Parallel Timing Simulation on a Distributed Memory Multiprocessor

Chih-Po Wen
Computer Science Division
University of California, Berkeley

January 7, 1993

Abstract

Circuit simulation is one of the most computationally expensive tasks in circuit design and optimization. Detailed simulation at the level of precision of SPICE is usually performed on critical circuit components only. In this paper we present a parallel timing simulator on a distributed memory multiprocessor, as an attempt to increase the speed and scale of circuit simulation for digital MOS circuits. The parallel implementation is based on the event-driven timing simulator SWEC, which outperforms SPICE by one to two orders of magnitude. Our approach to parallelization exploits runtime parallelism by scheduling the events *optimistically.* Trace-driven analysis shows that the optimistic simulation method exploits more parallelism than the conservative method for circuits with feedback signal paths. We describe the design tradeoffs in the implementation and report on its performance for several benchmark circuits. Speedups over SWEC on large realistic circuits are as high as 60 on a 128-node CM5 multiprocessor. The total speedup over SPICE can be as high as 1000. These results indicate the feasibility of using distributed memory multiprocessors to perform large-scale circuit simulation.

Contents

1 Introduction

Circuit simulation is one of the most important tasks in circuit design and optimization. It is also one of the most computationally expensive. Although programs such as SPICE have been used on a wide range of circuits to provide accurate simulation results, they are usually performed on critical circuit components only, due to the limited processing and memory capacity of conventional computers. Detailed simulation of an entire processor design at the level of precision of SPICE is virtually impossible. As VLSI technology advances and circuits become more complex, circuit simulation is likely to become the bottleneck of the design process, unless simulation speed can be scaled with increases in circuit size.

Circuit simulators such as SPICE employ *direct methods* to solve a large, sparse system of differential equations. Less expensive simulation techniques have been proposed for the class of digital MOS circuits. These techniques are based on the partitioning of a large circuit into many smaller, *loosely coupled* subcircuits. Direct methods can then be applied independently to the individual subcircuits within one time step, which requires far less computation than solving the entire circuit at once. In relaxation-based methods, this process may have to be iterated many times until the result converges. In timing simulation methods, each time step is processed only once by carefully choosing the time step size and the order of evaluation. Both the relaxation based simulators and the timing simulators show significant speed improvement over SPICE [5], at the cost of slightly lower precision.

In this paper we address the parallelization of timing simulators on distributed memory multiprocessors. Previous work [1, 2, 3] on parallel timing simulation has concentrated on evaluating multiple subcircuits in parallel within one time step. One approach is to execute subcircuit tasks in a loosely synchronous manner with synchronization between each time step, if all the subcircuits use a uniform time step (e.g., the "synchronous" algorithm in CEMU [2]). A second approach is to execute subcircuit tasks in a data-driven manner (e.g., the "asynchronous" algorithm in CEMU [2]). The data-driven approach exploits more parallelism, but incurs higher communication overhead [1]. Low-level parallelism is also available within the evaluation of each subcircuit. Techniques for exploiting the low-level parallelism are essentially those for parallelizing the direct methods [4].

Our parallel timing simulator is based on the parallelization of SWEC [5], a sequential, event-driven timing simulator. SWEC employs a stepwise linear waveform and device model, and thus the task of evaluating a subcircuit is reduced to solving a linear system of node voltages. Furthermore, the evaluation of a subcircuit is triggered by *events* generated by the state changes of the subcircuit and its fanins. The rate of state change determines the time step size to be used for a given subcircuit at a particular time point. These features take advantage of the *latency* and *multirate* properties in most digital circuits [6]. The latency property states that most digital signals change infrequently; the multirate property states that different parts of a circuit produce signals at different speeds.

Parallelizing SWEC is a challenging problem of parallel programming, particularly on distributed memory multiprocessors. The event-driven approach with variable time steps makes the communication between subcircuits unpredictable, and thus precludes the possibility of a synchronous, low-overhead implementation [2]. Empirical evidence from SWEC also indicates that the task granularity is usually small (hundreds of microseconds on a CM5), because the stepwise linear model avoids the costly Newton-Raphson iterations used for solving non-linear systems. Therefore, we must be careful not to introduce excessive parallelization overhead that would overwhelm the cost of computation.

Our approach to parallelizing SWEC is as follows. First, we use static data layout to distribute work among the processors; each processor is responsible only for simulating the subset of subcircuits stored in its local memory. This static partitioning strategy avoids the communication overhead of making global scheduling decisions, and of relocating subcircuit states for load balancing. These overheads can be significant when compared with the actual computation cost. Second, we employ optimistic simulation in scheduling the subcircuit evaluations. Optimistic scheduling exploits the parallelism determined by the actual signal flows at runtime, rather than limiting parallelism to the static interconnection of the circuit. A previous study [7] shows the lack of parallelism at the switch level for several benchmark CMOS circuits. Therefore, we choose to trade the memory and computation overhead of optimistic simulation for the increase in exploitable parallelism, in order to fully utilize the power of a large scale multiprocessor.

We use the CM5, a distributed memory multiprocessor, as the target machine for parallelization. Our configuration of the CM5 contains up to 128 33-Mhz SPARC/2 processors, interconnected by a fat-tree network with 5 to 20 MBytes/sec bandwidth between pairs of processors. Each processing node has 8 MBytes of local memory, totaling 1 GByte of memory in the system. Besides the message passing facilities, the CM5 also provides fast synchronization and global combining primitives. Our implementation takes advantage of these special hardware features whenever possible.

The rest of this paper is organized as follows. Section 2 introduces the timing simulation problem and outlines the sequential algorithm. Section 3 reports the measurements on the parallelism available in the benchmark circuits. These measurements are used to justify the use of multiprocessors and the optimistic approach of simulation. Section 4 describes our data layout and load balancing policy. Section 5 introduces approaches to discrete event simulation. Section 6 describes our optimistic simulation algorithm, and outlines its implementation on the CM5. Section 7 reports on the performance of our parallel timing simulator and gives analysis of the costs. Section 8 compares our approach to previous work on parallel timing simulation. Finally, we give our conclusions and suggest future work in Section 9.

2 Problem Statement

This section introduces the timing simulation problem, and in particular, the SWEC approach to timing simulation. The introduction describes only the parts of circuit simulation that are relevant to our parallelization problem.

2.1 Introduction to Timing Simulation

Timing simulation is a special form of circuit simulation that is optimized for digital MOS circuits. A timing simulator performs time-domain transient analysis, which gives the target circuit's state in response to the input signals. The resolution of the analysis is roughly the same as that of a general-purpose circuit simulator; it gives the circuit's response in terms of voltages on the real time axis. The precision of simulation is slightly lower, but the speed improvement is significant.

The first step in a timing simulation is to partition the circuit based on the charge coupling of transistors. In digital MOS circuits, two transistors are tightly coupled if they are connected via a source-drain charge path. On the other hand, they are loosely coupled if they are connected only via the gate terminal of some transistor, implying that the signal flow is unidirectional. Tightly coupled transistors must be evaluated simultaneously for precision, while loosely coupled transistors can be evaluated mostly independently with infrequent propagation of values between transistors. We refer to the clusters of tightly coupled transistors as *subcircuits*.

The evaluation of a subcircuit uses the same techniques as those for general purpose circuit simulators. A circuit is represented as a set of devices such as resistors or transistors interconnected at the *nodes*. The device behavior and the fundamental voltage and current laws (KVL and KCL) determine the relations between the node voltages and the branch currents. These relations are usually in the forms of first order ordinary differential equations for circuits containing non-linear devices and capacitance. The simulation then proceeds as follows:

- Apply a numerical integration step to convert the differential equations into a system of nonlinear algebraic equations.

- Apply Newton-Raphson iterations to convert the nonlinear algebraic system into a sequence of linear algebraic systems.
- Solve the linear system at each iteration (possibly using sparse methods for large systems).

These steps are repeated for each time step to obtain the entire time-domain transient response. Notice that the values computed at each time step are used by the subsequent time steps.

2.2 The SWEC Approach to Timing Simulation

SWEC further optimizes the timing simulation algorithm described in Section 2.1. In SWEC all the voltage waveforms are considered to be piecewise linear. The piecewise linear waveform model provides an efficient yet accurate encoding of most digital signals. By prudently controlling the time steps, the input and output voltage waveform can be approximated by straight line segments, and the devices can be treated as constant conductance devices. (Hence the name SWEC, which stands for "Step-Wise Equivalent Conductance".) These approximations result in a system of *linear* algebraic equations after the numerical integration step, and thus the Newton-Raphson iterations in step 2 of the direct method described above is no longer necessary.

In addition to reducing the cost of each evaluation, the piecewise linear waveform model also reduces the number of evaluations required. First, the time step size used by a subcircuit at a particular time depends on its activity; arbitrarily long time steps can be used for a subcircuit as long as its input and output waveforms for the time step are linear within a small error margin. Second, a subcircuit propagates its output only when the new output cannot be linearly extrapolated using the old one; several time steps may elapse before a subcircuit communicates its state. The act of communicating the new state to fanout subcircuits is called an *event*.

In summary, SWEC reduces the cost of each subcircuit evaluation by its stepwise equivalent conductance device model. It also reduces the number of evaluations by using variable time step control and the event-driven approach. Combining these techniques, SWEC is shown to outperform SPICE by one to two orders of magnitude on real world digital circuits [5]. The precision of SWEC is also shown to be very close to that of SPICE; the cycle time for a ring oscillator with 7 inverters estimated by SWEC is the same as that by SPICE.

2.3 The SWEC Simulation Algorithm

2.3.1 Terminology

We now introduce the terminology and algorithms in the SWEC timing simulator. Our focus will be on the control structure of the program; the reader is referred to the original paper on SWEC [5] for detailed descriptions of the numerical components in the simulator. However, they are not essential to the understanding of this presentation.

The basic data structure in SWEC is a *region*, which stores the data of a subcircuit. The subcircuits are determined by an automatic *partitioning* step, which is the first step in the simulation. A region contains the collection of MOS transistors and the nodes (or *nets*) they drive. We use $N(r)$ and $M(r)$ to denote the number of nodes and the number of MOS transistors in region r, respectively.

