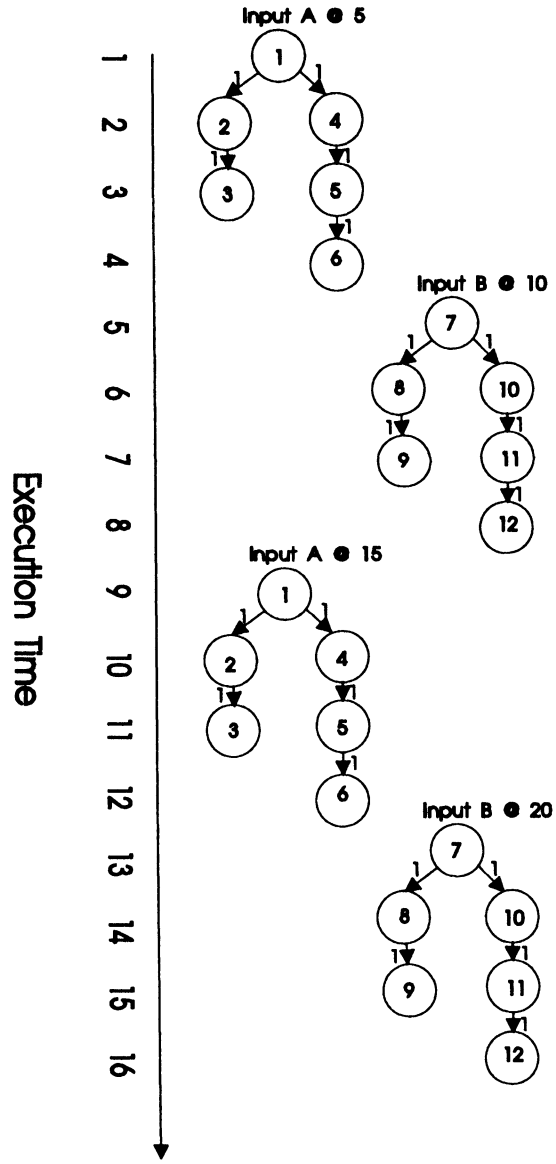


FIGURE 4 Execution Time for the Exclusive-OR Circuit Showing Lack of Pipelining and Concurrency in Synchronous Simulation

It should be noted that while the synchronous simulation processes all events at each simulation time completely before proceeding on to the next time, the asynchronous simulation can concurrently process events belonging to different simulation times as they are produced. This allows asynchronous simulation to exploit pipelining, due to a sequence of events on an input of a logic element, as well as concurrency, due to different logic elements receiving events at the same execution time (not necessarily at the same simulation time). As an example, if two different logic elements are ready for evaluation because all their inputs have received tokens and the tokens for the two logic elements contained different timestamps, the asynchronous model allows them to execute concurrently whereas the synchronous simulation will allow only one execution at a time.

For a general circuit with $n$ inputs and $e_i$ events on an input $i$, the best-case execution time of conservative asynchronous simulation is given by (5).

$$\tau_{c,u} = \underset{i = 0 \text{ to } n-1}{Max} (E * ((\text{depth}(\mathcal{D}_i) + 1) + e_i - 1)) \tag{5}$$

It occurs when there is maximum pipelining and concurrency available in simulation. Note that unlike synchronous simulation, the separation in terms of simulation time is not a factor for exploiting either pipelining or concurrency in asynchronous simulation.

The worst-case execution time for asynchronous simulation is caused by reduced parallelism due to the way it processes events. In asynchronous simulation, each logic element has to sequence the input events in terms of their timestamps to guarantee correct behavior. During evaluation, a logic element consumes the input token with the lowest timestamp and produces an output with a timestamp equal to the timestamp of the consumed token plus the delay of the element itself. Thus even if the events appearing on different inputs of a logic element were generated in parallel, a number of output events equal to the sum of all input events have to be generated sequentially in the worst case, thereby reducing the concurrency in simulation. An example of this is shown in Figure 5, where the two inverters process the events concurrently belonging to different simulation times but when passing through the AND gate, the generation of events is serialized on its output because of

**Format:** Event @ Simulation time$_{\text{execution time}}$

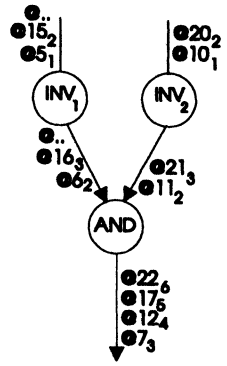


FIGURE 5 An Example Showing Serialization of Generation of Output Events

the differences in the input timestamps. In Figure 5, the execution time for the generation of each event is denoted as a subscript to the event and it is assumed that $E = 1$. The execution time for the output of a logic element equals one more than the maximum execution time on the front of its inputs. This is because, in conservative asynchronous simulation, a logic element is not ready for evaluation until it has received all of its inputs. In Figure 5, @.. indicates additional events on an input, thus allowing the consumption of all events in the example.

