Symphony: A Simulation Backplane for Parallel Mixed-Mode Co-Simulation of VLSI Systems

Antonio R.W. Todesco and Teresa H.-Y. Meng
Computer Systems Laboratory
Stanford University, CA 94305

Abstract — In this paper we present an integrated simulation paradigm in which parallel mixed-mode co-simulation is accomplished by integrating sequential simulators in a software simulation backplane. Distributed conservative event-driven scheduling is used in combination with an efficient deadlock-free mechanism for handling synchronous feedback circuits. Simulation concurrency can be further increased by utilizing circuit pipeline. The resulting parallel simulation backplane is capable of concurrently simulating systems at circuit, switch, gate, RTL and behavioral levels. We implemented this parallel mixed-mode simulator on both the iPSC/860 message-passing machine and the DASH shared-memory multiprocessor. Experimental results are presented.

1 Introduction

The last several years have seen a steady growth in the complexity of IC and system integration. As a consequence, there is a need for longer, larger and more realistic simulations performed within a finite design-to-market time. Usually multiple simulators are used in the design pass because no single simulator addresses all modeling, performance and verification issues.

Multi-level mixed-mode co-simulation has a great potential to efficiently simulate large systems containing both digital and analog components, with portions of the system described at different levels of abstraction. Limiting factors for large-scale mixed-mode simulation are the speed of sequential simulation software, the relatively high cost, and the low flexibility of hardware accelerators. With the growing availability of powerful parallel processing machines, parallel co-simulation is a viable approach to speeding up simulation of large systems. Furthermore, integration and extensibility could be achieved using a parallel software simulation backplane.

One of the major issues in integrating sequential simulators is the handling of feedback circuits, which may lead to simulation deadlock due to the following reason. The interface between simulators usually consists of an input update phase followed by a simulation interval phase during which output events are produced. By the very nature of this interface, the outputs are only known up to the time that the inputs have been simulated. At the beginning and the end of each simulation step, the time across the simulation is the same for every input and output node. As a consequence, there is no output *lookahead*[1][6] in the simulator instantiation, which means that the resulting simulation will not have the *predictability* [6] property. If there is a loop of interconnected blocks, the system will deadlock because it does not have a positive lower bound on how far ahead the outputs can be predicted in relation to its inputs.

This research was supported in part by ARPA, and a fellowship from CAPES, Brazil.

In this paper, we present an integrated simulation paradigm for implementing parallel multi-level mixed-mode co-simulators. Our integration approach is based on a distributed kernel that uniformly and efficiently manages multiple instantiations of sequential simulators. The kernel uses a distributed deadlock-free scheduling algorithm that handles feedback-connected circuit components to allow for efficient parallel execution. This parallel simulation paradigm has been demonstrated with a prototype that integrates SPICE3 [21], IRSIM [18] and THOR [2], implemented on both the iPSC/860 message passing multiprocessor and the DASH [14] shared-memory multiprocessor. Its performance and speedups due to multiprocessing are quantified and analyzed by simulating mixed-mode VLSI circuits with both analog and digital components.

The paper is organized as follows. We will present in section 2 a parallel discrete-event scheduling algorithm capable of handling synchronous feedback circuits efficiently. In section 3, we will describe Symphony, the parallel mixed-mode software simulation backplane based on this discrete-event scheduling algorithm. In section 4, we will analyze Symphony's performance using representative examples. In the final section, we will present a summary of results and conclusions.

2 Parallel Discrete-Event Simulation

This section presents an extension to the Chandy-Misra-Bryant (CMB) [5][6][15] algorithm in synchronous (clocked) systems based on the concept of *barriers*. The resulting deadlock-free parallel discrete-event algorithm is the synchronization and scheduling backbone of our parallel simulation paradigm which allows mixed-mode co-simulation.

2.1 Basic Concepts and Related Work

In the CMB algorithm and its variants, the system under simulation is modeled as a set of communicating processes that exchange information through messages and have distributed local times. The messages sent between processes consist of an event and the time of its occurrence. Scheduling policies can be divided into two major groups, namely the *conservative* scheduling and the *optimistic* scheduling. A survey of this work is described by Fujimoto [9].

