Analysis of Parallelism and Deadlocks in Distributed-Time Logic Simulation

Larry Soule and Anoop Gupta

Technical Report No. CSL-TR-89-378

May 1989

This research has been supported by DARPA contract N00014-87-K0828. Authors also acknowledge support from Digital Equipment Corp. and a Graduate Fellowship from the National Science Foundation.

Analysis of Parallelism and Deadlocks in Distributed-Time Logic Simulation

Larry Soule and Anoop Gupta

Technical Report No. CSL-Tr-89-378

May 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 943054055

Abstract

This paper explores the suitability of the Chandy-Misra algorithm for digital logic simulation. We use four realistic circuits as benchmarks for our analysis, including the vector-unit controller for the Titan supercomputer from Ardent. Our results show that the average number of logic elements available for concurrent execution ranges from 10 to 111 for the four circuits, with an overall average of 68. Although this is twice as much parallelism as that obtained by traditional event-driven algorithms for these circuits, we feel it is still too low. One major factor limiting concurrency is the large number of global synchronization points --- "deadlocks" in the Chandy-Misra terminology -- that occur during execution. Towards the goal of reducing the number of deadlocks, the paper presents a classification of the types of deadlocks that occur during digital logic simulation. Four different types are identified and described intuitively in terms of circuit structure. Using domain specific knowledge, we propose methods for reducing these deadlock occurrences and give some preliminary results regarding the effectiveness of these methods. For one of the benchmark circuits, the use of the proposed techniques eliminated {\em all} deadlocks and increased the average parallelism from 40 to 160. We believe that the use of such domain knowledge will make the Chandy-Misra algorithm significantly more effective than it would be in its generic form. We also present some preliminary results comparing the performance of the Chandy-Misra algorithm versus a parallel event-driven algorithm.

Key Words and Phrases: Distributed Logic Simulation, Chandy- Misra algorithm, Deadlock.

Contents

Analysis of Parallelism and Deadlocks in Distributed-Time Logic Simulation

Larry Soule and Anoop Gupta
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

This paper explores the suitability of the Chandy-Misra algorithm for digital logic simulation. We use four realistic circuits as benchmarks for our analysis, including the vector-unit controller for the Titan super-computer from Ardent- Our results show that the average number of logic elements available for concurrent execution ranges from 10 to 111 for the four circuits, with an overall average of 68. Although this is twice as much parallelism as that obtained by traditional event-driven algorithms for these circuits, we feel it is still too low. One major factor limiting concurrency is the large number of global synchronization points — "deadlocks" in the Chandy-Misra terminology — that occur during execution. Towards the goal of reducing the number of deadlocks, the paper presents a classification of the types of deadlocks that occur during digital logic simulation. Four different types are identified and described intuitively in terms of circuit structure. Using domain specific knowledge, we propose methods for reducing these deadlock occurrences and give some preliminary results regarding the effectiveness of these methods. For one of the benchmark circuits, the use of the proposed techniques eliminated *all* deadlocks and increased the average parallelism from 40 to 160. We believe that the use of such domain knowledge will make the Chandy-Misra algorithm significantly more effective than it would be in its generic form. We also present some preliminary results comparing the performance of the Chandy-Misra algorithm versus a parallel event-driven algorithm [1].

1 Introduction

Logic simulation is a very common and effective technique for verifying the behavior of digital designs before they are physically built. A thorough verification can reduce the number of expensive prototypes that are constructed and save vast amounts of debugging time. However, logic simulation is extremely time consuming for large designs where verification is needed the most The result is that for large digital systems only partial simulation is done, and even then the CPU time required may be days or weeks. The use of parallel computers to run these logic simulations offers one promising solution to the problem.

The two traditional parallel simulation algorithms for digital logic have been (i) compiled-mode simulations and (ii) centralized time event-driven simulations. In compiled-mode simulations, each logic element in the circuit is evaluated on each clock tick The main advantage is simplicity, with the main disadvantage being that the processors do a lot of avoidable work, since typically only a small fraction of logic elements change state on any clock tick. The algorithm's simplicity makes it suitable for direct implementation in hardware [2,6], but

[1] A condensed version of the first part of this paper appears in the Proceedings of the 26th Degisn Automation Conference[12].

such implementations make it difficult to incorporate user-defined models or represent the circuit elements at different levels of abstraction. In the second approach of centralized time event-driven algorithms, only those logic elements whose inputs have changed are evaluated on a clock tick. This avoids the redundant work done in the previous algorithm, however the notion of the global clock and synchronized advance of time for all elements in the circuit limits the amount of concurrency [1,10,14]. These centralized time approachs work efficiently on multiprocessors with 10 nodes or so [9,11], but for larger machines we need alternative approaches that move away from this centralized advance of the simulation clock.

The approach generating the most interest recently is the Bryant/Chandy-Misra distributed time discrete-event simulation algorithm [3,4]. It allows each logic element to have a local clock, and the elements communicate with each other using time-stamped messages. In this paper, we explore the suitability of the Chandy-Misra algorithm for parallel digital logic simulation. We use four realistic circuits as benchmarks for our analysis. In fact, one of the circuits is the vector-unit controller for the Titan supercomputer from Ardent[7]. Our results show that the basic unoptimized Chandy-Misra algorithm results in an average *concurrency (by* concurrency we refer to the number of logic elements that could be evaluated in parallel if there were infinite processors) of 68 for the four circuits. For two of the benchmark circuits which were also studied in an earlier paper [10], the ***unoptimized*** Chandy-Misra algorithm extracted 40% and 107% more parallelism than the centralized time event-driven simulation algorithm.

The 68-fold average concurrency observed in the four benchmark circuits, however, is still too low. Once all the overheads (e.g. distributed deadlock detection, bus traffic, cache coherency, etc.) are taken into account, the 68-fold concurrency may not result in much more than 20 fold speed-up. One major factor limiting concurrency is the large number of global synchronization points — "deadlocks" in the Chandy-Misra terminology — that occur during execution We believe that understanding the nature of the deadlocks, why they occur and how their number can be reduced, is the key to getting increased concurrency from the Chandy-Misra algorithm. To this end, the paper presents a classification of the types of deadlocks that occur during digital logic simulation. Four different types are identified and described. Using domain specific knowledge, we then propose methods for reducing these deadlock occurrences. For one benchmark circuit, we show how using information about logic gates can eliminate all of the deadlocks. We believe that the use of such domain knowledge will make the Chandy-Misra algorithm significantly more effective than it would be in its generic form. To get an idea of how this average concurrency translates into real parallel performance, speed-up curves are presented for the algorithms running on a simulated 60-processor machine.

The organization of the rest of the paper is as follows. The next section describes the basic Chandy-Misra algorithm. Then we describe the four benchmark circuits that were simulated to get the measurements. Section 4 presents measurements of the parallelism extracted by the algorithm and Section 5 presents the classification of the deadlocks and ways for resolving them. Section 6 presents some speed-up curves comparing the Chandy-Misra algorithm and an event-driven algorithm along with some preliminary results on the proposed solutions for avoiding deadlock by exploiting knowledge of element behavior. Finally, Section 7 presents a summary of the results and discusses directions for future research

2 Background

We begin with a brief description of the basic Chandy-Misra algorithm [4] as applied to the domain of digital logic simulation. The simulated circuit consists of several circuit elements (transistors, gates, latches, etc) called *physical processes* (*PP*). *One* or more of these *PP*s can be combined into a *logical process* (*LP*), and it is

with these *LP*s that the simulator works.' Each different type of *LP* has a corresponding section of code that simulates the underlying physical processes (note that the mapping between *PP*s and *LP*s is often trivial in gate-level circuits, with each gate represented as a simulation primitive). Each of these *LP*s has a *local* time associated with it that indicates how far the element has advanced in the simulation. Different *LP*s in the circuit can have different local times, and thus the name distributed time simulation algorithm. Conceptually, each *LP* receives time-stamped event messages on its inputs and consumes the messages whenever all of the inputs are ready. As a result of consuming the messages, the logic element advances its local time and possibly sends out one or more time-stamped event messages on its outputs. Note that each input queue can conceptually hold any number of pending event messages. In the actual implementation, each "wire" is represented as a list of events and each input of an element just points into the event-list of the corresponding node.

As an example, consider a two-input AND-gate with local-time 10, an event waiting on input-l at time 20 (thus the value of input-l is known between times 10 and 20), and no events pending on input-2. In this state, the AND-gate process is suspended and it waits for an event message on input-2. Now suppose that it gets an event on input-2 with a time-stamp of 15. The AND-gate now becomes active, consumes the event on input-2, advances its local time to 15, and possibly sends an output message with time stamp 15 plus AND-gate delay.

We now introduce the concept of *deadlocks*. In the basic Chandy-Misra algorithm, even when input events are consumed and the local time of an *LP* is advanced, no messages are sent on an output line unless the value of that output changes. This optimization is similar to that used in normal sequential event-driven simulators where only elements whose inputs have changed are evaluated and it makes the basic Chandy-Misra algorithm just as efficient However, this optimization also causes *deadlocks in the* Chandy-Misra algorithm. In a deadlock situation, no element can advance its local time, because each element has at least one input with no pending events. We emphasize that this deadlock has nothing to do with a deadlock in the physical circuit, but it is purely a result of the optimization discussed above. The deadlock is resolved by scanning all the unprocessed events in the system, finding the minimum time-stamp associated with these events, and updating the input-time of all inputs with no events to this time (note that this deadlock resolution can also be done in parallel). Consequently, the basic Chandy-Misra algorithm cycles between two phases: the *compute* phase when elements are advancing their local time, and the *deadlock resolution* phase when the elements are being freed from deadlock.

One way to totally bypass the deadlock problem is to not use the optimization discussed above. Thus elements would send output messages whenever input events are consumed and the local rime of an element is advanced. This would be done even if the value on the output does not change. Such messages are called NULL messages in the Chandy-Misra terminology, as they carry only time information and no value information. Unfortunately, always sending NULL messages makes the Chandy-Misra algorithm so inefficient that it is not a good alternative to avoiding deadlocks. However, in this paper we show how selective use of NULL messages can significantly reduce the number of deadlocks that need to be processed.

Regarding parallel implementation of the Chandy-Misra algorithm, since each element is able to advance its local *time independently* of other elements, all elements can potentially execute concurrently. However, only when all inputs to an element become ready (have a pending event), is the element marked as available for execution, and placed on a distributed work queue. The processors take these elements off the distributed queue, execute them, update their outputs, and possibly activate other elements connected to the outputs. This happens until a deadlock is reached, when the deadlock resolution procedure is invoked.

[2] In this paper the terms *LP* and element are used interchangeably.

3 Experimental Environment

This section describes the circuits used to evaluate the algorithms, the details of the simulator (e.g. the delay model used), and the hardware the experiments were run on.

3.1 Benchmark Circuits

In this section, we first provide a brief description of the benchmark circuits used in our study and then some general statistics characterizing these circuits. The four circuits that we use are:

1. Ardent-1: This circuit is that of the vector control unit (VCU) for the Ardent Titan graphics supercomputer[7]. The VCU is implemented in a 1.5μ CMOS gate array technology and it provides the interface among the integer processing unit, the register file, and the memory. It also allows multiple scalar instructions to be executed concurrently by scoreboarding. It consists of approximately 45,000 two-input gates.

2. H-FRISC: A small RISC generated by the HERCULES [8] high-level synthesis system from the 1988 High Level Synthesis Workshop. The RISC instruction set is stack based and fairly simple. This circuit consists of approximately 11,000 two-input gates.