The example shown in Figure 5 demonstrates that multi-input logic elements may reduce the concurrency in asynchronous simulation by serializing the generation of events if they receive events that are separated in simulation time on their different inputs. Taking this effect into account, the worst-case execution time for conservative asynchronous simulation is given by (6).

$$\tau_{c,u} = \underset{\text{each input to output path}}{Max} [E^*[\sum_{k=1}^{path\_length} \{(\sum_{i=0}^{fanin_k-1} e_{k,i}) - (\sum_{i=0}^{fanin_{k-1}-1} e_{k-1,i}) + 1\}]] \quad (6)$$

where $e_{k,i}$ denotes the number of events at the input $i$ of a logic element at level $k$ in a given input-to-output path. Before applying (6), the number of events at each output of a logic element is computed by accumulating the number of events on the fanin lines of that element. Equation (6) is derived from the circuit interconnection graph. Starting from an input-to-output path, it accumulates the events at the inputs of a logic element at each level $k$ as indicated by the term $\sum_{i=0}^{fanin_k-1} e_{k,i}$ (inputs are labeled starting from 0 to fanin-1). The pipelining effect is taken into account by subtracting the number of events at the previous $k - 1$ level by the term $\sum_{i=0}^{fanin_{k-1}-1} e_{k-1,i}$ (this term is zero for $k = 1$). This accumulation of events is carried out for each input-to-output path and the execution time of the simulation is the maximum over all these paths. By applying (6) to the circuit of Figure 5, it can be seen that $\tau_{c,u} = [(2) - (0) + 1]_{k=1 \text{ -for INVERTER}} + [(4) - (2) + 1]_{k=2 \text{ -for AND}} = E * 6$. To explain this further, both inverters in Figure 5 have two input events which can be executed in parallel, so for k = 1 level, the execution time is 2 (for this level) − 0 (for previous level) + 1 = $E$*3. For k = 2 level, the AND gate has 2 input events on each of its inputs, so the worst case execution time would be 4 (for this level) − 2 (for previous level, to account for the pipelining effect for k = 1) + 1 = $E$*3. Combining the execution time for all levels, we obtain a total time of $E * 6$.

## 4. COMPARISON OF SYNCHRONOUS AND CONSERVATIVE ASYNCHRONOUS SIMULATION

The best- and worst-case execution times for synchronous and conservative asynchronous simulation are given by Equations (2–3) and (5–6) respectively. In comparing the best cases, it can be seen that Equation (2) for synchronous simulation is exactly identical to Equation (5) for conservative asynchronous simulation. However, there are differences in the requirements for achieving this minimum time. The best case for synchronous simulation occurs when the events on an input are separated in simulation time by

only one time unit to exploit maximum pipelining, and events on different inputs occur at the same simulation time to get maximum concurrency. The conservative asynchronous simulation does not have this requirement and is capable of exploiting both pipelining and concurrency for widely separated events. For instance, the asynchronous simulation of a chain of inverters executes in the minimum time given by Equation (5) regardless of the separation time of input events. In contrast, the synchronous simulation requires input events to be separated by only one time unit to achieve the best execution time. Note also that in most practical simulations, the input data to a circuit is held stable for at least the delay through the circuit. Thus the asynchronous simulation may achieve the minimum time but the synchronous simulation cannot as the input events are almost always separated by more than one time unit in practical simulations.

In order to achieve the lowest possible execution time when there are multi-input logic elements involved, the conservative asynchronous simulation does require that the events on different inputs of a logic element have the same timestamps. This condition allows for consumption of multiple input events thus minimizing the effect of serialization in the generation of output events. Hence this condition ultimately requires a fixed simulation time difference in the external input events (depending upon the delay of the path of each input of a logic element to the external input) to achieve the best execution time. This is rather a stringent requirement as can be seen from an example. If the first input of an AND gate receives events through a chain of two inverters connected to an external input and the other input is an external input, then the external input events have to be separated by 2 simulation time units to result in minimum execution time in asynchronous simulation.