For the presentation of this paper, it suffices to think of the circuit as a directed multi-graph (referred to as G), where a vertex denotes a region, and an edge $e1$ from vertex r to vertex r' denotes a unidirectional signal path from some node in r to the gate terminal of some MOS transistor in r'. Note that there can be more than one edge between a pair of vertices. We define $In(r)$ as the set of vertices (regions) incident to r, and $Out(r)$ as the set of vertices (regions) incident from r. The number of edges emergent from r is denoted by $deg(r)$. Figure 1 gives an example of the data structure.

The state of a node n at time t is denoted $n(t)$; it consists of its voltage $n(t).v$ and its voltage derivative $n(t).dv$. The state of a MOS transistor m at time t, $m(t)$, is its conductance $m(t).g$, which is a function of the state of its driving node. The state of a region r at time t, $r(t)$, is simply the set of the states of the nodes and the MOS transistors in r. Note that the states are defined (evaluated) only at discrete time points in the simulation. For the rest of this paper, the universal quantifiers on times refer to these discrete time points only.

The voltage sources are modeled as regions whose states are completely defined by the input waveforms.

2.3.2 Time Step Control in SWEC

The state of a region is computed at sufficiently many time points to yield piece-wise linear waveforms of the node voltages. We refer to the computation of a region's state over one time step as an *evaluation*. An evaluation of a region may produce a new state value that has to be conveyed to its fanout regions to ensure accuracy. Specifically, a state value of a region is considered to be *new* if it cannot be approximated by a linear extrapolation on the old value. More precisely, $r(t)$ is new with respect to its preceding state $r(t')$ if for any

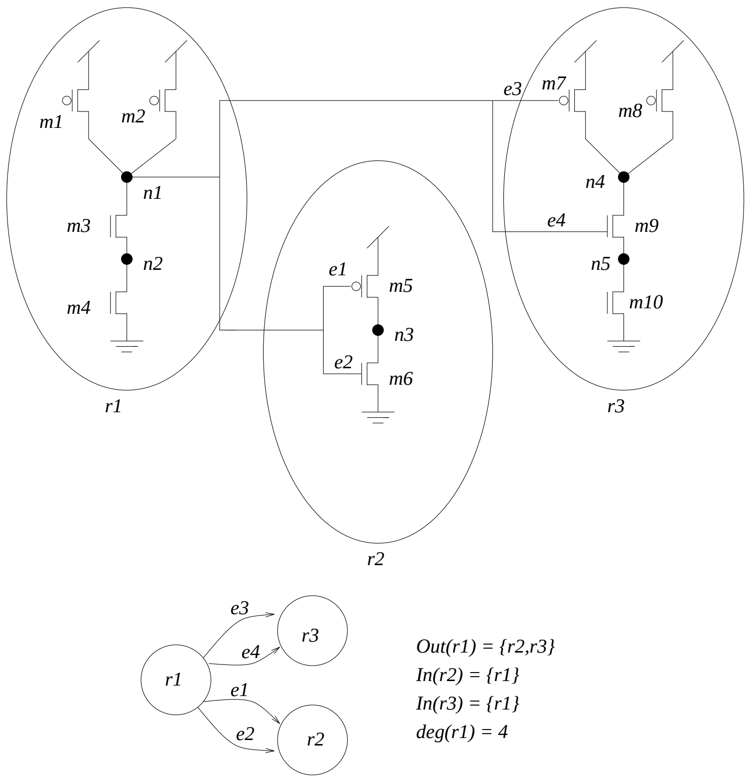

Figure 1: An example MOS circuit. The bottom diagram is the abstract view of the circuit in terms of regions.

node n in region r,

$$\left|1 - \frac{n(t).dv}{n(t').dv}\right| > \epsilon$$

where ϵ is a user-specified parameter to control the precision of the simulation. An evaluation that leads to a new state value is called an *event*. All events must be conveyed to the fanout regions (via the proper edges in G) to ensure correctness of simulation.

The resolution of the waveforms, or the number of time points in its evaluation, is controlled carefully to ensure the accuracy of the trapezoidal numerical integration and the stepwise equivalent conductance device model. Specifically, the next evaluation time for region r after time point t is determined by the minimum of:

- $t + h_{predict}(r(t))$: the next time point assuming no new fanin events.
- $\min_{\forall r' \in In(r')} \{t' + h_{adjust}(r'(t'))\}$, for all $t' \geq t$ such that $r'(t')$ is an event: the bound on time step in response to new fanin events.

where $h_{predict}$ *predicts* the next time step based on the current state, and h_{adjust} *adjusts* the next time step based on new events produced by r's fanin regions. All time values are discrete, and the basic time unit is one picosecond. Both $h_{predict}$ and h_{adjust} return at least 1 so that progress can be made. For convenience of discussion, we assume all time points are distinct[1].

To summarize, the simulation time step taken by a region varies with its activity. To advance the state of a region from time t to time t', we need to know (or to be able to extrapolate) the states of its fanin regions up to time t', in order to ensure the consistency of both control and computation. With this in mind, we proceed to present the SWEC simulation algorithm.

2.3.3 The SWEC Simulation Algorithm

In SWEC each region r maintains the variables *r.time* and *r.etime*, which store the time of the latest evaluation and the time of the next evaluation, respectively. The algorithm uses a priority queue Q (ordered by *r.etime*) to ensure that, when a region r is scheduled for evaluation at time *r.etime*, all states (and thus all events) before *r.etime* have been evaluated, and thus *r.etime* can be computed according to the definition of next evaluation time given above.

[1]Technically the time points can overlap, and the result will be dependent on the way the algorithm breaks ties. However, the choice has negligible effect on precision, and the original SWEC implementation breaks ties arbitrarily.

Before the simulation begins, $r.etime$ is initialized to ∞ for all regions except the voltages sources, for which the next evaluation time is set to be the first time point of the input waveform. $r.time$ is initialized to 0 for all regions. Every region r is put on the priority queue Q based on $r.etime$.

The algorithm then enters a scheduling loop containing the following steps:

- Remove the region r with the least $r.etime$ from Q.
- If $r.etime$ is greater than the total simulation time, exit.
- Extrapolate the states of the regions in $In(r)$ at $r.etime$ using the linear waveform model, based on the latest event produced by $In(r)$.
- Evaluate $r(r.etime)$, using $r(r.time)$, which is evaluated at the previous time step for r, and $In(r)(r.etime)$, which is extrapolated at the previous step. The evaluation is done in the following two steps:
  - The device models are evaluated to set up the system of linear equations of the node voltages. This is usually referred to as the *model evaluation* step.
  - A dense linear solver is applied to solve the system; the current SWEC implementation switches from a Gaussian elimination solver to a relaxation solver when the system is large.
- If $r(r.etime)$ is an event, adjust the time steps of all regions in $Out(r)$ using $h_{adjust}(r(r.etime))$. This is referred to as the *fanout* operation. Q is adjusted accordingly to reflect the new time order.
- Update $r.time$ to be $r.etime$. Predict the next time step using $h_{predict}(r(r.etime))$ and store the value in $r.etime$. Insert r to Q.

Note that if a region does not receive any signal (embodied in an event), it is never scheduled. It is easy to see that the algorithm is *safe*, meaning that all the states it generates are valid; it is also *live*, since the number of possible time points is finite and each iteration of the algorithm computes a new one.

3 Measurements of Available Parallelism

Previous attempts to parallelize SPICE-like circuit simulators have had limited success. For example, the parallel circuit simulators implemented by Yang [4] and Sadayappan [8] exhibited low efficiency even on a small number of processors, yielding maximum speedups of less than 5 on 8-processor shared memory multiprocessors. To explain the lack of success in parallelizing circuit simulation, Bailey and Snyder [7] instrumented a sequential switch-level simulator to measure the inherent parallelism in six real circuits. The results showed surprisingly low parallelism even for very large circuits. For example, the average parallelism, or the average number of transistors switching at the same time (*real time*), is only 6.3 for a 32-bit RISC processor containing over 24,000 transistors; the average parallelism for a IIR digital filter containing over 27,000 transistors is 6.4. The highest parallelism is shown in large arithmetic circuits with regular structures. For example, the average parallelism is 50 for a 32-bit shifter with 8 stages, and 24 for a 32-bit Baugh-Wooley multiplier. The lack of simultaneous activities in the circuits gives an indirect explanation of the low speedups shown by the direct method simulators.

To justify our use of a large-scale multiprocessor, we give the results of a similar study on the available parallelism under SWEC. Our study is different in that we measure the optimal parallelism achievable on the *simulation time* axis, instead of on the *real time* axis. The latter metric does not reflect the fact that multiple time points of the circuit can be simulated in parallel, as long as their dependency is preserved. Therefore, the real time metric is only suitable for synchronous parallel implementations. In this section we present our results on circuit parallelism, which are far more encouraging than the results obtained by Bailey and Synder.

3.1 Benchmark Circuits

Table 1 lists the benchmark circuits we used for measuring parallelism and for subsequent performance comparisons. We were not able to produce the parallelism profiles for all benchmark circuits, because the time to produce them would be prohibitive.

The ripple adder and the two multipliers are combinational circuits. The register file is sequential in nature, but there is no feedback path among the regions because all nodes on a feedback path are tightly coupled, and are partitioned into the same region. The 4-bit counter consists of 4 T type flipflops. The 4-bit counter, the PLA state machine, and the SIMD datapath are sequential circuits with many feedback paths among the regions. C1355, C2670, C5315, and C7552 are drawn from the ISCAS benchmark suite; their functions are unknown to us.

3.2 Instrumentation Method

We instrumented the sequential SWEC program to record the activities of the simulation. Specifically, we assume that an unlimited number of processors is available, and that each evaluation is assigned to an idle processor as soon as it is "ready" to start.