3. Multiplier: This circuit represents the inner core of a custom 3μ CMOS combinational 16x16 bit integer multiplier. Multiplies are pipelined and have a latency time of 7Ons. The approximate complexity is 7,000 two-input gates.

4. 8080: This circuit corresponds to a TTL board design that implements the 8080 instruction set. The design is pipelined, runs 8080 code at a speed of 3-5 MIPS, and provides an interface that is "pin-for-pin" compatible with the 8080. The approximate complexity is 3,000 two-input gates.

We note that the benchmark circuits cover a wide range of design styles and complexity — we have a large mixed-level synchronous gate array; a medium gate-level synthesized circuit; a medium gate-level combinational chip; and a small synchronous board-level design The fact that we have both synchronous pipelined circuits and totally combinational circuits is also important, because they exhibit very different deadlock behavior during simulation.

We now present some general statistics for these benchmark circuits in Table 1. The statistics consist of:

- Element count: The number of primitive elements (*LP*s) in the circuit One expects the amount of concurrency in the circuit to be positively correlated with this number (see Table 2).
- Element complexity: This is defined as the number of equivalent two input gates per primitive element and roughly correlates to the grain size of the application. The number of primitive elements multiplied by the element complexity gives a more uniform measure for the circuit complexity. The element complexity gives an indication of the compute time required to evaluate a primitive element, and thus specifies the grain of computation
- Element fan-in and fan-out: The average number of inputs/outputs of an element. These numbers are also correlated to the element complexity. If the average number of inputs is high, one would expect a higher probability of deadlock as there are more ways in which one of the inputs may have no event.
- Percentage of combinational and synchronous elements. (Note that synchronous elements have internal state)

Table 1: Basic Circuit Statistics

statistic	Ardent-l	H-FRISC	Mult-16	8080
Element Count	13349	8,076	4,990	281
Element Complexity	3.4	1.4	1.4	12.0
Element Fan-m	2.72	2.14	2.14	5.78
Element Fan-out	1.2	1.0	1.0	2.63
% Synchronous Elements	11.2	2.8	0.0	16.7
% Comb. Elements	88.8	97.2	100	83.3
Net Count	13,873	8,093	5,077	748
Net Fan-out	2.66	2.14	2.14	5.48
Representation	gate/RTL	gate	gate	RTL
Basic Unit of Delay	0.5ns	unit	1ns	1ns

- Net count: The number of wires in the circuit,
- Net fan-out: The average number of elements a wire is attached to. The Ardent and 8080 circuits have some global buses that affect many components thus their net fan-out numbers are high.
- Representation: The level of representation of the simulation primitives. A circuit made up of only logic gates and one-bit registers is at the gate-level while a design made up of TTL-chip components is at the RTL-level.
- Unit of delay: Each output pm of each element is assigned a delay value at netlist-compile time. The delay is specified in terms of the basic unit of delay.

3.2 Simulation Model

A fixed delay model is used and all delays are greater than zero. This means that each output pm of each element is assigned a non-zero delay value at netlistcompile time. The propagation delays are all specified in terms of the basic unit of delay (see Table 1). Traditional event-driven simulators usually implement a more complex delay model including inertial delays and spike analysis. These are implemented by canceling events already scheduled in the future. These accuracy enhancements are more complicated to implement in a *conservative* distributed-time simulation scheme since canceling an event violates the basic premise of "send an event only when you're *sure* it's ok". However, inertial delays and spike analysis can be implemented by adding some code to the elements and nodes in question (see Appendix A for implementation details). This could cost as much as 50% in terms of performance since the node evaluations take longer and may introduce deadlocks. If spike analysis or usage of inertial delays can be limited to only a portion of the circuit, the performance impact would be minimal.

3.3 Hardware Details

Implementations in two environments were used in collecting the results. One implementation runs on the Encore Multimax, a shared-memory multiprocessor with sixteen NS32032 processors, each processor delivering

approximately 0.75 MIPS. The other implementation runs under a simulated multiprocessor environment called Tango [5].

3.3.1 Multiprocessor Simulation With Tango

Tango is a software tracing system that helps in evaluating parallel applications and multiprocessors. Tango simulates multiprocessors by creating a real process for each process in the application code. These processes are then multiplexed onto a single processor. With this method, the system can simulate the timing behavior of the parallel application at three levels: at each global synchronization points, at each shared memory reference, or at each memory reference. This is done by associating a local clock with each user process being traced. The clocks are updated either at the end of each basic block, at each synchronization point, or at each memory reference depending on the desired level of timing detail. Tango schedules the processes in such a way that ensures correctness between the process clocks at the level of the timing detail (the trade-off being wallclock time versus timing accuracy). The methods used in Tango are not affected by the programming model or the architecture model.

4 Parallelism Measurements

In this section, we discuss how parallelism is exploited by the Chandy-Misra algorithm and present data regarding the amount of concurrency available in the four benchmark circuits. We also present data regarding the granularity of computation, the number of deadlocks per clock cycle, and the amount of time spent in deadlock resolution.

Since we are interested in the parallel implementations of the Chandy-Misra algorithm, the first question that arises is how much speed-up can be obtained if there were arbitrarily many processors, and if there were no synchronization or scheduling overheads. We call this measure the *concurrency* or intrinsic parallelism of the circuits under Chandy-Misra algorithm. For our concurrency data, we further assume that all element evaluations take exactly one unit of time. Thus, the simulation proceeds as follows. After a deadlock and the ensuing deadlock resolution phase, all elements that are activated (i.e., have at least one event on each of their inputs) are processed. This happens in exactly one unit-cost cycle as we assume arbitrarily many processors. The number of elements that are *evaluated constitutes the* concurrency for *this iteration*. The evaluation of the elements, of course, results in the activation of a whole new set of elements, and these are evaluated in one cycle in the next iteration, The computation proceeds through these iterations until a deadlock is reached, and we start all over again Typically, four or five iterations are completed before a deadlock is reached.

Figure 1 shows the concurrency data (shown using the dashed line) and event profiles (shown using the solid line) for the four benchmark circuits. The event profiles show a plot of the total number of logic elements evaluated between deadlocks. The profiles are generated over three to five simulated clock cycles in the middle of the simulation. We would like to reemphasize that the profiles in Figure 1 are not algorithm independent, but are specific to the basic Chandy-Misra algorithm. In fact, our research suggests enhancements to the basic Chandy-Misra algorithm, so that much more concurrency may be observed,

The profiles clearly show cyclical patterns with the highest peaks corresponding to the system clock(s) of the simulated `circuits`, and the portions between the peaks corresponding to the events propagating through the combinational logic between the sets of registers. The Ardent profile shows that the circuit quickly stabilizes after the clock with only a few deadlocks, while the multiplier, with many levels of combinational logic, takes quite a while to stabilize with many deadlocks. This close correspondence between the event profiles and the circuit being simulated shows the importance of exploiting domain specific information; any circuit characteristic

Table 2: Simulation Statistics

statistic	Ardent-l	H-FRISC	Mult-16	8080
Unit-cost Parallelism	107	111	45	10
Granularity (ms)	0.74	0.66	0.75	2.61
Deadlock Ratio	308	260	248	15
Cycle Ratio	1,644	1,999	6,712	132
Avg. Iterations Btw. Deadlock	3	4	5	2
Deadlocks Per Cycle	4.5	7.7	27.1	8.9
Avg. Deadlock Resl. Time (ms)	520	230	206	11
% Time in Deadlock Resl.	58	46	41	19

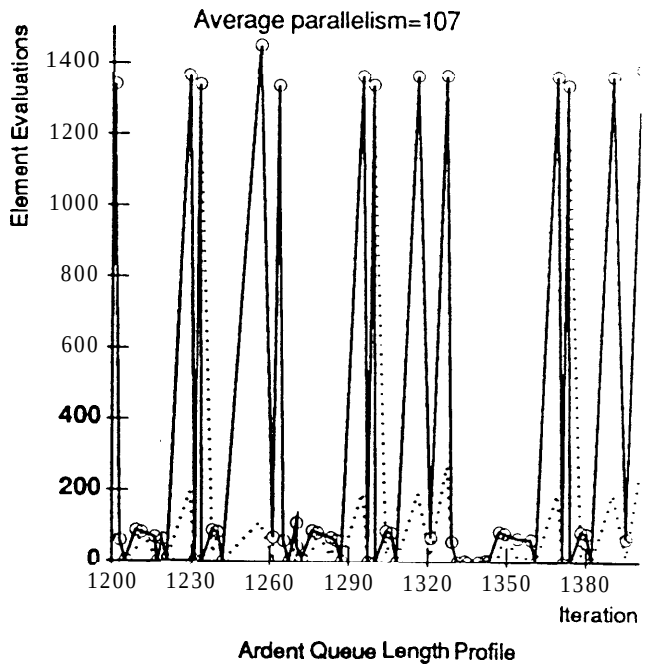
we change or exploit will be directly reflected in the event profiles. Understanding how these changes affect the profiles and beiig able to predict them is important in obtaining better performance. A summary of the concurrency information is also presented in Table 2. The top line of the table shows the concurrency as averaged over all iterations in the simulation.

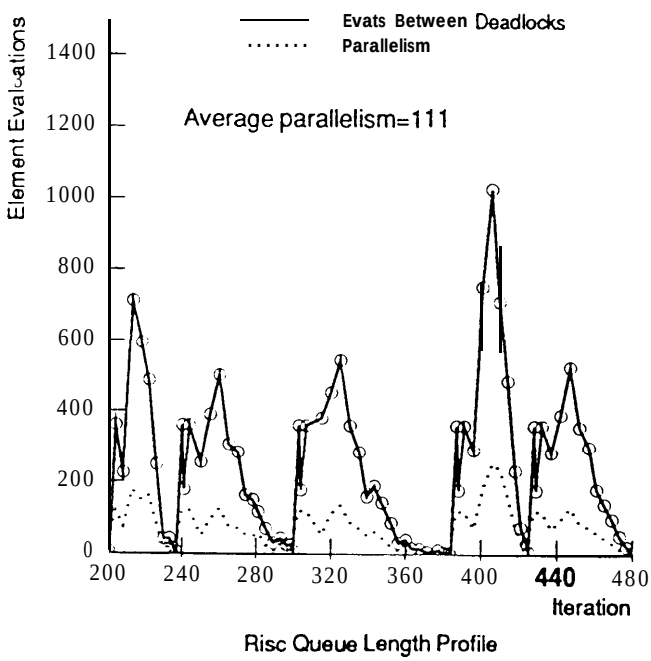
In addition to knowing how many concurrent element evaluations or tasks are available, we also need to know the task granularity and how often deadlocks (global processor synchronizations) occur. The granularity or basic task size for our application (a model evaluation) includes checking the input channel times, executing the model code, calculating the least next event and possibly activating the elements in its fan-out, The numbers discussing task granularity and frequency of deadlocks are summarized in Table 2. The table also presents the following ratios that help characterize the performance of the Chandy-Misra algorithm:

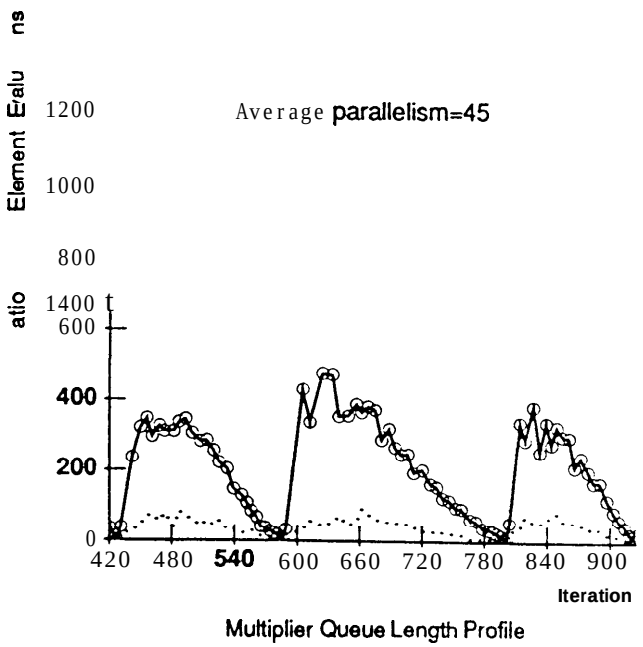
- Deadlock ratio **(DR):** Number of element evaluations divided by the number of deadlocks.
- **Cycle** ratio (CR): Number of element evaluations divided by the number of simulated clock cycles.
- Deadlocks per cycle: **Number** of deadlocks divided by the number of simulated clock cycles.