In the CMB algorithm and its variants, *conservative* scheduling is used. Specifically, the local time of a process can never exceed the minimum time of its input events. Deadlock can occur if there are cyclic dependencies between processes. The basic proposed solu-

1. *Lookahead* is the property where a simulator can deduce future output events solely from the simulator's state history. Strictly speaking, every simulator has the ***lookahead*** property. Its outputs will remain constant by at least the smallest simulation time increment. However, using this small increment to advance the simulation time is usually too inefficient.

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

tions to this problem are deadlock avoidance [6] and deadlock detection and recovery [7]. In deadlock avoidance, null messages are introduced with the purpose of advancing the output simulation time whenever the local time is advanced without output change. This strategy introduces overhead on message traffic. Deadlock recovery is based on a two-phase scheme in which simulation proceeds until it is blocked. When blocking is detected, a recovery phase resolves the deadlock. The performance of this solution is compromised by the detection and recovery overhead, specially in the case of message passing machines. Another strategy is to estimate the earliest possible time of the next event, as implemented in YADDES [8].

Optimistic scheduling is used in the Timewarp [11] algorithm. Such an algorithm allows the local time of a given process to be increased beyond the local time of its input processes. In so doing, it is possible to anticipate additional events. If events are rare, the speculations may often be correct. If the prediction proves to be wrong, the process would have to return to its previous correct state. Such a reversal is disadvantageous because it may require a large amount of memory for storing history events and enabling rollbacks. Because we recognize that existing sequential simulators usually do not have backtracking capability, optimistic scheduling will be difficult to implement, and therefore is not considered further.

2.2 Synchronous Feedback Simulation Mechanism

The majority of the VSLI systems designed today are clocked systems. By explicitly defining synchronization points in a feedback structure, it is possible to facilitate parallel simulation in a simpler and more efficient way. It should be pointed out that feedbacks inside a simulation instantiation are not a concern because they can be easily handled by sequential simulators.

The problematic deadlock arises when a feedback cycle exists between simulation instantiations. Within each of these feedback cycles, a restriction is made which requires at least one latch or edge-triggered flip-flop in the loop to break the feedback cyclic operation during one pass of simulation. In this paper, latches or edge-triggered flip-flops will be called registers. A register is controlled by its input clock. When the clock is *active*, it *opens* the register, during which time the register's outputs depend on the values of its inputs. The clock activation can be a transition, as in the case of an edge-triggered flip-flop, or a level, as in the case of a latch. A register is *closed* when its clock is *inactive*, during which time the register inputs do not affect its outputs.

The feedback resolution mechanism relies on using a *barrier*, which is simulated according to the value of the associated register state as well as the type of registers used in the design. Its main purpose is to resolve feedback deadlocks, guaranteeing simulation progress by generating output events during an interval of one clock period. To do so, when a barrier is closed by its clock, its input values are kept constant. The barrier may receive computed results from the feedback to update its input values. When the barrier is opened by its clock, its output values may be affected by its inputs.

Because we are restricted to an interface in which backtracking is not allowed, the simulation cannot be rolled back one clock period and resimulated with the updated inputs. To achieve input updating without rollback, time is allowed to proceed for the interval of one inactive clock interval as if time were stretched. This clock interval was simulated during the phase in which the barrier was closed and inputs kept constant. In addition, the barrier should also correctly handle external simulated times by simple bookkeeping. Although the output values of a barrier are the same as its associated register outputs, the internal values may not be the same due to clock stretching. To obtain the correct internal values, the register needs to be duplicated into a *shadow barrier.*

2.3 Barrier and Shadow Barrier

To illustrate the feedback breaking mechanism with a barrier, Figure 1 shows two processes that comprise a communication cycle, in which one is simulating a register *reg* and the other is a generic process P. To break the feedback cycle, the first step is the identification of the feedback register *reg*, which is replaced with a barrier B process. This process has explicit control of its input and output communication.

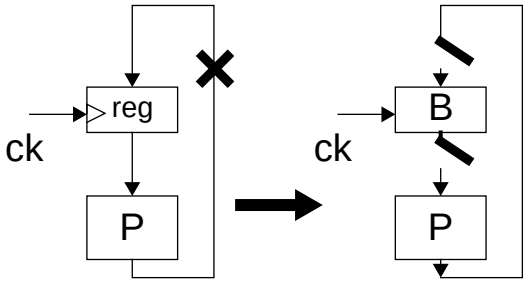

Figure 1. Feedback-breaking transformation with a barrier.