Name	function	mosfets	nodes	regions	running time
ADDER	16-bit ripple adder	442	226	129	59 sec.
MUL 1	16-bit multiplier I	7190	2576	401	293 sec.
MUL 2	16-bit multiplier II	6234	3332	1101	5297 sec.
COUNTER	4-bit counter	170	88	51	22 sec.
PLA	PLA state machine	2117	717	507	1423 sec.
REGFILE	register file bit slices	4832	1559	404	538 sec.
SIMD	fast processor datapath	37939	18860	7413	64916 sec.
C1355	ISCAS benchmark suite	2306	1196	678	942 sec.
C2670	ISCAS benchmark suite	5364	2917	2033	5919 sec.
C5315	ISCAS benchmark suite	11260	5810	3730	21731 sec.
C7552	ISCAS benchmark suite	15394	7906	5272	43086 sec.

Table 1: Benchmark circuits: the last column is the sequential running time of SWEC on a Sun/4.

We use the actual running time on the CM5 to obtain a realistic estimate of the available parallelism. First, the average cost (execution time) of the evaluation of region r, $cost(r)$, is measured to microsecond precision on the CM5. $cost(r)$ is an estimate of the actual cost, because the cost may vary for different evaluations of the same region when the relaxation solver is used. We then augment the sequential data structure for r with the field *r.start*, which stores the earliest possible time at which the next evaluation can begin. *r.start* is initialized to be 0, and is constantly updated as the algorithm proceeds.

We define the *parallel running time* under our model is the time at which the last evaluation completes, which is $r.start + cost(r)$ from the last evaluation. A time plot of the number of processors busy processing some evaluation gives us the parallelism profile over the entire duration of the simulation.

Note that the evaluation cost we measured is the pure computational cost to set up the device models and to perform the linear system solves; it does not include the scheduling cost and other overheads required to start the evaluation. Thus, the parallel running time is a *lower-bound* on the actual running time of any parallel implementation on the CM5 using only parallelism between subcircuits. (We do not consider any fine-grained parallelism within evaluations.) The *optimal speedup*, or the *average parallelism*, is then

$$\frac{\text{sequential running time}}{\text{parallel running time}}$$

The earliest possible time at which an evaluation is considered "ready" depends on the scheduling policy of the parallel algorithm. We consider two types of parallelism: *static parallelism* and *runtime parallelism*. These two types of parallelism are discussed and compared in the next section.

3.3 Circuit Parallelism Profiles

3.3.1 Static (Conservative) Parallelism

In the outline of the SWEC algorithm we noted that the evaluation of a region cannot be performed correctly until all the relevant events have taken effect. Any region that has a signal path to another region can contribute such events. The following definition formalizes this relationship:

> **Definition:** Let Anc be the transitive closure of In, when they are viewed as binary relations. That is, $r_1 \in Anc(r_n)$ iff $r_1 \in In(r_2), r_2 \in In(r_3) ..., r_{n-1} \in In(r_n)$, for some $r_2, r_3 ..., r_{n-1}$.

Any region $r1 \in Anc(r)$ has a signal path leading to r.

Since events for time t may be generated by any evaluation before t, the evaluation of $r(t)$ must happen after $r'(t')$ has been evaluated for all $r' \in Anc(r)$ and $t' < t$. That is,

$$r(t).start = \max_{t'<t} \left\{ r(t').start + cost(r), \max_{\forall r' \in Anc(r)} \left[r'(t')start + cost(r') \right] \right\} \qquad (1)$$

We call the average parallelism under this scheduling policy *static parallelism*, or *conservative parallelism*. It is static because the dependency between regions is determined by their interconnection in the circuit; it is conservative because at runtime, a pending evaluation may wait for information that it does not need (e.g., when the evaluation of its fanin region does not produce an event). The conservative scheduling policy guarantees that all evaluations performed are correct.

Table 2 shows the conservative parallelism for some of the benchmark circuits. The parallelism profiles are shown in Figure 2. Note that there is always a burst of activity at the beginning of the simulation, when the circuit is converging to a stable (DC) state. The circuits are simulated long enough so that the start-up effect is not significant. The "tails" of the parallelism profiles indicate the existence of *critical paths* that dominate the computation time.

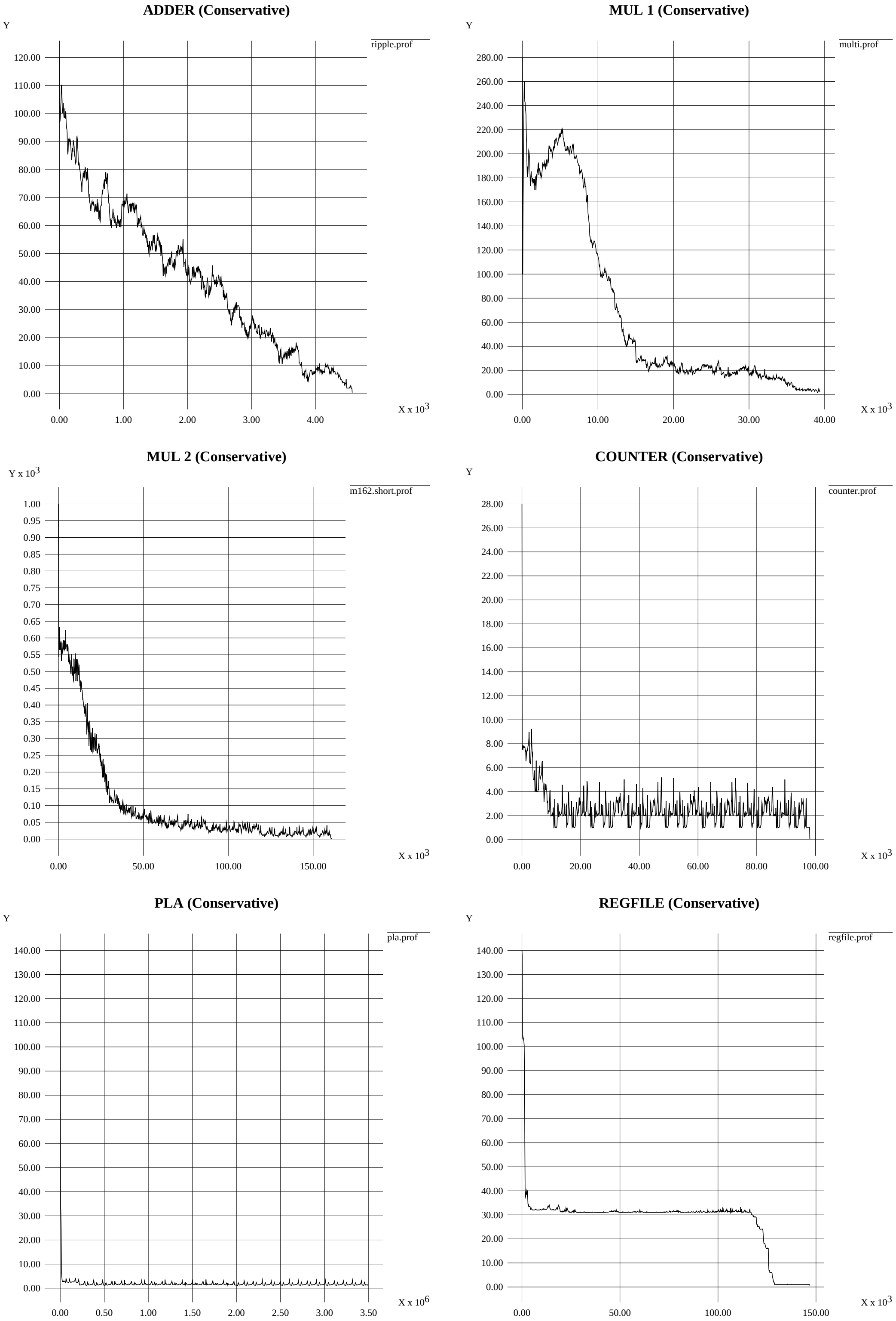

Figure 2: Conservative parallelism profiles: the Y axis denotes the amount of parallelism, and the X axis denotes the time.

ADDER	MUL 1	MUL 2	COUNTER	PLA	REGFILE
40.6	67.3	107.7	2.5	1.9	27.6

Table 2: Conservative parallelism of the benchmark circuits

ADDER	MUL 1	MUL 2	COUNTER	PLA	REGFILE
40.9	67.4	109.5	8.3	21.5	27.6

Table 3: Optimistic parallelism of the benchmark circuits

3.3.2 Runtime (Optimistic) Parallelism

The conservative scheduling policy assumes all evaluations generate events. This assumption can be lifted if we can have an *oracle* that tells the scheduler if all fanin events have taken effect, even before the state of the fanin regions are evaluated. The evaluation of a region can then take place ahead of its fanin regions, given that the remaining fanin evaluations will not produce new events. In this scenario, the earliest start time of an evaluation is

$$r(t).start = \max_{t'<t} \left\{ r(t').start + cost(r), \max_{\forall r' \in In(r)} \left[r'(t').start + cost(r') | \; r'(t') \text{ is an event} \right] \right\} \tag{2}$$

We call the average parallelism under this model *runtime parallelism*, or *optimistic parallelism*. It is a runtime metric because the dependency between regions is determined by the events generated at runtime; it is optimistic because a region makes optimistic assumptions on the values of its fanin regions to make progress.

Table 3 shows the optimistic parallelism for some of the benchmark circuits. The parallelism profiles are shown in Figure 3.

3.3.3 Comparison and Summary

Our results showed that the amount of parallelism usually grows with the number of regions in the circuit. In particular, combinational circuits with regular structures exhibit the highest parallelism among the benchmark circuits. This result is consistent with the findings by Bailey and Synder [7]. The computation in the register file is dominated by its 32 bit slices, which is reflected in its average parallelism (close to 32).