Since increased parallelism was the main motivation for using the Chandy-Misra algorithm, we now compare the concurrency it obtains to that obtained using a traditional event-based algorithm. For our comparison, we **use the concurrency** data presented for the 8080 and multiplier circuits in a parallel event-driven environment in **[11,10].** These papers showed that the available concurrency was about 3 for the 8080 and 30 for the multiplier. From Table 2, the corresponding numbers for the Chandy-Misra algorithm are 10 for the 8080 and 45 for the multiplier. The fact that the concurrency increases only by a factor of 15 for the multiplier and 3.3 for the **8080** (only bringing it up to 10) is somewhat disappointing, since Chandy-Misra algorithm is more complex to implement, **However,** we show that using the techniques proposed in the next section, the Chandy-Misra algorithm can be suitably enhanced to show much higher concurrency.

The last two lines of Table 2 give data about the average time taken by each call to deadlock resolution and the total fraction of time spent in deadlock resolution. The cost of resolving a deadlock for the three larger circuits is indeed high, especially when compared to the cost of evaluating a logic element (see the granularity line). In the time it takes to resolve one deadlock in the Ardent simulation, 700 logic element activations could have been processed. For the H-FRISC, 350 elements could have been evaluated, and in the multiplier, 275 elements could have been evaluated. However, for the 8080 with 281 elements, the deadlock resolution overhead is not







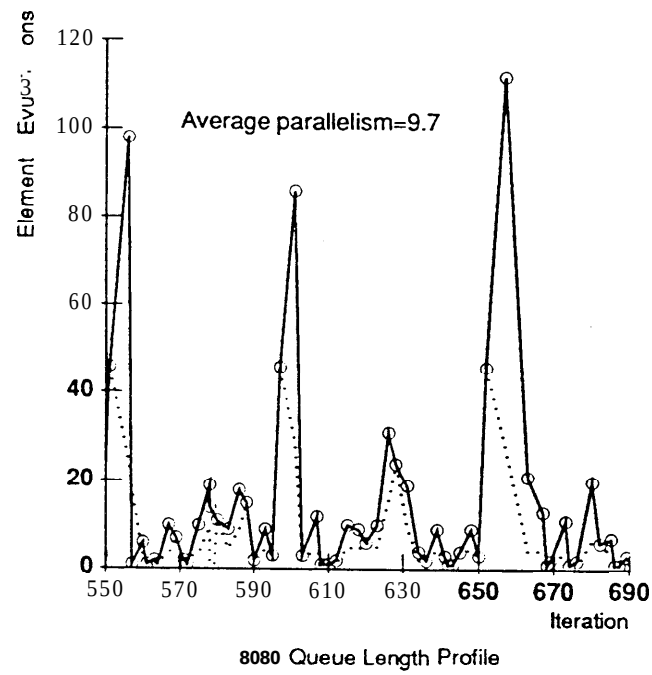


Figure 1: Event Profiles

too high taking the time of only 4 element evaluations. This is because there are so few elements (only 1/40 as many elements as the Ardent-l) to be checked and because each evaluation of an RTL element is about 3-4 times longer than the evaluation of a typical logic gate. Unfortunately the trend that emerges is: the more elements in the circuit, the larger the fraction of execution time consumed by deadlock resolution. This remains a problem, but in our research, we are also exploring techniques to reduce the deadlock resolution time significantly by caching information from previous simulation runs of the same circuit. No results are available at this time.

5 Characterizing Deadlocks

Even though there is reasonable parallelism available in the execution phase of the Chandy-Misra algorithm, deadlock resolution is so expensive in the larger circuits that it consumes 40-60% of the total execution time. Clearly we have to reduce this percentage in order to get good overall parallel performance. The first step towards this reduction is understanding why deadlocks occur and how they can be avoided. The types of deadlock that occur in logic simulation are characterized in this section and this characterization gives us insight into what aspects of logic simulation can be effectively exploited to achieve good overall performance.

In the logic simulations that were studied, the elements that became deadlocked can be put into two categories: (i) those deadlocked due to some aspect of the circuit structure (e.g topology, nature of registers, feed-back) and (ii) those deadlocked due to low activity levels (e.g. typically only 0.1% of elements need to be evaluated on each time step in eventdriven simulators[10]). In the following subsections, descriptions and examples of each of the types of deadlock are given, along with measurements that show how much each type contributes to the whole.

5.1 Registers and Generator Nodes

In a typical circuit, enough time is allowed for the changes in the output of one set of registers to propagate all the way to the next set of registers in the datapath and stabilize before the registers are clocked again For example, in Figure 2, the critical path of the combinational part of the circuit is 82ns, and the clock node changes every 100ns to allow everything to stabilize. *Reg1* is clocked at the start of the simulation, and the events propagate through the combinational logic, generating an event at time 82. This event at time 82 is consumed by *Reg2* since the clock node is defined for all time in this example. However, the next event for *Reg2* at time 100 is *not* consumed since the input to the latch is only defined up to time 82, not 100. This causes *Reg2 to* block and the deadlock resolution phase is entered

Generator deadlocks are similar to register deadlocks. Generators supply the external stimulus (e.g. system clocks, reset, inputs, etc.), and usually have no inputs. Since they do not have any inputs, they can be evaluated for all time at the beginning of the simulation. The events output by these generators cause deadlocks just like the register clocks in the previous example. Generators and register clocks combine to be a very large source of deadlocks since most circuits have many registers, latches and generator nodes.

In Table 6 we see that for the Ardent, registerclock deadlocks account for 92% of all the elements activated in the deadlock resolution phase even though registers only make up 11% of the elements. This is mainly due to the pipelined nature of the Ardent design where there is only a small amount of combinational logic between register stages. In the case of the RISC design, there is more combinational logic between the registers than the Ardent and more logic gates connected to the input stimulus generators. Thus registerclock and generator deadlocks both cause around 20% of the deadlock activations for a total of 40%. In the multiplier design, there are many levels of logic between the inputs and outputs and does not have any registers. Thus there can not be

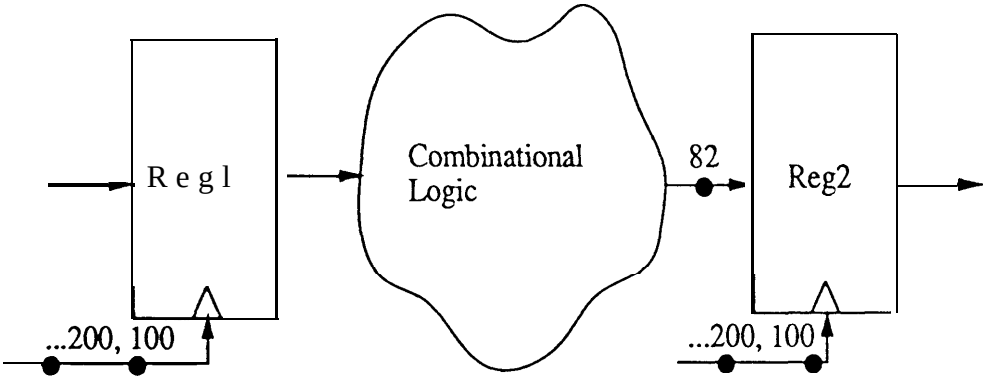


Figure 2: Deadlock Caused by a Clocked Register

Table 3: Register-Clock and Generator Deadlocks

Circuit	Total Deadlock Activations	Register-clock Activations	% of Total	Generator Activations	% of Total
Ardent-1	316.0k	290.0k	92	583	0.2
H-FRISC	45.6k	8.9k	20	8,800	19.0
Mult-16	27.2k	0.0k	0	40	0.1
8080	8.3k	4.6k	55	53	0.6

any register-clock deadlocks and very few generator deadlocks. The 8080 design, like the Ardent, is pipelined and hence register-clock deadlocks are the main source of deadlock. Here 55% of the activations are caused by register-clock deadlocks while only 17% of the elements are registers.

To identify *this type* of deadlock, a *register-clock* ***deadlock*** is said to occur whenever a clocked element *LP,* that is activated during deadlock resolution has the earliest unprocessed event on its clock input A ***generator*** *deadlock is* said to occur whenever the earliest unprocessed event was received directly from a generator element.

5.1.1 Proposed Solutions

Taking advantage of behavior: In general, **an** input event may arrive at any time in an element's future causing it to change its **output.** Thus, an element can only be sure of its outputs up to the minimum time its inputs are valid plus the output delay. In the case of registers and latches, however, we know that the output will not change until the next event occurs on the clock input regardless of the other inputs. This knowledge of input ***sensitization*** *is easy* to use and potentially very effective since the outputs can be advanced up to the next clock cycle. In registers and latches with asynchronous inputs (like set, clear, etc.), those inputs must be taken into account as well as the clock node when determining the valid time of an output.

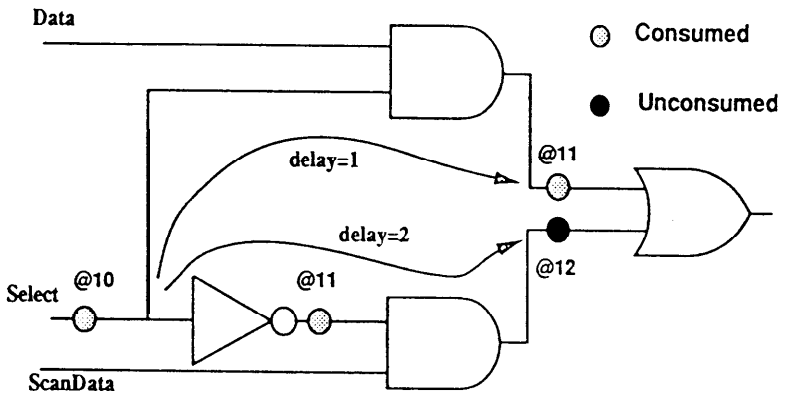


Figure 3: Deadlock Due to Paths of Different Delays

Fan-out Globbing: This technique reduces the overhead and the time needed to perform deadlock resolution Recall that a particular *LP* is composed of many *PP*s. These *PP*s can be combined in different ways to form larger units. Combining many registers that share the same clock node will reduce the overhead of activating each register separately. Typically hundreds of one-bit registers and gates are connected to the clock node(s) and often times during deadlock resolution, the minimum event is on the clock node (as in the example above). If *we* combine *these* registers and gates *in* groups of *n, we* call *this* grouping *fan-our globbing with a clumping factor of n* since we are combining the fan-out elements of the clock This reduces the overhead of inserting and deleting the elements in the evaluation queue. However, since it combines elements, it also reduces the parallelism available. We are currently looking into just how much reduction in overhead and parallelism this causes.