Figure 2 shows how a barrier can effectively break the feedback cycle and update its inputs without having to rollback. In the output phase, the barrier B has its input connection open and its output connection closed. Because of the semantic of the barrier, the inputs will not affect the outputs until the next clock event. During the clock interval, the process topology is acyclic, from B to P, and could be simulated with an acyclic algorithm [13].

In the input phase of barrier B, the outputs of process P are taken into account. With output opened, barrier B uses the values received from the process P to update its inputs and internal states for the next cycle. Barrier B is simulated with the updated inputs computed in the previous output phase. One direct consequence is that the inputs and outputs of barrier B are now offset by one clock interval. In the case of outputs, this offset can be easily cancelled by the barrier. However, the offset in internal values is unknown, and therefore needs special attention.

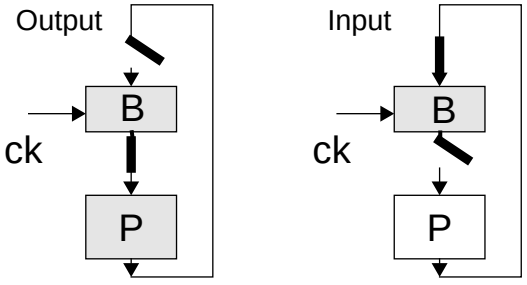

Figure 2. Barrier phases of operation.

To compute the correct internal values, the functionality of the register is duplicated in a shadow barrier, with only the input connections. This duplicated register is simulated as a regular process with an acyclic algorithm. Figure 2 illustrates the duplication of reg process onto a shadow barrier process SB.

Because the inputs are kept constant during the output phase, if there is a hold time violation in the real circuit, the violation will occur in the barrier. The assumed transition will be clocked in the

register simulated by the barrier B, but not in the real circuit. However, this would be an incorrect simulation.

Also, during the input phase we stretch the local simulated time by one clock interval. As a consequence, if there is a propagation time violation, which means the clock period is smaller than the propagation time of the register, the assumed transition may happen in the barrier. This would also create an incorrect simulation.

Hold time and propagation time violations are detected by comparing the outputs of the barrier B with the outputs of the shadow barrier SB. Violations are reported to the user.

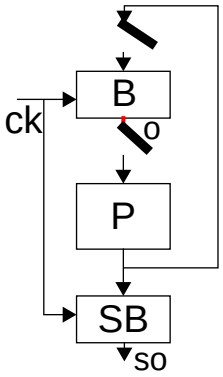

Figure 3. Barrier duplication into a shadow barrier.

2.4 Barrier Algorithm

The barrier algorithm is listed in Figure 4. It augments an acyclic simulation algorithm with the simulation of barriers and shadow barriers. When a barrier is closed (clock is not active), it executes an output phase to break a feedback. This is followed by an input feedback update phase. If the barrier is open (clock is active) during an interval, as in the case of a transparent latch, it will be simulated like any other feedforward process with an acyclic algorithm. In this case, using an acyclic algorithm to avoid deadlock requires another latch in the feedback cycle which is closed while the original latch remains open. That is the common case of a two-phase clock scheme.

```
Loop {
    Receive inputs until clock event
    if Clock is active {
        Acyclic simulation up to clock event time
        }
        else {
        Output:
            Open barrier inputs and close outputs
            Acyclic simulation up to clock event time
        Input:
            Close barrier inputs and open outputs
            Acyclic simulation up to clock event time
            Compare shadow barrier and barrier outputs
}
```

Figure 4. Barrier algorithm.

The cause of communication overhead introduced by simulating a register with the barrier algorithm is due to the fact that in a clock interval the acyclic algorithm is executed three times, twice in a barrier and once in a shadow barrier. Furthermore, there is additional communication between the barrier and shadow barrier for output comparisons.