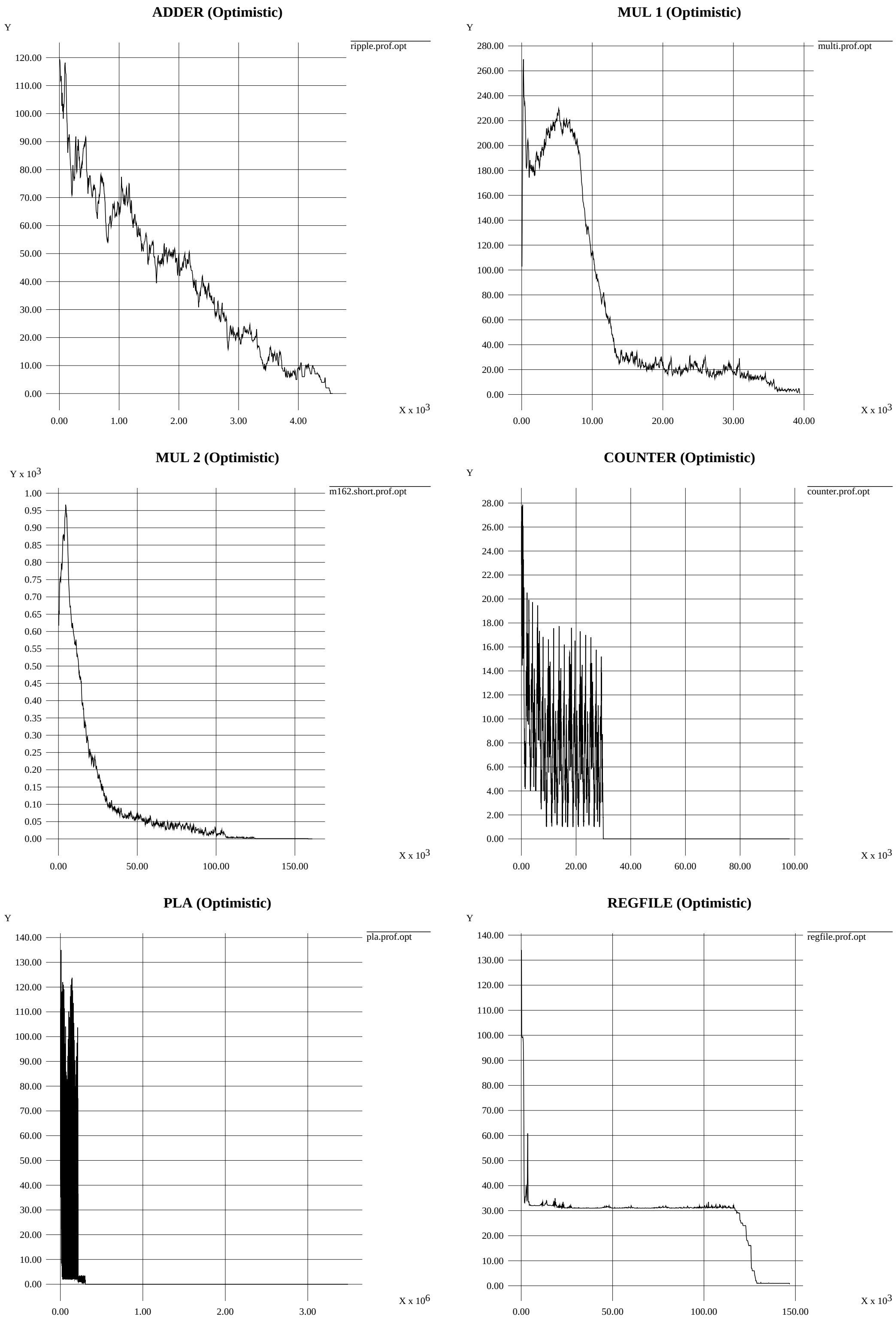

Figure 3: Optimistic parallelism profiles: the Y axis denotes the amount of parallelism, and the X axis denotes the time. Note how the parallelism profiles for COUNTER and PLA change.

Optimism has only minor effect on combinational circuits, as dependent computations can proceed concurrently in a *pipelining* fashion. However, optimism has a significant impact on the parallelism in sequential circuit with feedback paths. In particular, the PLA state machine receives an 11-fold increase in average parallelism. The dramatic improvement is best explained by the following observation:

> **Key Observation:** Let F_p be the set of regions on the feedback path p. $r, r' \in F_p \Rightarrow r \in Anc(r'), r' \in Anc(r)$. Assume all time points are distinct. Then no two regions in F_p can be evaluated in parallel under the conservative scheduling policy.

It is clear that if we evaluate two regions in a feedback path in parallel, the outcome of one may render the other incorrect, although the latency property of digital circuits indicates this is not the common case. The assumption of distinct time points is valid most of the time since the resolution of the time points in SWEC can be as fine as 1 picosecond, and most of the time steps are much larger than that.

Although we were not able to measure the SIMD processor datapath, the bus structure of the processor will certainly prevent the circuit from gaining high parallelism with the conservative scheduling policy. This observation leads us to the conclusion that, for large sequential circuits, conservative parallelism is not likely to be sufficient to keep a large scale multiprocessor busy.

On the other hand, we believe there is still a high degree of optimistic parallelism in large circuits. Consider the SIMD processor datapath as an example. Although the bus-oriented interconnection makes most parts of the circuit mutually dependent, the actual signal flow is carefully controlled by the timing scheme; most of the time the circuit is acting as a collection of independent functional units, whose signals are confined to its own latches or registers [2].

In summary, we show that the runtime parallelism in the benchmark circuits is sufficiently high to warrant the use of a large-scale multiprocessor. The rest of the paper presents our approach to exploiting runtime parallelism.

4 Load Balancing

In SWEC the evaluation of a time point for a region constitutes the unit of parallelization, and is referred to as a *task*. In this section we address the issue of load balancing, that is,

[2]There are other techniques (e.g., unit-delay simulation) that exploit this property in pipelined designs by handling clock cycles explicitly in the simulator. Our work subsumes these techniques in that we allow arbitrary feedbacks.

how to distribute the tasks among the processors. We start by exploring the alternatives for load balancing in distributed systems. Measurements from SWEC are used to illustrate the tradeoffs and to justify our choice of a static load balancing scheme. At the end of this section we also discuss the load balancing heuristics used in the current implementation.

4.1 Static Load Balancing vs. Dynamic Load Balancing

There are two basic alternatives to load balancing: a static load balancing scheme assigns tasks to processors in advance; a dynamic load balancing scheme assigns tasks on demand during the execution. In both cases assignment is done automatically at runtime.

Dynamic load balancing can maintain high processor efficiency without requiring *a priori* knowledge of the circuit's parallelism profile. In SWEC the activity of a region is highly dependent on the input pattern, and thus any static load balancing scheme will suffer certain loss of efficiency. However, dynamic load balancing incurs two types of costs that are particularly expensive in a distributed environment; these overheads are the *synchronization cost* and the *loss of data locality*.

The synchronization cost is incurred when information is managed in a centralized manner. The most straightforward implementation of a dynamic load balancer is a centralized task queue, from which all idle processors request tasks. The synchronization cost is then the sum of the communication time required to request a task, to assign a task, and the queueing delay due to access contention to the centralized queue. Since the synchronization overhead grows with the number of processors, dynamic load balancing is not likely to scale well for large multiprocessors. We are currently investigating a relaxed priority queue that is physically distributed among the processors. However, there is still some synchronization required to obtain approximate global load information.

The second kind of overhead, loss of data locality, is the main obstacle for dynamic load balancing. It would be impossible to replicate the data set of all regions on each processor for large circuits, as is evident in Table 4. Therefore, with dynamic load balancing, a processor may have to fetch the region data set from a remote processor. The average size of a region is above 1 KB, containing both the read-only region description and the writable states. The cost of transferring 1KB of data on the CM5 is at least several hundred microseconds, implying that the cost of an evaluation is at least doubled. Caching may be used to reduce the communication cost; however, it is not clear that the problem exhibits the working set property for large circuits.

Considering the above, we employ a static load balancing scheme in the parallel timing simulator. The memory saved by avoiding caching will be useful in implementing optimistic parallelization. Static load balancing also has the additional advantage of program

Name	data set	avg. cost	max cost	std. dev.
ADDER	213 KB	200 us	310 us	40 us
MUL 1	2.264 MB	920 us	2660 us	530 us
MUL 2	2.688 MB	340 us	4810 us	260 us
COUNTER	83 KB	220 us	690 us	90 us
PLA	881 KB	230 us	1850 us	190 us
REGFILE	1.820 MB	640 us	5200 us	1460 us
SIMD	17MB	400 us	10440 us	370 us
C1355	1.137MB	250 us	2060 us	120 us
C2670	3.012MB	240 us	720 us	80 us
C5315	5.805MB	250 us	4290 us	110 us
C7552	8.056MB	250 us	5070 us	70 us

Table 4: Statistics of the benchmark circuits: the data set size is the amount of memory used to store and process the circuits in SWEC; the next three columns list the statistics on the evaluation cost per region.

simplicity. The performance results in Section 7 will confirm that load balancing is not a major performance obstacle in most cases.

4.2 Static Load Balancing Heuristics

The static load balancing problem is to distribute the regions among the processors, so that the total running time is minimized. Finding the optimal distribution is a hard problem, because the activities of the regions can not be predicted prior to the simulation. Therefore, we assume that all regions are equally active.

By treating all regions equally, the problem becomes the estimation of the task costs, and the assignment of tasks to even out the total cost on each processor. In theory, the computational cost of an evaluation is linear in the number of transistors (for the model evaluation step), and cubic in the number of nodes (for the linear system solve step). This is not true in practice. First, there is a large fixed cost for each evaluation; second, a region usually contains only a few nodes and transistors, and the SWEC implementation switches from Gaussian elimination to a relaxation solver when the region is "large" (containing more than 8 nodes). Therefore, we use the sum $(N(r) + M(r))$ to estimate the cost of the region r, which tracks the actual cost well in practice. The sum also reflects the spaces required for storing the circuits and for implementing optimistic parallelism. The latter is very important when it comes to fitting a large circuit such as SIMD onto the node processors, each having less than 8 MBytes of memory available.

Finding the optimal partitioning of tasks is hard (NP-complete), and is not worthwhile because the task costs are merely estimates. The following approximation algorithm suffices in our case:

> *Assigning regions to processors*: put the regions in a list in arbitrary order; repeatedly remove a region from the list, and assign it to the processor that has the least amount of work so far.