5.2 Multiple Paths with Different Delays

Whenever there are multiple paths with different delays from a node to an element, there is a chance of that element deadlocking. An example of this is the MUX shown in Figure 3. There are two paths from the Select line to the OR-gate at the output. If the Data and ScanData lines are valid, an event on the Select line could propagate through the two paths and generate events at times 11 and 12. The event at time 12 will not be consumed by the OR-gate since its other input is only defined up to time 11 causing the OR-gate to deadlock. Thus, multiple paths from a node to an element can result in an unconsumed event on the path with the larger delay.

TO detect this type of deadlock, let LP_i *be the* deadlocked element and j be the index of the input with the unprocessed event. Then if there are two different paths from some element, LP_k, to the deadlocked element, LP_i, with *the* longer path ending at input j, then a *multiple path* deadlock has occurred.

5.2.1 Proposed Solutions

Since this type of deadlock is due to the local topology of the circuit, there is no easy way of avoiding it. However, there are a couple of options.

Demand-driven: The elements that are affected by multiple paths could be marked either while compiling the netlist or from previous simulation runs. When these elements are executed, a demand driven technique could be used. With a demand-driven technique, whenever an element can not consume an input event, requests are made to its other fan-in elements (the ones driving its input pins) asking "Can I proceed to this time?". These requests propagate backwards until a yes or no answer can be ensured. Propagating these requests can be expensive especially if the fan-in factor for the elements is large or there are long feedback chains in the circuit Thus we must be very selective in the elements we choose to use this technique with.

Structure globbing: If there are not too many elements involved in the multiple paths, we may be able to *hi&* the multiple paths by globbing those elements into one larger *LP*. However, the composite behavior of the gates must be generated and the detailed timing information must be preserved. Preserving the exact timing information is non-trivial. In essence a state variable must be made for each of the internal nodes and the element may have to schedule itself to make the outputs change at the correct times. This self-scheduling may cause the element to deadlock because, by requesting itself to be evaluated at some time, it must wait until the inputs are valid up to that time just as before. If the detailed timing information does not need to be preserved, the composite behavior is easy to generate (compiled-code simulation techniques can be used on the small portion of the circuit that is being globbed together) and this deadlock type will be avoided.

Taking advantage of behavior: If we know the behavior of an element, it may be possible to advance that element even though some of its inputs are not known. For example in Figure 3, if the event at time 12 going into the OR-gate has a value of 1, the output is known to be 1 regardless of the value of the other input and the OR-gate need not deadlock. In a gate-level simulation, the behavior of most of the elements is very simple and can be readily exploited

5.3 Order of Node Updates

The activation criteria for the basic Chandy-Misra algorithm is: activate an element only when an event is received on one of its inputs. Sometimes this activation criteria can cause a consumable input event to be stranded due to the order in which the node updates are performed. This stranded event will cause the element to deadlock In Figure 4, element el consumes the event at time 10, produces an event at time 11, and activates element e3. If e3 is now executed, e3 will not be able to consume the event at time 11 because the input from e2 will not be valid at time 11. If an event at time 10 now arrives at e2 and e2 is evaluated, it will update the valid-time of the input to e3 but it will not activate e3 because no event was generated If e3 had waited for e2 to be evaluated, the inputs to e3 would have both been valid at time 11 and the event at time 11 could have been consumed.

In Table 6, we see that the order of node updates type of deadlock is uncommon in the Ardent simulation which is dominated by the registerclock deadlocks. The order of node updates is, however, important in the combinational multiplier with its many levels of logic.

To detect this class of deadlock, suppose $L\ P_i$ is activated during deadlock resolution because it was waiting to consume an event at time t. If *all* of the input nodes to $L\ P_i$ are found to have advanced up to time t — that is if the element can safely consume the event without *any* input times being updated, an *or&r of node updates* deadlock has occurred.

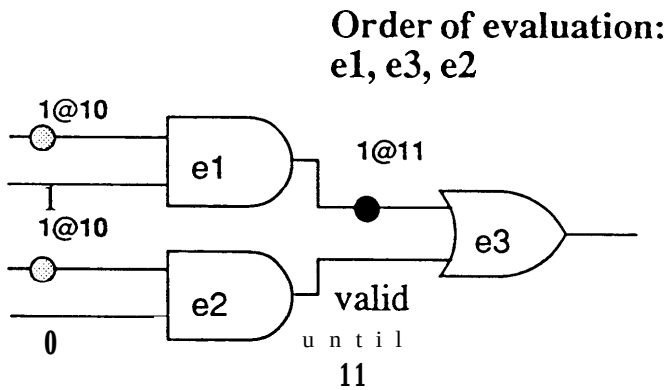


Figure 4: Deadlock Caused by Order Of Node Updates

Table 4: **Deadlock Activations** Caused by the Order of Node Updates

Circuit	Deadlock Activations	Order of Node Updates	% of Total
Ardent- 1	316.0k	1.4k	0.4
H-FRISC	45.6k	1.0k	2.2
Mult-16	27.2k	1.7k	6.2
8080	8.3k	0.2k	2.2

53.1 Proposed Solutions

New activation criteria: The problem is that the activation criteria does not activate an element when the valid times of its input node are updated. The problem can be eliminated if an element checks its fan-out elements when it updates the time of its output nodes. Any of those fan-out elements that have a real event at a time less than or equal to the new valid-time, should be activated. In the example, e2 would activate e3 when it updated the valid-time of its output to 11 since e3 has a real event at time 11. Note that this only works for the case where the updated node is directly connected to the element with the unconsumed event. If there are any intermediate elements the deadlock is considered to be caused by an unevaluated path which is explained in the next subsection. If e3 had a third input, it still may not be able to consume the event at time 11 even after e2 is evaluated This extra activation creates needless work and the effectiveness of this solution depends on the relative cost of performing a deadlock resolution on the particular circuit being simulated.

Rank ordering: The *rank* of an element is the maximum number of levels of logic between the element and any registers. It can be computed by assigning all registers and generator elements a rank of 0 and then iterating through the combinational elements assigning them a rank of one plus the maximum rank of its input elements. If the elements in the evaluation queue are ordered by their rank, the node updates will proceed in a more ordered fashion (i.e. elements farther away from the registers and external inputs that affect it will be evaluated later possibly letting their inputs become defined). In the example, e2 would be inserted before e3 since the inputs to e3 depend on the outputs of e2. Since the rank information is easy to compute while compiling the netlist, *the run-time* cost is very little. Also, this technique may actually eliminate some extra activations so the overall cost is cheap.

. 5.4 Unevaluated Path

The elements in the fan-out of a wire are activated only when a real event is produced on that wire. Thus, if element LP_i consumes an event but does not produce a new event (i.e. the activation does not result in a change in value of output signals), all paths from *LP*, to the other elements will not be evaluated or updated. Figure 5 shows the case where one event is consumed and, since no new event is produced, the OR-gate is not activated and the second AND-gate can not consume the event at time 11 since the valid-time of the input from the OR-gate was not updated

In Table 6 we see that unevaluated paths (labeled One Level NULL and Two Level NULL depending on how far back the paths were not evaluated) are very important in three of the four circuits. This is especially true for the RISC and multiplier designs which consist of many levels of combinational elements. For the RISC, the number of deadlocks caused by unevaluated paths is around 60% and that for the multiplier around 90%. In contrast, unevaluated paths are relatively unimportant in simulations of the pipelined Ardent design

If NULL messages are *always sent., the* simulation will never deadlock (see Section 2). Unfortunately, this is highly inefficient since typical activity levels in event-driven simulators are around 0.1% in each time step (actual run times range from 30 times slower to more than 100 times slower). To find out how many deadlocks we could avoid by only *selectively* sending NULL messages, we did the following. We measured how many deadlock activations would have been avoided if every deadlocked element had received NULL messages from its immediate fan-in — corresponding to what we call "one level" of NULL messages — and how many activations would have been avoided by two levels of NULL messages.

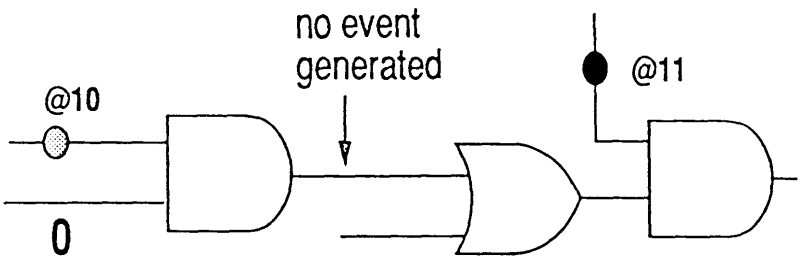


Figure 5: Deadlock Caused by Unevaluated Path

Table 5: Deadlock Activations Caused by Unevaluated Paths

circuit	Deadlock Activations	One Level NULL	%of Total	Two Level NULL	% of Total	Combined %
Ardent-1	308.0k	81.9k	26	221.9k	72.0	98
H-FRISC	45.8k	35.7k	78	1.2k	2.6	81
Mult-16	27.2k	21.6k	79	1.6k	5.9	85
8080	8.3k	2.6k	32	0.9k	105	42

Table 6: Deadlock Activations Classified by Type

Circuit	Total Deadlock Activations[3]	Reg-clk Actv.	Generator Actv.	Order of Node updates	One Level NULL	Two Level NULL	Unaccounted
Ardent-1	308.0k	280.8k	0.8k	1.6k	81.9k	221.9k	*543*
RISC	45.8k	8.9k	8.9k	1.2k	35.7k	1.2k	0
Mult	23.4k	0.0k	0.0k	1.6k	21.6k	1.6k	*288*
8080	8.3k	4.6k	4.6k	0.2k	2.6k	0.9k	*265*

54.1 Proposed Solutions

Caching: Since the activity levels are so low, we need to be very selective about which elements should send NULL messages. The proposed selection process follows the concept of *caching*. By caching information from previous runs, we can identify the elements that repeatedly deadlock due to an unevaluated path as the simulation progresses. When these elements get activated, they will send out NULL messages whenever their outputs times advance. In order to be effective, the caching algorithm must be quick and efficient.

Taking advantage of behavior: See Section 52.1 for a description. As it turns out, this technique works very well for the combinational multiplier circuit. It eliminates *all* deadlocks and increases the parallelism from *45 to 160*.

5.5 Summary of the Contributions From Each Deadlock Type

A summary of the composition of deadlocks for the benchmark circuits is given in Table 6. In all but the Ardent and 8080 circuits, the main deadlock type is the one-level NULL caused by unevaluated paths which are, in turn, caused by the very low activity levels in digital logic simulations. The Ardent and 8080 deadlocks are made up predominantly of register-clock deadlocks. They account for 91% and 55% of the deadlock activations, respectively, even though synchronous elements comprise only 11 to 17% of the total elements. This is mainly due to the heavily pipelined nature of the two circuits — many of latches with only a few levels of logic in between. Thus most of the deadlocks occur when the registers and latches are waiting for their inputs to become valid.