The minimum time given by Equation (5) would not be obtainable for most circuits because of the conflicting timing requirements from multiple paths through the circuit. Figure 6 illustrates this point using the data from Figure 3. The minimum execution time given by Equation (5) is not achieved by asynchronous simulation because events at the inputs to

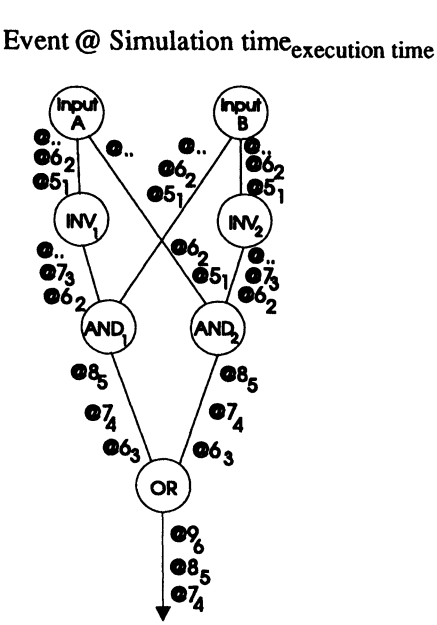


FIGURE 6 Asynchronous Simulation Execution for the Exclusive-OR Circuit with Input Events Separated by One Time Unit

the AND gates are not optimally separated in time leading to some serialization. The execution of this example using the asynchronous simulation takes 6 time units in comparison to 5 time units required for the synchronous approach which accomplishes the task in minimum time.

In short, the requirements on both synchronous and asynchronous simulation to achieve the best execution time as given by Equations (2) and (5) are quite strict. The best execution time may not be observed for either type of simulation. The requirements for Equation (2) to be used would never be achieved in practical circuits that often use an input data that is held constant for at least the delay through the entire circuit. Likewise the requirements for the use of Equation (5) would not be achievable by most circuits having recombination of paths with different delays, although this is mitigated by not having an output event for each input event as has been assumed in the development above.

The worst-case execution times for synchronous and asynchronous simulation are given by Equations (3) and (6) respectively. The synchronous simulation

exhibits the worst-case execution time when all input events are separated by time intervals greater than or equal to the depth of the simulation dependency graph. In this case it is unable to exploit any pipelining or concurrency. However, asynchronous simulation can extract some pipelining and concurrency independent of the time separation of input events. This pipelining and concurrency are reduced when passing through multiple input gates due to the serialization in generation of output events. For example, the asynchronous simulation shown in Figure 5 takes 6 time units (worst-case time for asynchronous simulation) to complete and has concurrent evaluations in the two inverters. The synchronous simulation can not execute the events at the inputs of the two inverters concurrently since they belong to different simulation times and thus it takes 8 time units (worst-case time for synchronous simulation) to complete. Another comparison is made in Figure 7, where the asynchronous simulation for the exclusive-OR circuit takes 11 time units to complete as opposed to 16 time units for synchronous simulation (which is shown in Figure 4). The execution time for the conservative asynchronous simulation can also be verified by applying Equation (6) to the exclusive-OR circuit in Figure 7 for the two input events on each input, yielding [(2) − (0) + 1] + [(4) − (2) + 1] + [(8) − (4) + 1] = 11 time units to execute. Thus other than some very special circuits e.g., a completely serial circuit or a circuit with only one gate, the worst-case execution time of conservative asynchronous simulation will be less than that of synchronous simulation.

As shown by the above analyses and examples, the theorem 4 in Bailey's paper [1] that the execution times of unit-delay, synchronous and asynchronous simulations are equal, is not valid for simulations experiencing multiple input events. Generally, the conservative asynchronous simulation can exploit better pipelining and concurrency as compared to the synchronous simulation for widely varying events in terms of their timestamps and thus results in less execution time. The execution time of conservative asynchronous simulation can be further improved by incorporating safe lookahead as described in the next subsection.