The above algorithm guarantees a maximum unevenness of

$$\max_{\forall r}(N(r) + M(r))$$

We use a pseudo random algorithm for task assignment to avoid pathological cases. For example, one may assume that assigning regions in the same functional block (e.g., the regions in the shifter of SIMD) to the same processor would minimize the communication cost, and thus be the best partitioning scheme. However, it is possible that the functional blocks are mutually dependent and are never activated at the same time. Such a partitioning will then lose most of the parallelism.

5 Scheduling Techniques for Discrete Event Simulations

In this section we introduce the two well-known techniques for distributed event simulation, namely the conservative approach and the optimistic approach. We also compare the performance tradeoffs of these two approaches.

5.1 Introduction to Discrete Event Simulation

In discrete event-driven simulation, the world to be simulated is modeled as a collection of *logical processes*, each representing a part of the physical system. The logical processes communicate by sending time-stamped *messages*. We assume that the messages initiated by the same sender are always processed in order. The state of the system progresses forward in time as the processes take appropriate actions in response to the events. For convenience of discussion, we will use the physical entities in the timing simulator to refer to these abstract terms; that is, we use regions in place of processes.

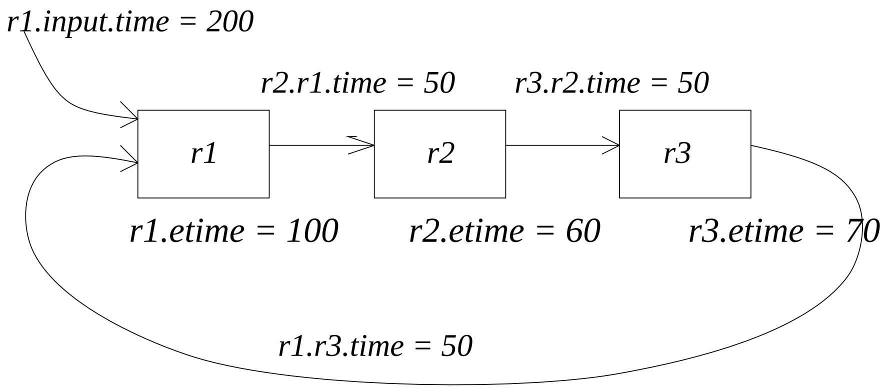

Figure 4: Deadlocks in conservative simulation: none of the three regions can proceed based on local information.

5.1.1 The Conservative Approach

The conservative approach is best represented by the Chandy-Misra algorithm [9]. Each region keeps a *logical clock*, which is analogous to $r.etime$ in the SWEC algorithm, to denote the progress of its state. The progress of a region is conveyed to its fanout via the timestamps of the events. The region $r1$'s perception of the time of its fanin region $r2$ is denoted by $r1.r2.time$, and is obtained from the timestamp carried by the latest event sent by $r2$ to $r1$.

The conservative algorithm schedules region r for evaluation when and only when:

$$r.etime \leq \min_{\forall r' \in In(r)} r.r'.time$$

A conservative event based simulation is not guaranteed to make progress, since there exists situations where none of the regions can proceed, and the simulation algorithm deadlocks. Figure 4 shows such an example. In the figure $r2$ is eligible for evaluation, but the lack of updated information on $r1$'s progress prevents $r2$ from being scheduled for evaluation. Deadlocks occur when there are feedback paths in the system, and can not be resolved without global information on the progress of the simulation.

The algorithm can be augmented to either avoid deadlock or detect and recover after its presence. These procedures collect global information to advance the timestamps in the system so that progress can be made. Both of these methods incur high communication or synchronization overhead in a distributed environment. Deadlock avoidance is usually done by sending *null messages*, which add to the communication overhead of the implementation. Deadlock detection, on the other hand, requires global communication and runs the risk of stalling processors while waiting for deadlock to be detected. Previous study [10] shows that deadlocks are commonplace in large digital circuits. Therefore, deadlocks can become the performance bottleneck in conservative simulation.

It is clear that the conservative scheduling algorithm always meets the scheduling criterion of equation 1 in Section 3.3.1. Therefore the conservative algorithm can only exploit the conservative parallelism in simulation.

5.1.2 The Optimistic Approach

The optimistic approach, or the Time-Warp approach [11], lifts the restriction that the simulation algorithm perform only correct evaluations. The assumption is that events are relatively rare. With the optimistic approach any evaluation is eligible for execution; if the result turns out to be wrong, it is discarded and the evaluation resumes from a previous time point. This operation is known as *rollback*, and the event causing a rollback is called a *straggler*. The only condition for starting an evaluation is that the time point and the data used must be consistent with *whatever has been seen so far*.

To restart correctly after a rollback, each region must maintain a sufficiently long *history* of the fanin events. To be able to cancel invalid events, and to prevent from restarting with the very first time point in response to every rollback, each region must also maintain a sufficiently long history of its own states. These are the space overheads of the optimistic approach.

Rollbacks also incur some computational overhead. The processor time spent on an invalid evaluation is wasted if it could have been spent on some other evaluation that would have produced a correct time point. Additional computation and communication may also be required to send *anti-messages* whose sole purpose is to undo the effect of invalid events (caused by stragglers or other anti-messages). Therefore, the evaluations must be properly prioritized to avoid the rollback overhead.

The main advantage of the optimistic approach is the increase in parallelism. The scheduling criterion used by the optimistic algorithm is clearly more general than equation 2 in Section 3.3.2. Therefore, the optimistic approach can potentially exploit the maximum runtime parallelism.

5.2 Summary and Comparison

The tradeoffs of the two approaches are summarized here.

The conservative approach:

- Never performs useless computation
- Needs less memory

- Exploits less parallelism
- Incurs deadlock avoidance/detection overhead

The optimistic approach:

- May perform useless computation
- Needs more memory
- Exploits more parallelism
- Incurs rollback and cancellation overhead

In Section 3 we showed that conservative parallelism is not sufficient to take full advantage of a large-scale multiprocessor. Since fast local memory is abundant in distributed memory multiprocessors, we are willing to trade memory for increases in parallelism. Therefore, we adopt the optimistic approach in the implementation of the parallel timing simulator.

6 Algorithms and Implementation

In this section we describe the implementation of the optimistic parallel timing simulator in detail. We start with a nondeterministic specification of the optimistic simulation algorithm, and then transform it into a efficient form for parallel execution on the CM5.

6.1 Specification of the Optimistic Simulation Algorithm

Before presenting the algorithm, we discuss three of the more subtle implementation issues: event messages, the concept of global time, and storage reclamation.

6.1.1 Effects of Event messages and Anti-messages

For the rest of this paper we assume that all message from the same region are processed in the order they are generated. The assumption greatly simplifies the states the algorithm has to keep to handle out-of-order messages (e.g., an anti-message that overruns the corresponding event message).

An event message carries the new state of a region. The effect of an event message is similar to that of the fanout operation in the SWEC algorithm, except that rollback occurs when the region produces states with time greater than the time of the event message received. These states may be inconsistent with the new event and must be thrown away. The region then backs up to a state with time less than or equal to the event time, and reschedules the next evaluation.

The purpose of an anti-message is to undo the effect of a previous event message. It first throws away the fanout's copy of the invalid event. It then discards all states with time greater than the time of the anti-message, which were produced using the invalid event. More anti-messages may, in turn, be generated to undo the effect of these discarded states.

The event messages and anti-messages do not have to be processed immediately upon their generation. We use $E(r)$ and $A(r)$ to denote the set of event messages and anti-messages from r that are yet to be processed. $E(r).time$ and $A(r).time$ denotes the minimum time of the messages in $E(r)$ and $A(r)$, respectively. Note that it takes $deg(r)$ event messages (anti-messages) to send (cancel) a state of r to all regions in $Out(r)$. In a practical implementation, only the parts of the state required by the fanout regions are sent.

For the presentation of the algorithms, we denote a message by the triple $< Type, Event, Time >$, where $Type$ is the type of the message ("event message" or "anti-message"), $Event$ is the state to send or cancel, and $Time$ is the timestamp of $Event$.

6.1.2 Termination Detection

Unlike in conservative simulation, the progress of a region in optimistic simulation cannot be assessed using local information; a region's view of its progress $r.etime$, or its *local virtual time* ($r.LVT$), may actually decrease due to rollbacks by the event messages, or due to cancellations by the anti-messages. To assess the system's progress we need to measure the *global virtual time* (GVT), which is defined as:

$$GVT = \min_{\forall r}\{r.LVT, A(r).time, E(r).time\} \tag{3}$$

In other words, the global virtual time represents the minimum time of all entities at some instant of the simulation.

The correctness of the optimistic simulation algorithm is guaranteed by the following facts:

Fact 1 *GVT never decreases.*

This follows from the fact that a straggler or an anti-message for time t can only decrease LVT to some time greater than t.

Fact 2 *GVT eventually increases if all regions and messages are eventually processed.*

In particular, GVT increases when all entities with the minimum time are processed. The program can terminate when GVT increases beyond the total simulation time.

Fact 3 *All valid states with time less than or equal to GVT have been evaluated; all invalid states with time less than or equal to GVT have been discarded.*

In any practical application, the state of the system must be collected after some duration of simulation. Fact 3 tells us that all states with time less than or equal to GVT are valid, and they are complete up to GVT. Such states can then be safely collected and produced as output of the simulation.

6.1.3 Fossil Collection

So far we have assumed unlimited space for storing the histories, whereas in practice, there is insufficient memory for storing the complete histories. We refer to the unnecessary states as *fossils*, and the reclamation of their space is the *fossil collection* problem. We know from the definition of GVT that a state will never be re-examined if its timestamp is less than or equal to GVT. Fossil collection is simply to reclaim the space of all such states. Note that all fossils are valid states by Fact 3.

In addition to keeping all states with timestamps greater than GVT, at least one state must be retained for each region, so that extrapolations and rollbacks can proceed correctly.