The main contributors to deadlock in the RISC circuit, after the one-level NULL deadlocks, are generator and registerclock deadlocks. This is due to the consistent control style used by the synthesis system. The system clocks are generated externally and first pass through a level of logic that controls which parts of the design are active. These qualified clocks are then distributed to their corresponding circuit sections — the result being that most registers are waiting on their inputs and the elements connected to the generator nodes are waiting on their other inputs.

In contrast to the other three circuits, the multiplier design is highly interconnected with many levels of combinational logic. Almost all of the deadlock activations are caused by the unevaluated paths in the circuit as shown by the two-level NULL column This is caused by a few paths that are active all the way from the inputs to the outputs while most of the paths do not have any activity at all after the first couple of levels.

[3] Each activation may have more than one cause so the sum of the deadlock activation types may be more than the total.

6 Preliminary Results

With the classifications of deadlock types presented in the previous section, we are now ready to evaluate the effectiveness of the proposed solutions for avoiding deadlock The results and analysis presented in this section are a collection of the preliminary results we have obtained. Wherever possible, these results are related to aspects of the circuit being simulated or effects particular to digital logic simulation. The structure of this section is as follows. We first present speed-up curves for the event-driven algorithm and Chandy-Misra algorithm on a simulated 60-processor machine. Next, we quantify the overheads incurred by the Chandy-Misra algorithm in relation to the event-driven algorithm. Then we compare the achieved concurrency versus the maximum concurrency as determined by the event distribution and structure of the circuit We then detail how we take advantage of element behavior and how effective it is in reducing execution times and deadlock activations. Finally we give results on the effectiveness of a one-level demand driven scheme and a global event counting scheme.

6.1 Speed-up Curves

The first step in analyzing the Chandy-Misra algorithm and the various optimizations is to see how the ideal concurrency presented earlier translates into actual parallel performance and how this performance compares to the parallel performance of the event-driven algorithm. This comparison is done with the help of the speed-up curves presented Figure 6. The four speed-up curves compare the generic Chandy-Misra implementation with an eventdriven simulator described in [11]. The "Relative Speed" axis on each plot is normalized to the generic Chandy-Misra algorithm running on one processor. The simulations were done using Tango[5], a simulated multiprocessor environment described in detail in Section 3.3.1. The results for each circuit will now be discussed in turn.

The Ardent-l simulation shows the most favorable performance comparison for the Chandy-Misra algorithm over the event-driven algorithm. Although the uniprocessor Chandy-Misra performs only about one-third as fast as the event driven algortihm, the Chandy-Misra algorithm exceeds the best event-driven performance when four or more processors are used. Furthermore, the Chandy-Misra algorithm achieves a relative speed around 37 with 60 processors while the event-driven algorithms best speed is only 3.5. The poor performance of the eventdriven algorithm is due to the distribution of the events across simulation time. A typical clock cycle consists of one or two time-steps with around 1000 events and about thirty time-steps with 1-10 events. Since the event-driven algorithm is forced to only look at one time-step at a time, the thirty or so time-steps with only l-10 events rum the parallel performance (recall that each time-step also causes a global synchronization of all the processors). Thus, the event-driven algorithm is unable to exploit most of the parallelism uncovered by the Chandy-Misra algorithm.

For the H-FRISC circuit, the performance for the event-driven algorithm starts out about six times faster than the Chandy-Misra algorithm and gets good speed-ups up to four processors. With more than four processors, the global synchronization of the processors at every time-step and low number of events in each time-step limits the performance to about 13 relative to the Chandy-Misra algorithm. Even though the Chandy-Misra starts at about one-sixth the speed of the event-driven, it gets relatively good speed ups up to 24 processors, crossing the event-driven performance at about 16 processors. With 32 processors, the Chandy-Misra algorithm performs 30-40% better than the best event-driven performance.

The Mu-h-16 simulation shows the most favorable results for the event-driven algorithm with its best parallel performance achieving about twice the relative speed of the best parallel performance of the Chandy-Misra algorithm. The event-driven relative performance tops out at 54 with 24 processors. With more than 24

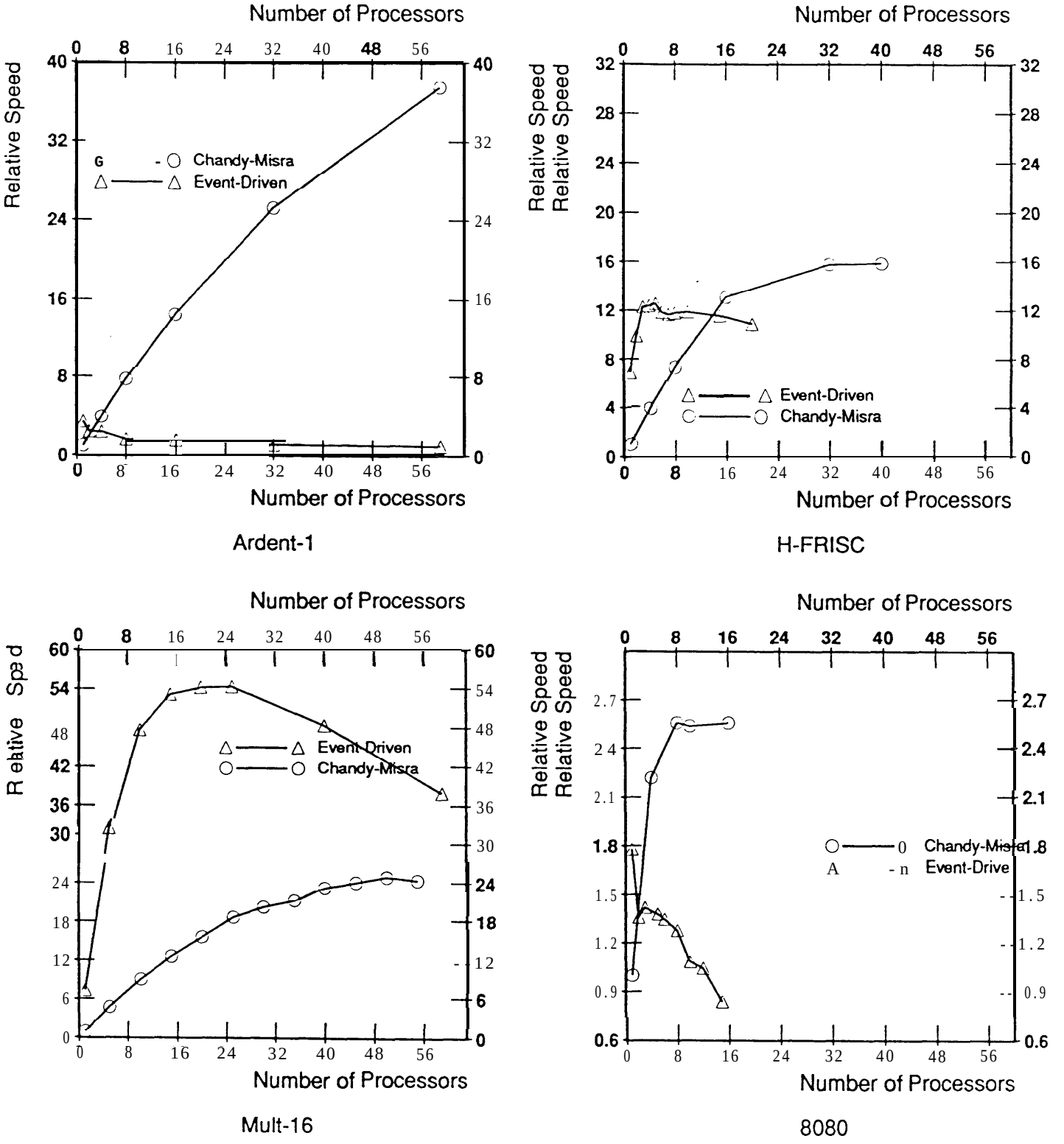


Figure 6: Speed-Ups

processors, there are not enough available concurrent events relative to the cost of synchronizing and load imbalance. The poor Chandy-Misra performance, reaching 24 with 48 processors, is due to the large number of deadlocks that cause global processor synchronization (there are about 27 deadlocks per cycle - Section 6.4 shows how all deadlocks in this simulation can be eliminated allowing the Chandy-Misra algorithm to perform twice as fast as the event driven algorithm).

For the 8080, both algorithms perform poorly. The event-driven algorithm starts out about twice as fast but achieves no speed-up with added processors. The Chandy-Misra algorithm achieves a 2.5 fold speed-up with 8 processors and not much more as more processors are added. The reason for the poor performance for both implementations is the very low number of elements. The 8080 circuit only has 281 elements -- there just isn't much to work with.

6.2 Overheads in the Chandy-Misra and Event-Driven Algorithms

In the preceeding section, we observed that on a uniprocessor, the event-driven algorithm is roughly 2-6 times faster than the Chandy-Misra algorithm for the four circuits. This performance penalty is very large and, for the Chandy-Misra algorithm to be practical, it must be reduced significantly. We are currently working on ways of reducing this overhead and have found that it consists of three components: deadlock resolution time, more complex element evaluations, and extra element evaluations. The contribution of each of these components is detailed in Table 7. We see that in the larger circuits, deadlock resolution accounts for 40-60% of the total execution time. This alone causes a 2-fold slowdown over the event-driven algorithm. Secondly, each element evaluation is more complex than in an event-driven scheme. Both algorithms must compute the element behavior, but the Chandy-Misra algorithm must also compute which input events can be consumed during the current evaluation. Table 7 shows that the average element evaluation in the Chandy-Misra environment takes about 86% longer than the corresponding evaluation in an event-driven environment. The last component of overhead is an average of 26% more element evaluations. As an example of how these extra evaluations occur, consider an AND-gate receiving an event at time 10. This event will cause the gate to be scheduled for evaluation. However, suppose this event can not consumed by the gate due to another of the gates inputs lagging behind in simulation time. The gate must then either wait for deadlock resolution to advance all nodes to time 10 or wait for another event to arrive on one of its inputs. In either case, the original evaluation (caused by the receipt of the event at time 10) did not consume any events or advance the local time of the AND-gate. This event at time 10 can only be consumed later by a second evaluation. In the eventdriven algorithm, only one evaluation would have been necessary. These extra evaluations combined with the longer evaluation tunes and the overhead for deadlock resolution cause the Chandy-Misra to execute 2-6 times slower on a uniprocessor. This overhead must be reduced in order to make the Chandy-Misra algorithm suitable for use with small-scale parallel systems. We are currently looking at several methods to reduce each of three types of overheads including better activation criteria, exploiting more domain specific knowledge about element behavior, and caching schemes in deadlock resolution.