**Format:** Event @ Simulation time$_{\text{execution time}}$

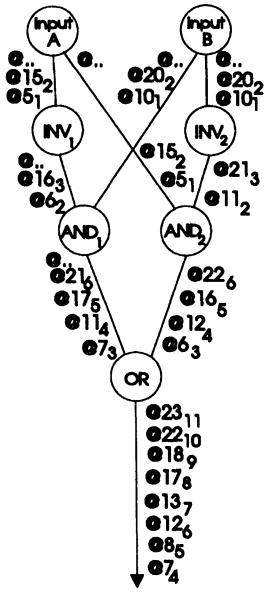


FIGURE 7 Asynchronous Simulation Execution for the Exclusive-OR Circuit with Widely Separated Input Events

### 4.1 Improving Asynchronous Simulation by Incorporating Lookahead

Asynchronous simulation can exploit lookahead to further improve upon its execution time. Lookahead corresponds to a prediction of the output when not all input tokens of an element have been received. In the conservative asynchronous simulation, lookahead should always produce a correct prediction of the output. This can be achieved by performing an evaluation based on a controlling input (e.g., 0 is controlling value for an AND gate, and 1 for an OR gate). The presence of an input token with a controlling value is sufficient to determine the output. Hence, by incorporating lookahead in the conservative algorithm, it is not necessary to wait until a token is present on all its inputs before an element can be evaluated. If any of the tokens in the front of an input

queue of a logic element has a controlling value, the element is evaluated. The output token produced has a timestamp equal to the highest timestamp of the controlling input token plus the delay of the element.

In order to implement lookahead, each element maintains a "lookahead counter" and a location to store its controlling value. The lookahead counter stores the highest timestamp of the tokens in the front of the input queues that have a controlling value. Any incoming input token having a timestamp less than this lookahead count is absorbed. Thus many input tokens can be absorbed in one evaluation and an output produced with a much higher timestamp than would be possible without using lookahead. This minimizes the number of messages and improves the execution time of the conservative asynchronous simulation. The pseudocode for the lookahead based conservative asynchronous algorithm is shown in Appendix B.

The lookahead scheme can be implemented on multi-input AND, NAND, OR and NOR gates. Inverters and exclusive-OR gates do not have any controlling values as such and thus cannot take advantage of lookahead. Lookahead can also be applied to edge-triggered flip flops. Since, after the triggering edge has been detected, the output can be correctly predicted up to the next triggering edge.

Conservative asynchronous simulation on a combinational circuit comprised of multi-input AND, OR type gates can generally improve 50% upon its execution time by employing lookahead. This can be seen by assuming the probability that the output of a gate is 0 to be 0.5 i.e., the output is 0 half the time and 1, the rest of the time. The number of gate evaluations using lookahead will thus be reduced by half because half the time at least one of the inputs will have a controlling value. For sequential circuits, the conservative asynchronous simulation based on the deadlock avoidance scheme can have a much higher performance gain by using lookahead. This is because in addition to the reduced gate evaluations, lookahead greatly minimizes the number of NULL messages needed to avoid the deadlocks in feedback loops. Some results on benchmark circuits are presented in the next section that demonstrate the effectiveness of lookahead.

## 5. EVALUATION ON BENCHMARK CIRCUITS

We measured the execution times of combinational ISCAS-85 [4] and sequential ISCAS-89 [5] benchmark circuits on both synchronous and conservative asynchronous simulation algorithms. All circuits were simulated under unit-delay, as unit-delay has been shown to be the lower bound on the execution time of either synchronous or conservative asynchronous algorithm [1]. In the implementation of synchronous algorithm, a timing wheel is used whose time slots contain events that can be executed in parallel. Thus for a given data set (with unlimited number of processors and one time unit for evaluation of an element), the execution time of synchronous simulation is equal to the number of non-empty time slots [1].

For conservative asynchronous simulation, we first implemented the algorithm presented in [6,9] which uses an avoidance scheme to handle deadlocks. This algorithm was then further improved upon by incorporating lookahead. Our lookahead implementation used lookahead on multiple input gates as well as flip flops. The pseudocodes for the conservative asynchronous algorithm and the improved form incorporating lookahead are given in appendices A and B respectively. In this algorithm, NULL messages are generated only if there is a possibility of a deadlock. This is detected when one of the inputs of a logic element becomes empty as a result of an evaluation. In this case, the output is sent to its fanout elements regardless of a change from its previous value. Note that this is an optimization over Chandy and Misra's always send NULL message strategy in [2–3].