2.5 Circuit Pipeline Exploitation

It is possible to increase simulation concurrency in a feedback circuit by using more barriers than needed to avoid deadlock. As in the case of circuit pipeline, there can be more than one register in a circuit feedback. By simulating the registers with the barrier algorithm, it is possible to take advantage of the circuit pipeline concurrency during simulation.

Figure 5 illustrates a circuit with more than one register in the feedback cycle. Suppose the circuit is partitioned in parts A and B for parallel simulation. With the barrier algorithm, at least one register should be simulated as a barrier to avoid deadlock. Let the register in part A be simulated by the barrier algorithm. Because of the event synchronization effect of the barrier on part A and input communication dependency of part A on part B, the simulation would be executed sequentially. If the register in part B is also simulated as a barrier, both parts can be executed in parallel during one clock cycle. This reflects the actual circuit independency when the registers do not change their output values.

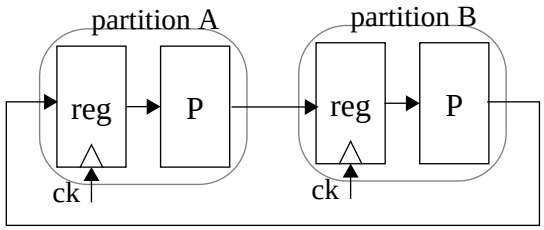

Figure 5. Two barrier case in a feedback cycle.

The actual speedup will depend on the trade-off between the increase in concurrency and the increase in computation due to the barriers and shadow barriers. If the register computation is small in comparison with its associated section P, the increase in speedup can be substantial.

3 Software Backplane

This section presents an extensible simulator paradigm suitable for parallel mixed-mode co-simulation. Distributed conservative event-driven scheduling is used in combination with the barrier algorithm for synchronous feedback systems presented in the previous section. In the following subsections, we will review some background in sequential mixed-mode simulation and co-simulation. Then we will present our co-simulation method, followed by a description of an implementation and the integration of THOR, IRSIM, and SPICE.

3.1 Related Work

In the area of sequential multi-level mixed-mode simulation [17], several simulators have been developed. iSPLICE3[1] is a classic example, combining circuit-level, switch-level, and logic-level simulation modes. iMACSIM [19] has behavioral, functional and electrical levels, and provides a mixed-mode domain (the s-domain and z-domain). Another example, LSIM [24], typifies current commercial use.

At any simulation level, there have been many parallel implementations. PARSWEC [23] is an example of a timing simulator that used optimistic scheduling in a CM5 multiprocessor. A parallel RSIM implementation was presented by Briner [4] using optimistic scheduling in a 16-processor BBN. PTHOR [20] is a parallel version of THOR that uses conservative scheduling and runs on shared-memory multiprocessors.

In the area of hardware/software co-design, research on co-simulation [3] [10] [12] [22] so far has not addressed an extensible parallel mixed-mode simulation backplane. We present Symphony which employs a barrier algorithm to simulate feedback systems with reasonable speedups on a multiprocessor machine.

3.2 Symphony

In this subsection, we describe a framework to integrate existing simulators on parallel machines. The framework leverages on the barrier algorithm for synchronization and scheduling. The actual simulator integration is accomplished by linking the object code of existing simulators to the Symphony software backplane. This requires a few specific procedure name calls. Load balancing for different simulation granularity is addressed by local interleaving which requires the interface calls to be reentrant. Integrating well-known sequential simulators with small changes, and porting the parallel co-simulator to three different machine platforms were easily accomplished with the Symphony implementation.

We want to simulate a complex system that is decomposed into coarse-grain blocks. The partition of these blocks is based in the simulation level and mode of interest. The blocks are further partitioned in order to expose existing parallelism and speedup the simulation.

The simulation framework consists of a distributed kernel that has a code section attached to each instance of a sequential simulator. The kernel is responsible for event communication, event synchronization, scheduling, general management of sequential simulators, and gathering of simulation results. The integration of existing simulators to the kernel is accomplished through a few kernel entry points. These entry points consist of procedure calls that should exist in the source code of the simulator to be integrated. The procedure names should be available to the kernel in its internal table and the object code of the simulator should also be available for linkage with the kernel. The kernel must also have a set of procedure calls to implement analog/digital convertion. This conversion has already been explored and resolved [17].