6.3 Maximum Parallelism Available From This Approach

Since the Chandy-Misra algorithm executes 2-6 times slower than the event-driven algorithm on a uniprocessor, the Chandy-Misra algorithm had better uncover significantly more concurrency in order to be effective. Earlier, we showed that the Chandy-Misra algorithm achieved an average concurrency of 68 for the four benchmark circuits. To see just how much this concurrency can be increased, we performed simulation runs that computed the maximum amount of exploitable concurrency constrained only by the distributed-time method itself, regardless of the particular choices of activation cirteria, deadlock resolution scheme, or parallel architecture used. The

Table 7: Overheads in the Chandy-Misra Algorithm

	Element Evaluations		Grain Size (Simulated Cycles)		% Time In
	C-M	Event-Driven	C-M	Event-Driven	Deadlock Resolution
Ardent-1	346,552	329,430	803	615	58
H-FRISC	99,930	60,604	564	276	46
Mult-16	58,554	41,784	651	277	41
8080	13,294	11,364	2130	1309	19

maximum concurrency that can be exploited by an event-based simulator depends on the dynamic dependencies between the unprocessed events in the circuit, the topology of the circuit, and the behavior of the circuit. The dependencies change as events are processed and new events are produced. Since the Chandy-Misra algorithm only activates elements when an event arrives on one of its inputs and does not always propagate time updates, it does not extract the maximal concurrency; in order to find out what this maximum concurrency is, each element evaluation that updates the valid-time of a wire, must propagate the effects of that update forward through the circuit (this is similar to having every element send a NULL message after every evaluation as described in Section 2). The simulation execution times are very long with this "maximum propagation** turned on, but the results are instructive. The average Q lengths obtained from these simulations show the maximum available concurrency constrained only by the dynamic interaction of the events in the system and the topology of the circuit. Table 8 presents data showing how the concurrency and uni-processor run times are affected by this "maximum propagation". The average Q length in the "Max" runs represent the maximal concurrency. The "Max+sens" runs show how knowledge of the element behavior (sensitization) dramatically reduces the time taken for deadlock avoidance (though still high when compared with deadlock detection) while still achieving high parallelism. Note that the Q lengths for the "Max+sens" runs can be greater than the "Max" runs, as they are for the Ardent and 8080 circuits. The reason that sensitization can extract more than the "maximal" concurrency is that incorporating element behavior into the simulation changes the behavior of the simulation.

Table 8 shows that for the Ardent-l, H-FRISC, and 8080 circuits, the generic Chandy-Misra algorithm extracts about 40-50% of the maximum concurrency as constrained by the dynamic event distribution and topology of the circuit For the Mult-16 circuit, the Chandy-Misra algorithm extracts only 15% of the maximum concurrency. This is much lower than the simulations of the other three circuits due to the high incidence of deadlock. Since there are 20 deadlocks per clock cycle in the multiplier simulations and the evaluation queue length must dwindle down to zero before each deadlock is detected, the average queue length is cut down to 15% of maximum. Thus, the generic Chandy-Misra algorithm exploits an average concurrency of 68. This concurrency is 34% of the maximal average concurrency of 198. However, if we change the simulation by introducing knowledge of the behavior of some of the elements, the maximal average concurrency is increased to 249 and the concurrency exploited by the Chandy-Misra is increased to 93 or 37% of maximal. In either case, the Chandy-Misra algorithm effectively exploits a good portion of the actual concurrency present in the logic simulations.

6.4 Exploiting Element Behavior

In general, an element can safely advance its local time only when *all* of its inputs are ready. However, for some types of elements it is known that the outputs will be stable for a certain period of time regardless of some of the other inputs. For example, for some simple registers, it is known the output will not change until the

Table 8: Maximum Parallelism (Always Update Times)

Circuit	Deadlocks	Total Execution Time	Avg. Q Length
Ardent- 1	1749	171.2	108
Ardent-l Max	0	13.415-o	235
Ardent-l Max+sens	0	1663.3	527
H-FRISC	384	23.6	111
H-FRISC Max	0	600.5	223
H-FRISC Max+sens	0	41.8	158
Mult-16	273	16.6	45
Mult-16 Max	0	9.7	307
Mult-16 Max+sens	0	9.7	264
8080	878	7.1	10.0
8080 Max	0	220.7	26.0
8080 Max+sens	0	30.5	45.0

next clock pulse arrives. Thus, if the register is clocked at time 100 and it already has the next clock pulse at time 200 in its input queue, it knows its output will not change until time 199. If we did not know anything about the behavior of the element, the output could only be determined up to time 100. This technique of taking advantage of register behavior is called register sensitization. As another example, consider an AND gate with a 0 on one of its inputs. The output of the gate will not change until this 0 changes. Using this knowledge is called combinational sensitization.. Exploiting these types of *domain specific information can* advance the local times of these elements further in the simulation, possibly avoiding some deadlocks. (For details about how this sensitization idea can be extended to any functional element as well as implementation details, see Appendix B).

Table 9 shows how often it was possible to exploit the sensitization information and how often that information helped. In order for sensitization to count as "helping", the element must be able to advance its local time further than it otherwise could have without using the sensitization information., For example, if an AND-gate receives an event at time 10 with a value of 0, but the node is only known up until time 10, the knowledge of its behavior won't help. If, however, the node was known to be 0 up until time 20, the sensitization information would help. The column showing how often register sensitization helped reflects how often a register knew when its *next* clock coming. The help percentages are fairly high in the three circuits with registers. In the 8080, register sensitization helps 98% of the time. This high number is due to all of the system clocks coming *directly* from the generators. The circuit with the lowest help percentage is the H-FRISC at 34%. This is caused by a small central controller in the H-FRISC that sends out start signals and clocks to the main sections, which, in turn send back done signals when appropriate. This feedback causes the clocks to be generated more or less one at a time for the main portions of the circuit and lowers the help ratio. However, as Table 10 shows, sensitization reduces execution times by an average of 30% (and 80% in the multiplier) and is definitely a worthwhile optimization.

Table 11 shows how the composition of a deadlock changed when the sensitization information was used (again, please note that each deadlock activation may be categorized in more than one type). The number of deadlocks is reduced 13% in the Ardent case, 14% for the H-FRISC, 100% for the multiplier, and 11% for the 8080.

Figure 1 shows the event profiles for the standard Chandy-Misra algorithm while Figure 7 shows the event

Table 9: How Often Can Element Behavior Be Exploited

Circuit	Total Evals	Times Register Sens. Possible	% Times Helped	Times Combinational Sens. Possible	% Times Helped
Ardent-1	336,679	264,021	63	18,306	18
H-FRISC	86,632	18,759	34	57,122	35
Mult-16	10,926	0	0	8,801	37
8080	11,655	5,912	98	1,310	44

Table 10: Effects of Sensitization on Execution Times

	Element Evaluations		Grain Size		Execution Time	
	C-M	C-M+sens	C-M	C-M+sens	C-M	C-M+sens
Ardent-1	346,552	228,648	803	801	1,372	1,090
H-FRISC	99,930	87,239	564	596	172	157
Mult-16	58,554	10,926	651	1,693	114	23
8080	13,294	11,655	2,130	2,326	48	46

Table 11: Deadlock Activations by Type With Sensitization

Circuit	Deadlocks	Reduction (%)	Total Deadlock Activations	Register Clock Activations	Generator Activations	Order of Evaluation	One-level NULL
Ardent-1	1808		308k	281k	836	1.6k	4.1k
Ardent-1+both	1576	13	297k	276k	629	3.0k	7.5k
H-FRISC	404		45.6k	8.9k	8.9k	1.2k	4.3k
H-FRISC+both	348	14	32.2k	8.3k	8.3k	1.0k	8.5k
Mult-16	273		23.4k	0	40	1.6k	1.0k
Mult-16+both	0	100	0	0	0	0	0
8080	878		8.3k	4.6k	4.6k	236	502
8080+both	780	11	6.6k	4.4k	4.5k	304	799

profiles for the Chandy-Misra algorithm using sensitization information (the multiplier plot is not included since there were no deadlocks - a plot of the parallelism and activations per deadlock would be uninteresting). There is not too much difference, again excluding the multiplier circuit, in the appearance of the two sets of profiles.

Figure 8 shows the relative parallel performance for the Chandy-Misra algorithm with and without sensitization information.

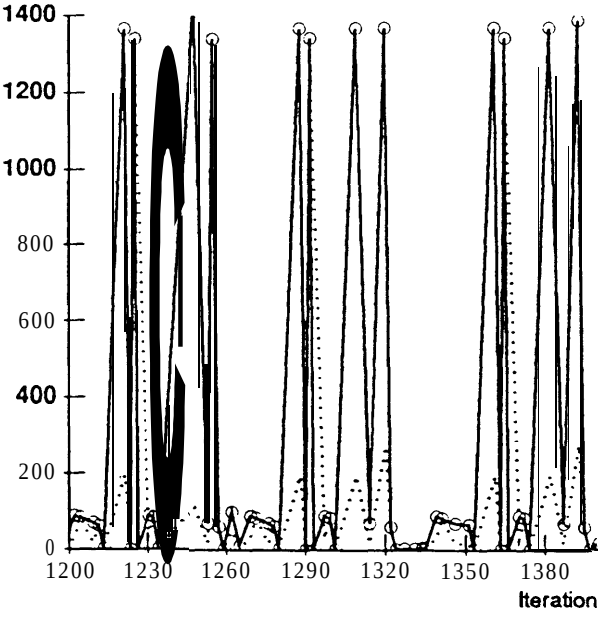
6.5 One Level Checking

The last section described how to advance an elements local time by exploiting some aspects of its behavior. A different approach to advance an elements local time is to look at the gates driving the elements inputs in addition to the valid-times of the inputs. This should be helpful because a node time update alone without an new event does not activate the fan-out element for that node. This lack of activations makes the Chandy-Misra algorithm efficient but introduces many unevaluated paths. These unevaluated paths, described in Section 5.4, account for 40-98% of all deadlock activations in the simulations of the four circuits. Each deadlock activation caused by an unevaluted path is characterized by how many *levels* of circuit elements would have to be evaluated in order to break the deadlock. The activations caused by the shortest possible path are classified as one level NULLs. These one level NULL activations are the largest class of unevaluated paths, accounting for 30-80% of all deadlock activations. To avoid the one-level NULL class of activations, we proposed a "one-level checking" scheme. One level checking is a method of marking the elements activated as one level NULLs during one simulation, and using that information during the next run. When these marked elements are evaluated, they check back one level – that is, if any input of their inputs are lagging behind in time preventing an input event from being consumed, the element driving that input (the fan-in element) is checked and its output time is updated if possible (some of its input-times may have been updated since its last evaluation). Once all the necessary fan-m elements have had their times updated, the current element is checked again which may allow it to consume the event it otherwise would have had to wait on. The hope is that if we pick these elements carefully, we can avoid enough deadlocks, hence lowering global synchronizations and execution time, that it will be worth the cost of checking back one level on all the marked elements.

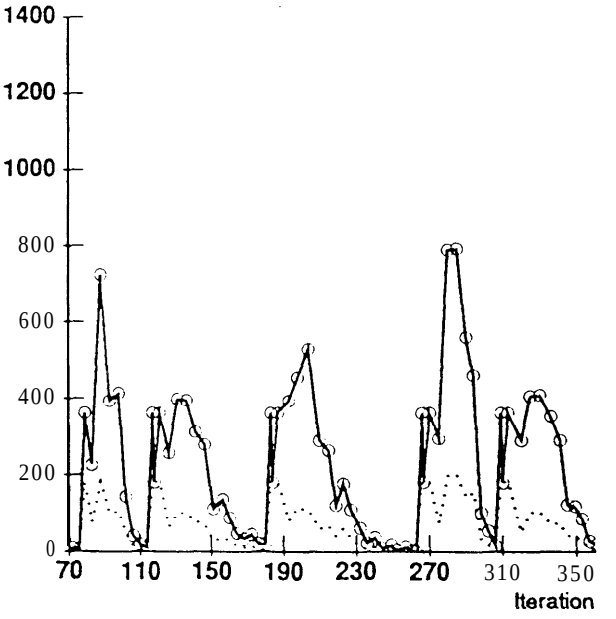
Table 12 shows how one-level checking reduces the number of deadlocks, deadlock activations, and one-level NULL activations. The table also shows what happens when desensitization is combined with one-level checking. In the following paragraphs, the results for each circuit will be discussed in more detail.