In our implementation, we have an input queue of size 16 for all inputs to a logic element. For an asynchronous algorithm using the avoidance scheme, the simulation execution time generally improves as the

input queue size is increased and usually saturates for a queue size of about 5. In our execution time measurements, an unlimited number of processors is assumed with one unit evaluation time for a logic element and zero communication time for distributing tokens to the fanout of a gate. This is consistent with [1] and chosen so that the parallelism in an algorithm can be determined independent of the communication overhead. However, as communication time increases, the synchronous and asynchronous algorithms would perform relatively the same. The total time units to complete the asynchronous simulation were measured using the same data set as used for the synchronous simulation. Table I shows the characteristics of the benchmark circuits and the data set. Data for the ISCAS-85 combinational circuits (c prefix) consisted of 30 random sets. The length of a set for a particular circuit was adjusted so that the circuit would reach a stable state before the next data was entered i.e., the length of a data set corresponds to the maximum depth of the circuit. Data for the ISCAS-89 sequential circuits consisted of 40 random sets. Data was preceded by several clock cycles to reset the flip flops in the circuit. Data was changed only during the middle of the positive clock pulse, and remained constant for a single clock period. Clock cycle times were adjusted for different circuits so that the circuit would reach a stable state before the next clock cycle.

The results of the execution times of the two algorithms on combinational and sequential benchmark circuits are shown in Table II. It can be seen that the execution time of asynchronous simulation with lookahead is much lower than the synchronous simulation for all circuits. On the average, the conservative asynchronous simulation is almost three and a half times faster than synchronous simulation for combinational circuits, and two times faster for sequential circuits. The redundant or NULL messages used in the asynchronous algorithm cause the overall execution time of conservative asynchronous simulation to increase because extra evaluations may take place at the element receiving these messages. The sequential circuit simulations generate a large number of NULL messages to avoid a large number of deadlocks (see Table IV). The execution time data in Table II includes this effect and despite the overhead of NULL messages, the asynchronous simulation still outperforms synchronous simulation for combinational as well as sequential circuits when lookahead is employed.

We carried out a similar comparison between the synchronous algorithm and an asynchronous algorithm based on deadlock detection and recovery scheme. In the deadlock detection and recovery scheme, the circuit is allowed to deadlock which is a condition in which no logic element can evaluate because at least one of its inputs is missing a token. After a deadlock has been detected, the circuit recovers by computing a global minimum time "gmt" (which is the smallest time of an unconsumed event in the circuit) and updating token timestamps which

TABLE I Characteristics of the Benchmark Circuits and the Data Used for Evaluating Simulation Algorithms

| Circuit | Number of primary inputs | Number of primary outputs | Number of gates | Number of Flip Flops | Total Number of test vectors | Number of sets in data |
|---|---|---|---|---|---|---|
| c432 | 36 | 7 | 160 | | 1200 | 30 |
| c1355 | 41 | 32 | 546 | | 1200 | 30 |
| c1908 | 33 | 25 | 880 | | 1200 | 30 |
| c2670 | 233 | 140 | 1193 | | 1200 | 30 |
| c3540 | 50 | 22 | 1669 | | 1500 | 30 |
| c7552 | 207 | 108 | 3512 | | 1650 | 30 |
| s208.1 | 10 | 1 | 104 | 8 | 2171 | 40 |
| s382 | 3 | 6 | 158 | 21 | 2565 | 40 |
| s641 | 35 | 24 | 379 | 19 | 2505 | 40 |
| s1423 | 17 | 5 | 657 | 74 | 2505 | 40 |
| s5378 | 35 | 49 | 2779 | 179 | 2515 | 40 |
| s9234.1 | 36 | 39 | 5597 | 211 | 2521 | 40 |

TABLE II Execution Times of Benchmark Circuits on the Synchronous Algorithm and an Asynchronous Algorithm using the Avoidance Scheme

| Circuit | Synchronous (time units) | Conservative Asynchronous-avoidance scheme w/o lookahead (time units) | Conservative Asynchronous-avoidance scheme with lookahead (time units) | Synchronous Asynchronous-w/lookahead (avoidance scheme) |
|---|---|---|---|---|
| c432 | 417 | 198 | 110 | 3.8 |
| c1355 | 637 | 539 | 152 | 4.2 |
| c1908 | 772 | 538 | 200 | 3.9 |
| c2670 | 650 | 297 | 178 | 3.7 |
| c3540 | 989 | 507 | 296 | 3.3 |
| c7552 | 1160 | 617 | 445 | 2.6 |
| s208.1 | 548 | 3555 | 277 | 2.0 |
| s382 | 791 | 5206 | 361 | 2.2 |
| s641 | 1417 | 4928 | 811 | 1.8 |
| s1423 | 1144 | 6866 | 513 | 2.2 |
| s5378 | 1444 | 7436 | 594 | 2.4 |
| s9234.1 | 1666 | 7943 | 725 | 2.3 |