The kernel schedules a given simulator instance and synchronizes it with other instantiations. The kernel also schedules the events with their associated simulators through the inputs that it receives from other simulators and local simulated times. Scheduling and synchronization are accomplished through the use of the previously described parallel discrete-event simulation algorithm based on barriers. The kernel exchanges information through messages that have time-stamped events. It also executes a basic acyclic algorithm [23], except in the case of a register in a cycle, where the kernel executes the barrier algorithm.

3.2.1 Implementation

Symphony kernel was written in C++. The main data structures used by each local instantiation of the kernel for scheduling are input events and sorted time queues. An input event, either local or from communicating with other simulators, and the input event time are stored in the corresponding input queue. The input queues implement a form of time wheel optimized to this particular case. The input event time also updates a sorted time queue to detect the minimum time increment of all the input connections and local inputs. The simulator receives input events as well as a stop time when there is an interval to simulate. In Symphony, we implemented very simple, but fast analog/digital conversations. The user could trade-off conversion accuracy with speed by duplicating interface fan-in or fan-out netlist.

Because of static assignment of instantiations to processors and the potential small granularity of duplicated shadow barriers, interleaving of sequential simulators in a given processor was implemented to improve the load balance. This feature also helps load balancing in feedforward systems. If multi-threading was available in a system, these tasks could be implemented as threads. We did not use threads because of portability issues and the fact that, at the time we initiated this work, not all parallel processing machines were multi-threading systems. Scheduling of instantiations is programmed to favor the process that is farthest behind in its simulated time with a task queue. The main difficulty of interleaving is that it requires sequential simulators to have reentrant code at the interface entry points.

3.2.2 Integrating and Porting with Symphony

The simulators integrated into the prototype were chosen to cover a reasonable spectrum, ranging from the behavioral- and structural-levels, to the switch- and circuit-levels. At the circuit-level, SPICE3 [21] (or its variant hSPICE) was selected because it represents probably the most widely used simulator. The switch-level simulator IRSIM [18] is a logic mode switch-level simulator with timing information, and is widely used in universities. As a representative of behavioral and functional-level simulators we used THOR [2]. The selection of these simulators was also influenced by the fact that their source codes are available in the public domain.

As the three simulators used in our parallel simulator were written in C, our solution was to convert the C code to C++ by trivially packaging together local static variables with its associated functions. Reetrant code was achieved in SPICE with the convertion of a relatively few SPICE files to C++, leaving the majority intact in C. IRSIM and THOR convertion from C to C++ produced an execution overhead of less then 5%. The overhead in SPICE is negligible because most of its execution time is spent in the device formulation and matrix inversion code.

The primary target of our implementation is a parallel machine. The kernel has a software layer that isolates the architectural primitives, which makes it easy to port to different parallel-processor platforms. We have implemented Symphony on an Intel iPSC/860 message passing machine, a shared-memory multiprocessor DASH [14], and single-processor DECstations. Although there is no speedup in one processor, tremendous savings in simulation time can be gained through multi-level mixed-mode tradeoffs.

4 Performance

In this section we analyze Symphony's performance through a suit of VLSI circuit designs. We ran our simulation on two multiprocessor platforms: the Stanford shared-memory DASH and the Intel iPSC/860. DASH has a cache-coherent non-uniform-memory-access architecture, with 32 processors organized into 8 clusters of 4 processors each. Each processor is a 33 MHz MIPS R3000, and has a 64 KB first-level cache and a 256 KB second-level cache. Each cluster has 28 MB of main memory. A directory-based protocol is used to maintain cache coherence across the 8 clusters. The iPSC/860 multi-processor also has 32 processors. Each processor is a 40 MHz Intel i860 XR with 16 MB of memory. The interconnection network uses a hypercube topology.

In the benchmark circuit selection, we looked for circuit sizes in which the sequential case would fit in memory and execute on both the multiprocessor platforms. Although the resulting circuits are relatively small, they had enough coarse grain parallelism for a 32-multiprocessor machine. The circuits were partitioned manually for the 32-processor case. For bigger circuits and large-size multiprocessor machines the partition procedure needs to be automated.