6.1.4 Nondeterministic Optimistic Simulation Algorithm (OPT)

We now give a nondeterministic specification of the optimistic simulation algorithm (OPT). The algorithm consists of a set of operations that appear to execute atomically. These operations can be interleaved in any order.

Each region holds a history of states evaluated so far, referred to as the *state history*; it also holds a history of events for each region in $In(r)$, referred to as the *event history*. In addition, there is a FIFO message queue for each edge in G, which stores the event messages and anti-messages that have not yet been processed. The variables $r.etime$ and $r.time$ are initialized as in SWEC.

The global variable GVT holds a copy (possibly stale) of the global virtual time. GVT is initialized to 0, and is updated as needed. The algorithm repeatedly selects any of the following operations:

- *Termination.* If GVT is greater than the total simulation time, exit.
- *Update GVT.* Compute the new GVT according to equation 3.
- *Collect fossil.* Pick a region r containing states with time less than or equal to GVT (fossils), discard all fossils (except the latest state).
- *Schedule.* Pick a region r such that $r.etime$ is within the total simulation time. Evaluate the state of r at $r.etime$ as in SWEC. Append the new state to the state

history of r. If the new state is an event, generate the event messages and add them to their corresponding message queues.

- *Send.* Pick the first message $< Type, Event, Time >$ from any of the nonempty message queues. Suppose the message is for region r'. Process the message as follows:
  - If $Type$ is "event message", append $Event$ to the event history of r'; otherwise, remove $Event$ from the event history of r'.
  - Generate the anti-messages for all events of r' with time greater than $Time$. Add the anti-messages to their corresponding message queues.
  - Discard all states of r' with time greater than $Time$ (*rollback*).
  - Restore $r'.time$ to the time of the latest state if necessary.
  - Recalculate $r'.etime$ as described in Section 2.3.2.

6.2 The Distributed Optimistic Simulation Algorithm (DOPT)

We now transform OPT into a parallel form that is suitable for execution on a distributed memory multiprocessor. The correctness of the parallelization is demonstrated by its *serializability* with respect to the nondeterministic specification: A parallel execution is *serializable* if it is equivalent to some sequential execution.

The operations will execute concurrently, but behave as if they were executed one after another.

6.2.1 Distributed Implementation of Operations

Recall that the regions are statically partitioned among the processors. To specify the partitioning we define $proc(r)$ as the processor on which region r resides, and $reg(p)$ as the set of regions residing on processor p. In a distributed memory environment, accessing data on processor p by processor q will involve both p and q. The detailed protocol for remote accesses is dependent on the machine architecture, and will be discussed later.

Algorithm OPT is mapped to a distributed memory multiprocessor as follows. The histories of r and the message queues storing messages from r reside on $proc(r)$. The variable GVT is replicated on all processors. Each processor executes the same algorithm OPT, except that the selection of r in *Collect fossil*, *Schedule*, and *Send* on processor p must satisfy $r \in reg(p)$. The computation is always performed on the processors that own the data to be updated.

Termination detection, *Collect fossil* and *Schedule* are all local operations. *Send* requires a local update to the message queue, followed by a remote update to process the message, if the fanout region resides on a different processor.

Update GVT requires accessing all regions, and is thus a global operation involving all processors in the system. The computation proceed as the independent local min to compute the *processor virtual time* followed by a total min to compute the global virtual time, since

$$GVT = \min_{\forall p} \min_{\forall r \in reg(p)} \{r.LVT, A(r).time, E(r).time\}$$

The detailed protocol for calculating GVT is architecture dependent. We will leave the discussion to Section 6.2.5.

6.2.2 Space Management

The time-warp algorithm is not guaranteed to work if the space for storing the states and the messages is bounded. For example, the *Send* operation for an event message may fail to find space to store the new event at the fanout, or a rollback may fail to find space in the message queue to store the anti-messages. A flow control protocol was suggested by Jefferson [11] to solve this problem. The protocol returns a message when space cannot be allocated, and the returned message causes the sender to rollback and try again.

The *cancel back* protocol described above complicates the proof of correctness of the algorithm, and it may add communication overhead to the algorithm. Our approach to solving the space management problem is to place some restrictions on the scheduling of the operations.

We require that a *Schedule* operation for a region r be selected only after the previous event messages from r have been processed. This requires acknowledgement of all event messages. Therefore, there can be at most one event message per message queue. Furthermore, the number of anti-messages in each message queue of r is at most the size of the state history of r plus 1. This is derived as follows. Let the number of events in the state history of r be $e(r)$. $e(r)$ decreases by one when an old event is invalidated, and increases by one when a new event is generated. Since messages in the same message queue are always processed in order, the message queues of r maintains a partial but continuous log of such activities. It follows that, for each message queue of r, the difference between the number of event messages and the number of antimessages denotes the net increase of $e(r)$ for some period of time. Since the number of event messages per queue is at most one, and $e(r)$ is at least zero, we have the desired bound on the number of anti-messages per queue.

Because messages in the same message queue are processed in order, the maximum number of states in the state history of r is at least that in the corresponding event history of $Out(r)$. We assume the sizes of all histories are the same. The *Schedule* operation can then proceed as long as there is space left *locally.* No flow control is required to make sure there is space at the fanout. Since the number of fanouts for an event can be quite large, a great deal of flow control overhead can be avoided. Some space overhead is incurred because the event history always store a subset of the states in the corresponding state history.

Note that the anti-messages need not be stored explicitly; a count of the events to cancel will do. This is because all event messages are processed and stored at the remote processors before their anti-messages are processed. The rollback operation can then proceed correctly using the stored information before it is removed.

6.2.3 Concurrency Among the Operations

The key to parallelization is to exploit the concurrency among the operations, while maintaining the serializability of the execution.

First note that any set of operations that access disjoint data sets can be performed concurrently, and the result of the parallel execution is equivalent to some sequential execution of the operations.

Termination is a local check on GVT, and runs atomically on each processor.

Collect fossil updates the data structure of r, and thus may conflicts with the *Send* operations initiated by other processors. We let *Collect fossil* run atomically, because it is a fast local operation, and it seldom needs to be invoked.

Schedule is a long running local operation, and its data set may conflict with a remote *Send* operation. We have the *Send* operation invalidate a running *Schedule* if they update the same region and if the *Send* operation would cause a rollback when executed immediately after the *Schedule* operation. Therefore, *Schedule* can be interrupted by a remote *Send* without violating serializability.

Send is a long running operation requiring communication to remote processors. Different *Send* operations can run concurrently as long as the remote update is made atomic.

Update GVT cannot run concurrently with *Send*, since a global snapshot of the minimum time cannot be obtained by examining the local states individually. This is illustrated in Figure 5. Therefore, the execution of *Update GVT* requires all processors to synchronize – the processors wait until their running *Send* operations complete, and do not select *Send* during the execution of *Update GVT*. This protocol is more restrictive than the asynchronous solution proposed in [11], and may stall processors unnecessarily if not implemented wisely. We will show an efficient implementation in the next subsection.

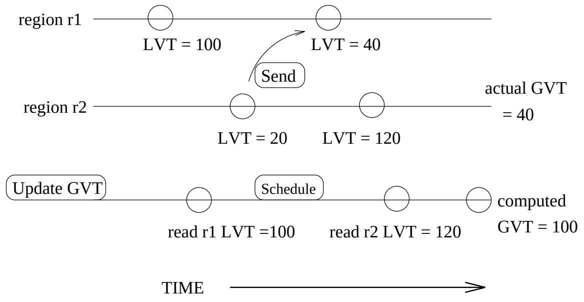

Figure 5: Violation of serializability: the computed GVT is an over-estimate of the actual GVT.

6.2.4 Scheduling of the Operations

Proper scheduling is essential for performance. The algorithm should not schedule an operation whose condition for execution is not met at all (in *Collect fossil* for example). This is done by adding states to the algorithm to incrementally record the operations eligible for execution. The following states are added for the *Schedule* and the *Collect fossil* operations:

- L_{fossil}: the list of all regions which run out of space in some of their state or event histories. Only the regions on L_{fossil} are considered for the operation *Collect fossil.*
- L_{done}: the list of all r such that $\min\{r.etime, A(r).time, E(r).time\}$ is greater than the total simulation time. There is clearly no point in evaluating $r(t)$ when t is beyond the range of interest. Such regions become eligible for *Schedule* when they are roll backed to within the total simulation time.
- L_{ready}: the list of all other regions. These regions are eligible for the *Schedule* operation.

The next improvement is to evaluate the regions on L_{ready} that will most likely lead to valid states. Application-specific information should be utilized as much as possible to judge the "goodness" of an evaluation to avoid the rollback overhead. In SWEC we can use the function $h_{predict}$ to estimate the time of the next event. Specifically, we add the field t_{next} to every event history of r to store this estimate. The estimates can be piggybacked on the event messages without significant communication overhead. The scheme leads to the following three priority classes for L_{ready}:

- *Conservative.* The evaluation for r is conservative if the minimum time of the most recent states in the event histories of r is greater than or equal to

$\min\{r.etime, A(r).time, E(r).time\}$. That is, r's fanin regions have progressed ahead of r. Note that a conservative evaluation is not necessarily valid.

- *Speculative.* The evaluation for r is speculative if it is not conservative, and the minimum of all t_{next} of the event histories in r is greater than or equal to $\min\{r.etime, A(r).time, E(r).time\}$. That is, some of r's fanin regions have not reached $r.etime$, but their predicted next evaluation times are past $r.etime$.
- *Unlikely*: An evaluation is unlikely if it is neither conservative nor speculative. Unlikely evaluations are rarely valid.

The regions are scheduled in increasing order of $\min\{r.etime, A(r).time, E(r).time\}$ within each class. This corresponds to the traditional Time-Warp scheduling heuristics that evaluations with smaller virtual times are scheduled first.

To quickly propagate the new information generated by *Schedule*, we start all *Send* operations for r immediately after the evaluation of r. We choose to block the processor until all running *Send* operations complete, as they can be executed concurrently and the total latency is not large. This simplies the scheduling of the *Update GVT* operation. The *Send* operation interrupts a running operation whenever the concurrency model allows.