are less than gmt to gmt [7]. Table III shows a comparison on benchmark circuits between the synchronous algorithm and the asynchronous algorithm based on deadlock detection and recovery scheme (DDR). In Table III, it is assumed that the circuit recovers from a deadlock in 0 time. Even with this unrealistic assumption, the conservative asynchronous simulation based on the deadlock detection and recovery scheme performs worse than the synchronous simulation. This is because the *deadlock detection and recovery scheme looses much of the pipelining when the circuit deadlocks* causing its performance to be worse than the synchronous simulation.

It can be seen from Table III that when the number of deadlocks is relatively small (e.g., c2670 circuit), the asynchronous simulation approaches the synchronous simulation execution times, and its performance is relatively worse when the number of deadlocks is high. The results in Table III agree with other researcher's conclusions about the relatively poor per-

TABLE III Execution Times of Benchmark Circuits on the Synchronous Algorithm and an Asynchronous Algorithm Using the Deadlock Detection and Recovery Scheme

| | Synchronous (time units) | Conservative Asynchronous DDR scheme w/o lookahead (time units) | Number of Deadlocks in DDR scheme w/o lookahead | Conservative Asynchronous DDR scheme with lookahead (time units) | Number of Deadlocks in DDR scheme with lookahead | Synchronous Asynchronous-w/lookahead (DDR scheme) |
|---|---|---|---|---|---|---|
| c432 | 417 | 468 | 78 | 470 | 78 | 0.9 |
| c1355 | 637 | 2200 | 319 | 2198 | 319 | 0.3 |
| c1908 | 772 | 3437 | 437 | 3214 | 413 | 0.2 |
| c2670 | 650 | 987 | 47 | 645 | 21 | 1.0 |
| c3540 | 989 | 3978 | 520 | 2182 | 190 | 0.5 |
| c7552 | 1160 | 5090 | 374 | 4238 | 325 | 0.3 |
| s208.1 | 548 | 1015 | 412 | 1004 | 364 | 0.6 |
| s382 | 791 | 1336 | 438 | 1488 | 398 | 0.6 |
| s641 | 1417 | 2903 | 607 | 2930 | 594 | 0.5 |
| s1423 | 1144 | 2486 | 790 | 2585 | 767 | 0.4 |
| s5378 | 1444 | 5115 | 990 | 5474 | 929 | 0.3 |
| s9234.1 | 1666 | 8204 | 1369 | 8024 | 1344 | 0.2 |

formance of the conservative asynchronous deadlock detection and recovery scheme as compared to the synchronous scheme. Soule' [7] has done a similar comparison on a variety of circuits under the same assumptions as ours (i.e., unlimited number of processors and zero time to recover from a deadlock) and found the asynchronous simulation using the deadlock detection and recovery scheme to perform worse than the synchronous simulation. Soule' also examined the conservative asynchronous avoidance scheme and found it to be extremely poor relative to the synchronous simulation, but his implementation did not carry the NULL message optimization as we have described, instead in his implementation a NULL message was sent out after every evaluation (Chandy and Misra's always send NULL message scheme).

Table IV compares the NULL message overhead in different conservative asynchronous schemes based on deadlock avoidance and it can be seen that the conservative asynchronous scheme with lookahead has the least overhead in terms of NULL messages as compared to actual events in the circuit. Even though for sequential circuits, the number of NULL messages is two to three times more than the number of events in the lookahead based avoidance scheme, the execution time is still better than the synchronous simulation because of the increased pipelining and concurrency in event processing.

All ISCAS benchmark circuits were tested in this work. However, for keeping the paper to a reasonable length, we report the results on only a few of these circuits. More results on other circuits can be found in [8]. The results on remaining circuits are relatively similar to the ones we have presented in this paper. Further, in an implementation on a data flow architecture based hardware accelerator with limited number of processors [9], the performance of the synchronous and the optimized conservative asynchronous algorithms shows relatively similar results as we report in this paper.

Overall, the ability of the conservative asynchronous algorithm to concurrently evaluate logic elements with each element's inputs having differing timestamps from other element's inputs and its ability to exploit better pipelining along with lookahead allow it to execute faster than the synchronous simulation. The conservative asynchronous algorithm implementing the deadlock avoidance scheme maintains better pipelining of events on the input(s) of a logic element and thus executes faster than the deadlock detection and recovery scheme in which the pipelining effect is lost when the circuit deadlocks.