For the Ardent, one-level checking reduces the number of one-level NULL activations by 10% but only reduces the number of deadlocks and deadlock activations by 2%. The reduction of only 10% in the one-level NULL, activations is due to the low activity on the register inputs. Since most of the clock lines are generated during the start up of the simulation, the registers are only activated when an input event occurs on one of its inputs. Elements that are not activated can not perform the one-level check and must wait for the deadlock resolution phase to activate them. These registers account for over 90% of the one-level activations. Since the activity levels on the register inputs is so low, the registers rarely get a chance to do the one-level checking. When one-level checking is combined with desensitization, the total reduction of the one-level NULL activations is 62% with the number of deadlocks reduced by 19% and deadlock activations reduced by 41%.

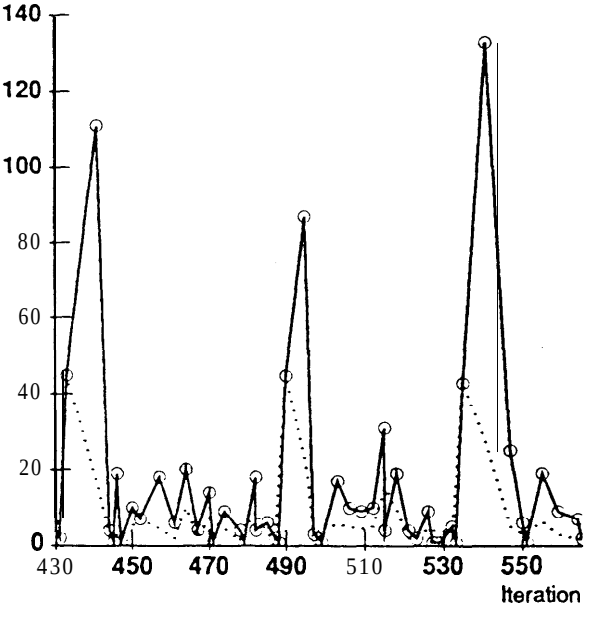
For the H-FRISC, one-level checking reduces the number of one-level NULL activations by 68%. This reduces the number of deadlocks by 20% and the number of deadlock activations by 45%. Unlike the Ardent circuit, most of the elements marked for checking do get activated during the normal simulation thus allowing them to perform the one-level check. When one-level checking is combined with desensitization, the number of one-level NULL activations is reduced by 78% with the number of deadlocks reduced by 38% and deadlock activations reduced by 72%.



Ardent Queue Length Profile



H-FRISC Queue Length Profile



8080 Queue Length Profile

Figure 7: Event Profiles With Sensitization

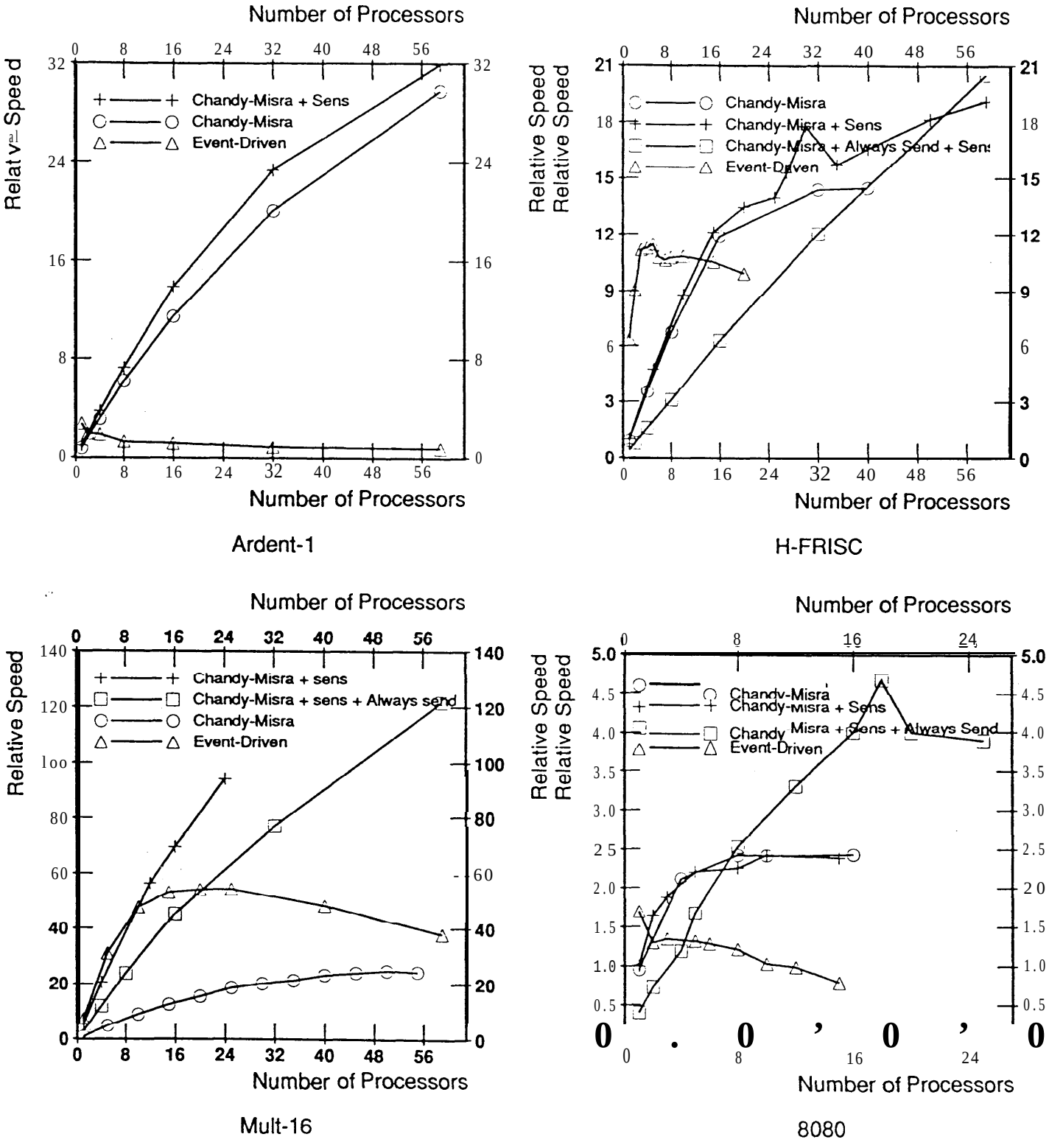


Figure 8: Speed-Ups With Sensitization

Table 12: Deadlock Activations With One Level Checking - Part 1

Circuit	Deadlocks	% Redct.	Deadlock Actv.	% Redct	One Level NULL	% Redct	Execution Time
Ardent-1	1749		306.0k		80.3k		1372
Ardent- 1 +Chk	1710	2.2	299.3k	2.2	72.0k	10.3	1539
Ardent- I +Sens	1512	13.6	197.8k	35.3	45.4k		1090
Ardent-l+Chk+Sens	1406	19.1	181.8k	40.6	30.8k	61.6	
H-FRISC	384		45.8k		35.7k		I72
H-FRISC+Chk	307	20.0	25.1k	45.2	115k	67.8	168
H-FRISC+Sens	327	14.8	32.3k	29.5	23.2k		157
H-FRISC+Chlc+Sens	236	38.5	13.6k	70.3	7.7k	78.4	137
Mult-16	273		27.2k		21.6k		115
Mult-16+Chk	269	1.5	10.4k	61.8	6.8k	68.5	126
Mult-16+Sens	0	100.0	0	100.0	0		23
Mult-16+Chk+Sens	0	100.0	0	100.0	0		23
8080	878		8.3k		2.6k		46
8080+Chk	689	21.5	6.8k	18.1	0.8k	69.2	157
8080+Sens	775	11.7	6.3k	24.1	1.6k		48
8080+Chk+Sens	548	37.6	5.4k	34.9	0.4k	84.6	

For the Mult-16 circuit, one-level checking reduces the number of deadlocks by 2%, the number of deadlock activations by 62%, and the number of one-level NULL activations by 68%. The reason a 62% reduction in deadlock activations only removes 2% of the deadlocks is that the reduction is more or less uniform across all the deadlocks and rarely removes *all* of the activations. (Note that desensitization removes all of the deadlocks so this is just for measurements sake).

For the 8080, one-level checking reduces the number of one-level NULL activations by 69%, the number of deadlocks by 22% and the number of deadlock activations by 18%. One-level checking combined with desensitization, reduces the number of one-level NULL activations by 85%, deadlocks by 38% and deadlock activations by 35%.

As shown by Table 12, one-level checking combined with sensitization eliminates 2040% of all deadlocks while reducing uniprocessor execution times as well. This combination is an effective way to increase the parallel performance of the Chandy-Misra algorithm.

6.6 Global Event Counting

The characterization of deadlocks and the optimizations discussed so far point to one basic question that each element must answer in a conservative simulation: "Can I proceed to time *t*?" Normally, this is answered by having each element check the valid times of its inputs. In the event-driven algorithm, this question is answered by checking the global event queue to see if all events before time *t* have been processed. This technique used in the event-driven algorithm can be incorporated into the Chandy-Misra algorithm by maintaining counts of

Table 13: Deadlock Activations With Cycle Counting

Circuit	Deadlocks	% Reduction	Deadlock Activations	% Reduction	Avg Queue Length
Ardent-1	1749		306.0k		108
Ardent-1+Count	1350	22.9	305.4k	0.2	122
H-FRISC	384		45.8k		111
H-FRISC+Count	335	12.8	36.7k	19.9	104
Mult-16	273		23.4k		45
Mult-16+Count	272	0.4	23.4k	0	45
8080	878		8.3k		10.0
8080+Count	733	16.5	4.4k	47.0	9.3

the unprocessed events in the system with event-times below some time t. Unfortunately, keeping counts of the events left at each time-step would cause serious bottlenecks at the counter locks. To avoid these bottlenecks, we decided to keep a counter of the events left in each clock cycle. When the number of events reaches 0 for this clock cycle, the valid times for all nodes can be implicitly updated to the beginning of the next clock cycle. This allows elements to start consuming events in the next clock cycle thus eliminating one deadlock per clock cycle. Table 13 presents a performance summary of this technique. Since it removes one deadlock per clock cycle, the reduction in the number of deadlocks depends only on the number of deadlocks per cycle.

6.7 Deadlock Avoidance by Always Sending NULLs

As pointed out in Section 6.3, one way to completely avoid deadlocks is to always send out a NULL message after every element evaluation. Unfortunately, when used in digital logic simulations, this causes a flood of NULL messages and simulation times increase 10 or 100-fold. However, if we use sensitization information in conjunction with always sending NULLs, the simulation times become reasonable and the exploited concurrency increases about 10-fold. The speed-up curves in Figure 8 show some very preliminary results for the Chandy-Misra algorithm using this "Always Send+Sens" technique. For the H-FRISC and 8080 circuits, always sending a NULL message achieves the best overall performance on a 60-processor machine. However, the Ardent simulation using this technique did not finish after 8 hours of simulation (typical runs were around 10-15 minutes) and for the circuits it did work on, large numbers of processors were needed to outperform the other variants of the Chandy-Misra algorithm. These results are prehminary but we feel the limited success warrants further study into this optimization.