The scheduling of *Update GVT* must be synchronized to avoid stalling processors unnecessarily. A processor selects the operation only when some *consensus* is reached by all processors. The frequency of *Update GVT* operations must be balanced between the synchronization overhead of the updates, and the stalls because the valid states are blocked from evaluation due to the lack of space. The following heuristics seems to work well:

> A processor initiates *Update GVT* when a region is added to L_{fossil}, when L_{ready} is empty, or when the highest priority region in L_{ready} is *unlikely*.

The following is the scheduling loop of the distributed optimistic simulation algorithm (DOPT):

- *Termination.* If GVT is greater than the total simulation time, exit.
- *Update GVT* and *Collect fossil.* If all processors have initiated *Update GVT*, do the following:
  - Wait until all processors select *Update GVT*
  - Compute $GVT = \min_{\forall p} \min_{\forall r \in reg(p)}\{r.LVT, A(r).time, E(r).time\}$
  - *Collect fossil.* For all r in L_{fossil}, discard all fossils except the latest states. Move r to L_{ready} if some space is reclaimed.

- *Schedule* and *Send.* Pick the highest priority region r in L_{ready}. If no such region can be found, initiate *Update GVT* and repeat the scheduling loop. Otherwise, do the following:
  - Initiate *Update GVT* if r is *Unlikely.*
  - *Schedule.* Evaluate r at $r.etime$ as in OPT, if $r.etime$ is within the total simulation time.
  - Put r in L_{ready}, L_{done}, or L_{fossil} as is appropriate.
  - *Send.* If a new event is produced, initiate all *Send* operations as in OPT to propagate the new event. The unprocessed anti-messages are piggybacked on these event messages. Wait until all *Send* operations complete.

Termination is tried first because it is inexpensive. *Collect fossil* is done only when *GVT* changes. It is clear that all regions or messages will be processed eventually in DOPT.

6.2.5 Implementation on the CM5

We now map the architecture dependent components of the DOPT algorithm to the CM5, which is a distributed memory multiprocessor.

We use the Active Communication Layer CMAM [12] to deliver the event messages and anti-messages. The current implementation of CMAM requires the programmer to insert explicit polls to receive incoming messages. Upon receiving the message, the communication layer invokes a user-specified handler to process the message.

CMAM exposes the two distinct data networks in the CM5 (the left and the right network) to the programmer. The bursts of communication by the *Send* operations exceed the bandwidth of a single network, and we have to alternate between the left and the right networks when sending the packets. The aggregate bandwidth of the two networks seem to be adequate for our timing simulator. If only one of them is used, it is seriously congested and the program exhibits a slow-down between 10% to 500% (30% for most circuits). The congestion may be caused by contention at the receiving processors, or by insufficient polls.

Since anti-messages are essentially small integer counts, they are always piggybacked on the event messages. Instead of processing the event immediately upon reception, the incoming messages are queued in the region data structure, and scheduled for processing when the required condition is met. The space needed to queue the messages is bounded, as each processor can have at most one outstanding message per fanout (i.e., per edge in G). The queueing of messages guarantees the atomicity of the critical sections in the operations, while allowing sufficient polls to rapidly remove messages from the network to avoid congestion.

The global consensus is done by the asynchronous global-or bit in the CM5 control network. A processor asserts 0 to the global-or bit when it is ready to synchronize, and resets the bit to 1 after *Update GVT* is done. Consensus is detected when the global-or bit reads 0, at which time the processors enter a global barrier to drain the *Send* operations. When all processors reach the barrier, they compute their processor virtual time and combine the result using the synchronous global-min primitive in the CM5 control network.

The time a processor stalls in the barrier is bounded by the time for the other processors to execute one iteration of the algorithm. The frequency of *Update GVT* is reasonable using our synchronization heuristics. Therefore, the total loss of efficiency due to synchronization is minimal.

7 Performance

In this section we summarize the performance of the parallel timing simulator on the benchmark circuits. The most important control parameter in the simulator is the maximum size of the history. The default configuration of the simulator allows 100 states per history. We will show how different settings affect the performance of the simulator.

7.1 Experiments and Results

The speedup is calculated as the ratio of the sequential running time on a Sun/4 (given in Table 1) and the parallel running time on the CM5. The CM5 nodes contain the same processors as the Sun/4, although the memory hierarchies are different. We were not able to run the sequential program on a single node of the CM5 because of the lack of space. All timing numbers are taken after the actual simulation starts. We exclude the I/O and preprocessing time needed to set up the data structures, to broadcast the required data to the CM5 backend, and to write the results to the output files. The I/O and preprocessing time ranges from a few minutes for ADDER to over 15 minutes for SIMD; it can be reduced after I/O nodes are installed in our CM5 system.

We first ran the timing simulators on all benchmark circuits under the default configuration with 100 states per history, varying the number of processors used in the experiments. The speedup curves are shown in Figure 6. To investigate the effect of history size on performance, we also ran some of the benchmarks on 128 processors using history sizes of 50, 100, 200, 300, and 400 states. The results are given in Table 5.

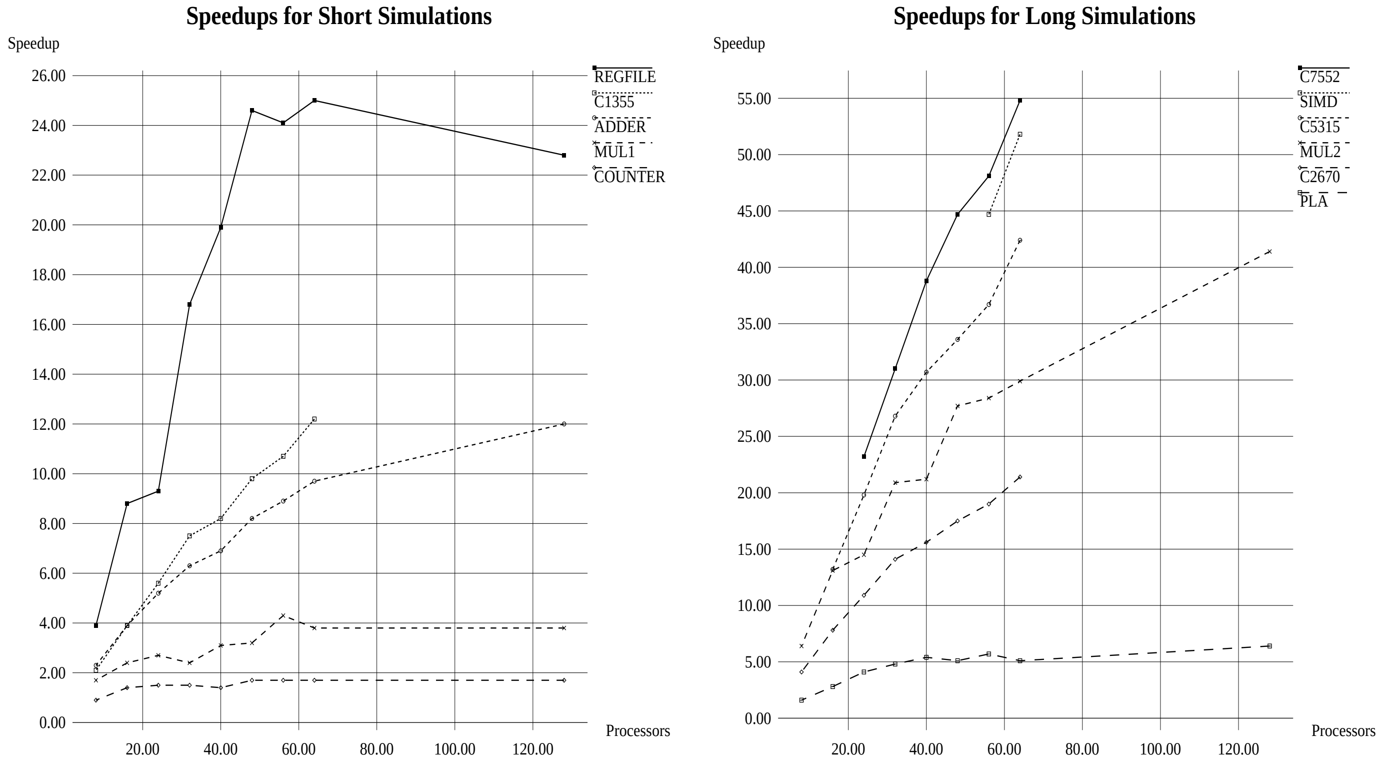

Figure 6: Speedup curves: short simulations include circuits that took less than 1000 seconds to simulate on a Sun/4. The 128-processor timings for some of the benchmark circuits are missing, because our installed CM5 was scaled down to 64 processors when they was obtained.

Name	50	100	200	300	400	worst	best
ADDER	8.56	9.68	14.92	15.99	13.4	8.56(50)	15.99(400)
MUL 1	3.89	3.77	4.08	5.56	5.38	3.77(100)	5.56(300)
MUL 2	29.89	41.37	56.55	53.93	62.17	29.89(50)	62.17(400)
COUNTER	1.34	1.74	1.52	2.48	2.26	1.34(50)	2.48(300)
PLA	10.20	6.37	5.06	4.04	3.86	3.86(400)	10.20(50)
REGFILE	20.48	22.81	22.32	22.54	22.75	20.48(50)	22.81(100)

Table 5: Effect of history size on performance: the speedups for different history sizes are given in the table, with the worst and the best among all 5 configurations highlighted for each circuit. We were not able to run SIMD for most of the larger configurations due to the lack of memory space.

7.2 Analysis

We note that the performance of the simulator improves when more processors are used, except for some minor perturbation for REGFILE. The peak speedups of our simulator are far greater than those reported for similar timing simulators in [2, 3]. The speedup for SIMD is particular encouraging, and it shows the feasibility of using large distributed memory multiprocessors to perform large simulations (optimistically).

However, the speedup is usually below the theoretical parallelism in Section 3.3.2. The difference is due to the overhead of data management, scheduling, and communication that arise in practice.