## 6. CONCLUSIONS

In this paper, we have extended Bailey's analysis of synchronous and conservative asynchronous logic simulation by considering multiple input events. By taking into account both event pipelining and concurrency due to multiple input events, the expressions for the best- and worst-case execution times of synchronous and conservative asynchronous simulations for general multi-input, multi-output circuits were developed. It is then shown that the conservative asynchronous simulation has the ability to exploit better pipelining and concurrency due to widely varying times in input events and thus can execute faster than the synchronous simulation in general.

Our conclusions are supported by the simulation execution times of combinational ISCAS-85 and sequential ISCAS-89 benchmark circuits on the synchronous and conservative asynchronous algorithms. Even with the overhead of NULL messages, the conservative asynchronous simulation using the optimized deadlock avoidance scheme exploits better pipelining and concurrency, and thus executes faster than both the synchronous simulation and the conservative asynchronous simulation based on the deadlock detection and recovery scheme which looses all its pipelining when deadlocks occur.

Thus our work presents important conclusions different than previously proved in [1], and shows the effectiveness of conservative asynchronous simulation in terms of parallelism and execution time over synchronous simulation when a lookahead scheme is employed. Although the overhead associated with asynchronous simulation (maintaining input queues in each logic element etc.) is higher than synchronous simulation which makes it unattractive for software

TABLE IV Comparison of NULL Message Overhead in different Asynchronous Conservative Schemes based on Deadlock Avoidance

| Circuit | Unoptimized Conservative Asynchronous-Chandy and Misra's always send NULL scheme Null_count / event_count | Conservative Asynchronous -avoidance scheme w/o lookahead Null_count / event_count | Conservative Asynchronous -avoidance scheme with lookahead Null_count / event count |
|---|---|---|---|
| c432 | 3.97 | 0.62 | 0.28 |
| c1355 | 5.10 | 0.77 | 0.51 |
| c1908 | 4.42 | 0.54 | 0.27 |
| c2670 | 2.54 | 0.50 | 0.26 |
| c3540 | 3.74 | 0.53 | 0.36 |
| c7552 | 2.36 | 0.42 | 0.28 |
| s208.1 | 131.28 | 11.61 | 2.62 |
| s382 | 147.53 | 17.53 | 2.63 |
| s641 | 114.69 | 27.81 | 1.74 |
| s1423 | 114.15 | 49.45 | 2.02 |
| s5378 | 107.32 | 41.53 | 1.61 |
| s9234.1 | 189.08 | 60.32 | 3.7 |

implementations, our work shows that it has high potential for hardware acceleration of logic simulation.

### *Acknowledgements*

This research was supported in part by a grant from the National Science Foundation Center for Design of Analog-Digital Integrated Circuits under grant #CDADIC 92-4.

### *References*

[1] Bailey, M. L., "A Time Based Model for Investigating Parallel Logic-Level Simulation," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 816–824, July 1992.

[2] Chandy, K. M. and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, September 1979, pp. 440–452.

[3] Chandy, K. M., and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 11, pp. 198–206, April, 1981.

[4] Brglez, F., P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," Proceedings of the *IEEE International Symposium on Circuits and Systems*, 1985, pp. 695–698.

[5] Brglez, F., D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," Proceedings of the *IEEE International Symposium on Circuits and Systems*, 1989, pp. 1929–1934.

[6] Mahmood, A., W. I. Baker, J. Herath and A. Jayasumana, "A Logic Simulation Engine Based on a Modified Data Flow Architecture," Proceedings of the *IEEE International Conference on Computer Aided Design, ICCAD-92*, pp. 377–380.

[7] Soule', L. P., "Parallel-Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms," *Technical Report: CSL-TR-92-527, Computer Systems Laboratory*, Stanford University, 1992.

[8] Baker, William I., "An Evaluation of Parallel Logic Simulation Algorithms," *Ph.D. Dissertation*, Washington State University, April, 1994.

[9] Mahmood, Ausif, and William I. Baker, "Parallel Event Driven Logic Simulation on an Application-Specific Data Flow Architecture," *Journal of High Performance Computing*, vol. 1, no. 2, 1994.