7 Conclusions

In characterizing the parallelism in distributed-time simulations of real circuits, we have shown that the Chandy-Misra algorithm extracts an average parallelism of 68 for the four benchmark circuits used. While this is 1.5 times better than traditional parallel event-driven algorithms, it is still too low to be used effectively in large parallel processing systems. Since deadlocks are the major factor limiting parallelism and the overall performance, the paper focused on understanding the nature of deadlocks. We classify the deadlocks that occur in logic simulation into four types: register clocks and generator nodes, multiple paths, unevaluated paths and the order of node

Table 14: Best Performance on a 60 Processor Machine (Times for each circuit are normalized to the Chandy-Misra performance on 1 processor)

Circuit	Event-Driven	Chandy-Misra	Chandy-Misra With Optimizations	Ratio of Best CM to EV
Ardent-1	3.5	375	40.2	11.4
H-FRISC	12.6	15.9	22.5	1.8
Mult-16	54.3	24.4	121.8	2.2
8080	1.7	2.6	4.9	2.9

updates. These four types are able to cover almost all of the deadlocks that occur. Concentrating on each type, we presented specific solutions to avoid or resolve the deadlocks caused by that type. These solutions exploit several different aspects of circuit behavior and the preliminary results are very promising. Table 14 shows the best performance achieved by the event-driven, the generic Chandy-Misra algorithm, and the Chandy-Misra algorithm with domain specific optimizations. In all four benchmarks, the Chandy-Misra algorithm with optimizations is able to out perform the event-driven algorithm on a 60 processor machine by a factor of 2 to 11 with the best results being for the Ardent-l simulation. Even though there is more overhead in the Chandy-Misra algorithm – a factor of 2 to 6 in the four circuits simulated – the cross-over point with the eventdriven performance always occurs between 3 to 12 processors. Also in all four circuits, the event-driven algorithm performance peaked before reaching 20 processors while the Chandy-Misra algorithm could effectively use more than 60 processors. These results make the Chandy-Misra algorithm an effective alternative for logic simulation on small to large scale multiprocessors.

8 Acknowledgments

The authors would like to thank Bruce Delagi and Tom Blank for their contributions to this paper. The authors are supported by DARPA contract N00014-87-K-0828. In addition, Anoop Gupta is supported by a DEC faculty award, and Larry Soule is supported by DEC, and was previously supported by a National Science Foundation Graduate Fellowship. We would also like to thank Mark Horowitz and Glen Miranker for all their help in getting the Ardent circuit

References

[1] M. Bailey and L. Snyder. An Empirical Study of On-Chip Parallelism. In *25th Design Automation Conference,* pages 160–165. University of Washington, June 1988.

[2] Tom Blank. A Survey of Hardware Accelerators Used in Computer-Aided Design. *IEEE Transactions on Design and Test,* pages 21-39, August 1984.

[3] R. E. Bryant Simulation of packet communication architecture computer systems. Technical Report MIT,LCS,TR-188, MIT, July 1977.

[4] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. Comm *of the ACM,* 24(11):198–206, April 1981.

[5] Helen Davis and Steven Goldschmidt. Tango: A multiprocessor simulation and tracing system. March 1989.

[6] Monty Denneau. The Yorktown Simulation Engine. *In 19th Design Automation Conference,* pages 55-59. ACM/IEEE, 1982.

[7] Tom Diede, Carl Hagenmaier, Glen Miranker, Johnathan Rubinstein, and William Worley. The Titan Graphics Supercomputer Architecture. *Computer,* 21(9):13–30, September 1988.

[8] David Ku and Giovanni DeMicheli. *HERCULES* - A *System* for High-Level Synthesis. *In 25th Design Automation Conference,* pages 483-488. ACM/IEEE, June 1988.

[9] J. Smith, K. Smith, and R. Smith. Faster Architectural Simulation Through Parallelism. In *24th Design Automarion Conference,* pages 189-194. ACM/IEEE, June 1987.

[10] Larry Soule and Tom Blank. Statistics for Parallelism and Abstraction Level in Digital Simulation. In *Proceedings of the 24th Design Automation Conference,* pages 588-591. Stanford University, 1987.

[11] Larry Soule and Tom Blank. Parallel *Logic* Simulation on General Purpose Machines. In *Proceedings of the 25th Design Automation Conference,* pages 166–171. Stanford University, 1988.

[12] Larry Soule and Anoop Gupta. Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation. In *Proceedings of the 26th Design Automation Conference,* page Session 7.1. Stanford University, 1989.

[13] Steven Swope and Richard Fujimoto. Optimal performance of distributed simulation programs. Technical Report UUCS-87-023, University of Utah, Department of Computer Science, 1987.

[14] K. F. Wong, M. A. Franklin, R. D. Chamberlain, and B. L. Shing. Statistics on Logic Simulation. In *23rd Design Automation Conference,* pages 13-19. ACM/IEEE, July 1986.

APPENDICES

A Implementing Inertial Delays

Most commercial event-driven simulators support a more detailed delay model than the one presented here. This appendix shows how this approach can be extended to support inertial delays and spike analysis. Figure 9 show the basic mechanism for implementing inertial delays. The code for the new component, cm-inv looks like:

```
if (Check_cancel_input has not changed from the last time we were called){
        /* We just have an event on the normal input */
    if (Test == 1){    /* We can always pass falling output transitions */
        Test b = 0;
        last_fall = current_time;
    } else {
        Toggle Check_cancel-output;
    }
} else {
       /* Check to see if we should cancel the rising transition on the output
or not */
        if (last fall - current-time > rise-delay-fall-delay)
            Test_b = 1;    /* It's ok to send out the rising transition */
        else
           /* Do nothing since we don't want to send out the rising transition
*/
```

Since either rising or falling transitions (whichever has the shortest delay) can always be passed through, they cause no degradation in performance. The transition type with the longer delay will cause the element to be evaluated twice (once on receiving the input event and once when checking for cancellation). Thus for every cycle ofrisingandfalling transitions we will have three evaluations whereas we would have two inertial delays were not implemented, yielding a potential performance degradation of 50%.

B Generalized Sensitization

The previous two sections described how to "desensitize" logic gates and synchronous registers. Since we have a very flexible functional simulator that can use any type of functional element, we want to generalize this concept of sensitization to work for *all* functional elements. For example, if a two-input MUX knows the value of its select line, it will be able to ignore one of the inputs which may allow it to advance its local time more that if it were treated as a generic element. This generalized desensitization is represented just like the functionality is represented. For the MUX example, there is a routine which determines the values of the outputs based on the inputs just like any functional simulator. There is also a routine which determines the valid times of the outputs based on the values and valid-times of the inputs. Code for the MUX implementation looks like:

```
/* Standard functional code */
MUX(inputs, select_line, output){
    if (select_line == 0)
        output = inputs[0];
    else if (select_line == 1)
        output = inputs[1];
    else
        output = Undefined;     /* select_1ine is undefined or floating */


/* Timing code */
/* "signal"_time is the maximum valid time of the */
/* input or the time of the next event on "signal" */
MUX-time(inputs, inputs_time, select_line, select_line_time, output_time) {

/* If we know select line is a 0 or a 1, the output is valid until */
/* the selected input changes or the select-line changes.          */
    if (select_line == 0)
        output_time = Minimum(inputs_time[0], select_line_time);
    else if (select_line == 1)
        output_time = Minimum(inputs_time[1], select_line_time);

    else
/* The output is guaranteed not to change until */
/* something happens on the select_line         */
        output_time = select_line_time;
```

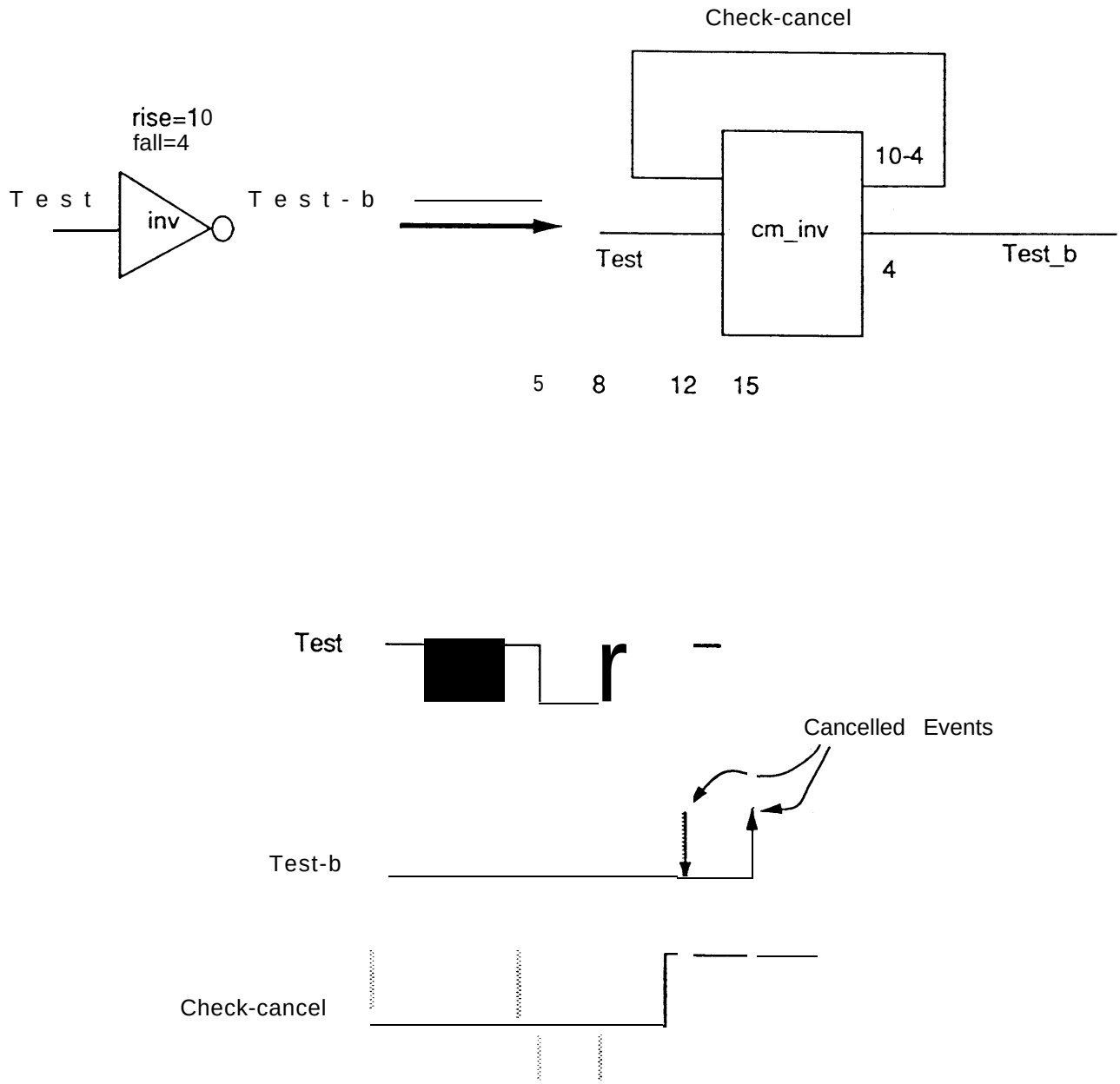


Figure 9: Implementing Inertial Delays in a Distributed-Tie Environment