One main loss of efficiency is from the communication overhead incurred by parallelization. Table 6 lists the communication overhead for some of the benchmark circuits using the default configuration on 64 processors. Note that the wait time is an over-estimate; the processor can service remote *Send*s while waiting for its own *Send*s to complete, but this overlap is not taken into account. The send time is a useful estimate, as the processors can only remove and queue packets from the network while they block. It may be possible to hide the latency of *Send* by overlapping it with the *Schedule* for another region. However, the overlapping scheme complicates the presentation of the algorithm, and its performance benefit may not be significant – a large portion of the communication overhead is because the processor is blocked by the communication layer when shipping messages to the network (the current implementation of CMAM keeps polling the network interface until it

	ADDER	MUL 1	MUL 2	COUNTER	PLA	REGFILE	SIMD
send	5.8%	5.4%	9.6%	1.0%	16.7%	1.1%	16.7%
wait	9.2%	9.0%	8.6%	2.6%	9.2%	1.8%	25.0%

Table 6: Communication overhead in the parallel timing simulator. The table lists the percentage of time spent in sending packets, and in waiting for *Send* to complete.

is ready to accept the message). If the communication layer could be made non-blocking, the overlapping scheme would be worth implementing.

Another loss of efficiency is due to the static load balancing scheme. The typical bad case for static load balancing is MUL 1. Post-mortem analysis showed that the activities in MUL 1 are highly concentrated, which is indicated by the imbalance of time the processors spent in *Schedule* for each region. The runtime behavior of MUL 1 is not compatible with our load balancing assumption, and thus it leads to poor results. The other design of a 16-bit multiplier, MUl 2, showed much better results because the size and activity of the regions are well balanced. Note that if we were to increase the efficiency for MUL 1 by dynamic load balancing, we would introduce much higher communication overhead.

The benefit of optimistic parallelization is demonstrated by the speedups for the sequential circuits. Notice that even with the overhead of optimistic parallelization, the *actual* speedup achieved with PLA is far greater than the theoretical speedup achievable by the conservative method. The actual speedup for COUNTER is also close to the theoretical maximum speedup for conservative simulation. We believe the advantage of optimism increases for larger sequential circuits.

An interesting side benefit of optimistic parallelization is the decreases in scheduling cost. Due to the nature of optimistic parallelization, each processor performs local scheduling using a priority queue that is $\frac{1}{P}$ the size of the priority queue Q in SWEC, where P is the number of processors. Therefore, the cost of each access to the priority queue is reduced, while the number of accesses increases only slightly if a good scheduling heuristics is used to avoid excessive rollbacks. This contributes to the overall speedup when the number of regions is large, as we actually observed superlinear speedup for SIMD during the start-up transient of the simulation [3].

The size of the history impacts the performance of the simulator. Table 5 shows that

[3]This result indicates that, with a very good scheduling heuristic, we might improve the uniprocessor performance by simulating DOPT on one processor!

the speedup usually grows with the number of states in the history, with PLA as an exception. The adverse effect of increasing history size for PLA probably comes from the rollback overhead due to excessive speculation, plus the increased cost for managing longer histories.

In summary, our parallel timing simulators usually shows far more encouraging results than was previously reported. Limits to further speedup includes the communication overhead, the speculation penalty, and the load imbalance due to the use of a static load balancing scheme. The design decisions we made are justified by most circuits.

8 Previous Work on Parallel Circuit Simulation

Previous works on parallel circuit simulation largely fall into the following categories: direct method simulation, waveform relaxation, and timing simulation.

The parallel direct methods typically consists of two modules: the *LOAD* module and the *SOLVE* module. The *LOAD* module performs model evaluations, and is more parallel than the *SOLVE* module, which solves large sparse matrices. Previous works on parallel direct method simulators by Sadayappan and Visvanathan[8] and by Yang [4] showed maximum speedups of 3.6 for circuits containing about 3000 nodes on a 6 processor Alliant FX/8, and 4.53 for circuits containing about 1000 nodes on a 8 processor Alliant FX/8, respectively.

Previous Work on parallel waveform relaxation includes those by Wen and Saleh [13], and by Smart and Trick [14], both showing a maximum speedup of 4 on a 8 processor Alliant FX/8.

Parallel timing simulators include XPSIM[3] and EMU[2]. XPSIM gives a maximum speedup of 4.6 for a decrementer circuit with 319 regions on a 11 processor Sequent. EMU gives the best result so far. Its maximum speedup on a distributed memory hypercube with 64 nodes is 20 using the synchronous version, and 10 using the asynchronous version, for a fuzzy controller containing 8620 transistors [4] . The difference in performance is due to the communication overhead incurred by the asynchronous algorithm.

9 Conclusions

Some limitations of our current work suggest topics for future research:

[4]Lucco [1] shows a maximum speedup of 33 for EMU on circuits containing more than 20,000 transistors, using a 64-node hypercube. The speedup is from the "fastest" version of the program and is relative to a VAX/780.

- The scale of simulation (in terms of circuit size, precision, and duration of simulation) is constrained by the amount of memory, and by the bandwidth of the communication network. The former can be solved by adding more memory or processors to the system; the latter, however, will remain a bottleneck, as we have already consumed most of the available bandwidth just to communicate the essential data.

- We implemented only a subset of the functionality of SWEC in the parallel timing simulator. For example, the parallel timing simulator does not handle explicit resistors, floating capacitors, and lossy transmission lines. However, it is adequate for most digital MOS circuits.

- We argued that distributing regions according to their functional blocks is not a good idea. However, there exist situations where a pseudo-random distribution policy is a bad idea. For example, regions implementing a flip-flop or a ring oscillator should be placed on the same processor whenever possible, since they are guaranteed to be mutually dependent. Therefore, we expect the user's intervention in load distribution to be inevitable in a commercial quality implementation.

- The load balancing scheme and the configuration of simulation (history size) are determined statically. However, it may be possible to fine-tune these parameters dynamically based on the statistics collected at runtime.

- Accesses to histories, rollbacks, and cancellations are currently implemented in an ad-hoc manner. These data structures and operations can be generalized to a distributed object that would be useful for parallelizing other applications.

We summarize our work below:

- We measured the maximum parallelism available in the SWEC simulation algorithm, showed that the amount of parallelism is sufficient to justify the use of a large multiprocessor. In particular, we showed that optimism is essential for exploiting parallelism in sequential circuits.

- We gave a design of an optimistic simulation algorithm, and mapped it to a distributed memory multiprocessor.

- Our parallel timing simulator showed much better results than previous attempts in parallel circuit simulation. The best speedup for the largest circuit, SIMD, is over 50 on 64 processors; the best speedup for the sequential circuit PLA is over 10, higher than the theoretical maximum of 1.9 using conservative simulation.

- The parallelization of SWEC also serves as an interesting example of general parallel programming on a distributed memory multiprocessor.

In conclusion, our work shows that the time cost of timing simulation for digital MOS circuits can be significantly reduced by applying optimistic simulation on a large multiprocessor. The scalability of some of the larger circuits leaves room for even better results on larger machines using larger circuits.

Acknowledgements

I would like to thank my advisor, Kathy Yelick, for her advice, encouragement, and support for my work, and for the very stimulating research environment she has fostered. I also would like to thank Shen Lin for the basis of this work and for his early involvement in this project, John Wawrzynek for his valuable comments as a circuit designer, Bertrand Irissou for his help with the collection of benchmark circuits, and Jeff Jones for proofreading drafts of this paper. The project is supported in part by SRC Contract Number (92-DC-008), AT&T, Lawrence Livermore National Laboratory, and National Science Foundation Infrastructure Grant number CDA-8722788.

References

[1] S. Lucco, K. Nichols, *A Performance Analysis of Two Parallel Programming Methodologies in the Context of MOS Timing Simulation*, *Proc. Spring COMPCON 87'*, February 1987.

[2] E. DeBenedictis, B. Ackland, *Circuit Simulation on a Hypercube*, *Distributed Simulation Conference*, 1988.

[3] T. Lee, *Parallel Circuit Simulation: A Case Study in Parallelizing Programs*, Master thesis, Computer Science Division, University of California at Berkeley, 1991.

[4] G.C. Yang, *PARASPICE: A Parallel Circuit Simulator for Shared-Memory Multiprocessors*, *Proc. 27th Design Automation Conference*, 1990.

[5] S. Lin, E. Kuh, M. Marek-Sadowska, *A New Accurate and Efficient Timing Simulator.* Also appears as *SWEC: A StepWise Equivalent Conductance Timing Simulator for CMOS VLSI Circuits*, *Proc. EDAC*, Feb. 1991.

[6] R. Saleh et al., *Parallel Circuit Simulation on Supercomputers*, *Proc. of the IEEE*, Vol.77, No.12, December 1989.

[7] M. Bailey, L. Snyder, *An Empirical Study of On-Chip Parallelism*, *Proc. 25th Design Automation Conference*, 1988.

[8] P. Sadayappan, V. Visvanathan, *Circuit Simulation on a multiprocessor*, *Proc. Custom Integrated Circuit Conf.*, May 1987.

[9] K.M. Chandy, J. Misra, *Asynchronous Distributed Simulation via a Sequence of Parallel Computations*, *Communications of the ACM*, Vol.24, No.11, April 1981.

[10] J. Pal Singh et al., *SPLASH: Stanford Parallel Application for Shared-Memory*, *Computer Architecture News*, Vol.20, No.1, March 1992.

[11] D.R. Jefferson, *Virtual Time*, *ACM Transactions on Programming Languages and Systems*, Vol.7, No.3, July 1985.

[12] T. von Eicken et al., *Active Messages: a Mechanism for Integrated Communication and Computation*, *Proc. 19th Annual International Symposium on Computer Architecture*, May 1992.

[13] Y.C. Wen, K. Gallivan, R. Saleh, *Parallel Event-Driven Waveform Relaxation*, *Proc. International Conf. on Computer Aided Design*, 1991.

[14] D.W. Smart, T.N. Trick, *Increasing Parallelism in Multiprocessor Waveform Relaxation*, *Proc. International Conf. on Computer Aided Design*, 1987.