## APPENDIX A

### Conservative Asynchronous Simulation Algorithm

Each logic element contains finite size input and output queues. To avoid the overflow of input queues, a demand driven scheme is implemented for sending output tokens to fanout elements. An element can transmit an output token only if a demand exists from each of the fanout elements. Every element maintains a demand

count for each of the fanout elements which is initially set to the maximum input queue size. The demand count for each fanout element is decremented whenever an output is sent out. A signal to increment the demand count for this element is sent to the fanin element whenever a token is consumed in the input buffer.

A transmit_flag is associated with each element which indicates if the present output has been transmitted to the fanout elements.

Evaluate-Transmit PROCESS executed in each logic element:

***—evaluate phase***

```
If ((all input queues have at least one token) && (new output queue is not full))
 {  Compute the logical output based on the inputs in the front of the input queues.
    Add new output token to new output queue
    /* The computed output known as the new output has a time stamp = time stamp of absorbed input(s)
       + delay of the element*/
    Compute the lowest time stamp of the inputs in the front of the input queues.
    Absorb the input(s) with the lowest time stamp.
    Send demand signal(s) to the input(s) absorbed.}
```

***—transmit phase***

```
If ((new output queue is not empty) && (transmit_flag = FALSE)
     && (logical value of token in front of new output queue ≠ present output))
 {  If (demand > 0 for all fanout elements)
    {  Transmit present token to fanout elements.
       Decrement the demand count for each fanout element.
       Present token = new token and remove token from front of new output queue.}
    else WAIT on (demand > 0 for all fanout elements)
 }

If ((new output = present output || transmit_flag = TRUE) && new output queue not empty)
 {  Present token = new token and remove token from front of new output queue. transmit_flag = FALSE.}

If ((at least one input buffer is empty) && (new output queue is empty))
 {  If (demand > 0 for all fanout elements)
     {Transmit present token to fanout elements.
      Decrement the demand count for each fanout element.
      transmit_flag = TRUE.}
    else WAIT on (demand > 0 for all fanout elements || inputs becoming not empty)}
```

## APPENDIX B

### Conservative Asynchronous Simulation Algorithm with Gate Level Lookahead

***—evaluate phase***

```
done_lookahead = 0;
while (! done_lookahead)
  { done_lookahead = 1; /* if another lookahead cycle is possible, it is reset to 0 */
    case_inputs_present = 0; case_lookahead = 0;
    if (each input has at least one token) case_inputs_present = 1;
    if (a front input has controlling value with time stamp > element's lookahead_cnt)
```

```
{  case_lookahead = 1; element's lookahead_cnt = time stamp}

if (((case_inputs_present) || (case_lookahead)) && (new output queue has room))
  {  if (case_lookahead == 1)
      { Determine output based on the controlling value of the gate;
        Absorb inputs up to element's lookahead_cnt;
        For (each input in the front of the queue of the element)
          { done_absorption = 0;
            while (!done_absorption)
              { /* keep absorbing until input's time stamp > element's lookahead_cnt */
                if (input queue is empty) done_absorption = 1;
                else {
                  if (input token has time stamp ≤ element's lookahead_cnt)
                    absorb input and send demand to fanin element
                   else done_absorption = 1
                  if ((input queue is not empty) && (! done_absorption))
                    {              /* check if to continue the lookahead cycle */
                      if (input == controlling value && time stamp > lookahead_cnt)
                        {done_lookahead = 0; done_absorption = 1;}
                      } else done_absorption = 1; }
              } /* end while loop in absorption */
          } /* end for each input ... */
      } /* end if case lookahead */

      if ((case_inputs_present) && (case_lookahead == 0))
       { current_controlling_output = 0;
         Determine lowest time stamp and evaluate output;
         Absorb inputs with lowest time stamp and send demand to fanin element(s)
         if (input queue is not empty && (front input has controlling value with time stamp > element's
         lookahead_cnt))
           done_lookahead = 0;}
    } /* end if case_lookahead or case inputs_present */
  } /* end while */
```

***—transmit phase***

(same as the transmit phase of conservative asynchronous algorithm of Appendix A)

## Authors' Biographies

Ausif Mahmood is currently an associate professor in the department of computer science and engineering at university of Bridgeport, CT. He received his M.S. and Ph.D. degrees in Electrical and Computer Engineering from Washington State University in 1982 and 1985 respectively. His research interests are in computer aided design for VLSI, and parallel and distributed computing.

William I. Baker is an adjunct assistant professor at Washington State University. He received his M.S. and Ph.D. degrees in Electrical Engineering from Washington State University in 1985 and 1994 respectively. His research interests are in CAD for VLSI and computer architecture.