The circuits consist of a combinational design to stress upper speedup limits, a sequential design to stress the shadow barrier algorithm overhead, and a mixed-signal design. All three cases had their netlists extracted from a CMOS mask layout. All speedups were calculated over the best sequential version of each program executing on the two multiprocessors.

4.1 Acyclic Case

The first example is an acyclic circuit that implements a relatively large feedforward FIR filter. It consists of sixteen 32-bit registers and multipliers connected to a binary tree of fifteen 32-bit adders. This case is a combinational circuit with simulation dominated by IRSIM time, in addition to four small SPICE runs. The simulation time attributed to THOR is negligible.

The adder tree circuit was extracted from an actual CMOS layout with 33,900 transistors and 15,723 nodes. Four small netlists were duplicated from the adder tree to simulate in SPICE. The entire adder tree is simulated in IRSIM. The registers and multipliers were simulated in THOR, having a C code for each block. We simulated the circuit for one thousand clock cycles in order for the execution to take approximately one hour in one DASH processor. For multiprocessor runs, the adder tree netlist was partitioned to have roughly 2 partitions per processor when using 32 processors.

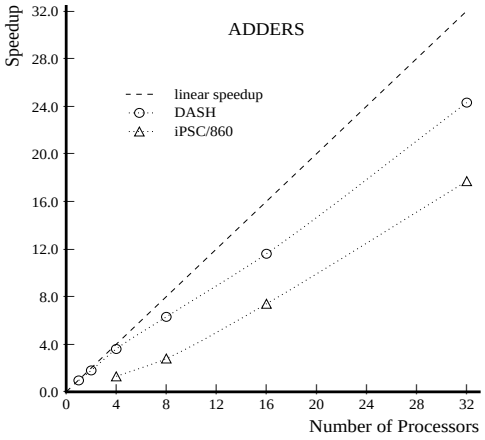

Figure 6. Adder tree circuit simulation performance.

As shown in Figure 6, using 32 processors on the Intel iPSC/860 the speedup is 16; it is nearly 25 on DASH. The DASH implementation benefits from more concurrency because the communicating events are stored directly into the kernel data structures while an instantiation is being executed. In addition, the speedup on the iPSC/860 implementation is hampered by the blocking of output data being sent to other processors, a side effect of our interleaving implementation on a message-passing machine. The multiprocessor runs with one and two processors on the iPSC/860 did not fit in the memory. There is a difference of less then 5% in execution time between the sequential and the parallel (interleaved) execution in one DASH processor. The interleaving and number of communication messages incurred small overhead in the simulation.

4.2 Feedback Case

To test our feedback mechanism performance, the second example uses a sequential circuit implementing a feedback pipeline filter. The simulation has a THOR section to generate data inputs, and an IRSIM section for timing simulation. The circuit consists of thirty-one 31-bit registers and sixteen 32-bit adders in a total of 67,904 transistors. We simulated the circuit for four hundred and thirty clock cycles in order for the execution to take approximately one hour in one DASH processor. For multiprocessor runs, the circuit netlist was partitioned to have roughly 2 partitions per processor when using 32 processors. To avoid deadlock with the feedback, at least one register has to be executed with the barrier algorithm. In this circuit, a register was implemented as fully static edge-triggered set/reset flip-flops. The register netlist duplication into a shadow barrier represents an overhead of 1.8% per register.

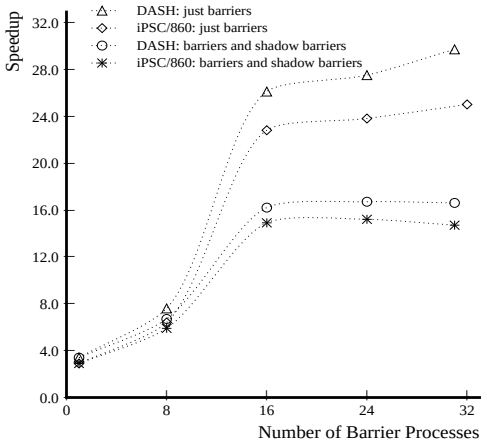

Figure 7. Effect of the number of barriers on speedup performance for the feedback circuit with 32 processors.

First we look at the effect of simulating registers with the barrier algorithm in the case of 32 processors. Two cases were simulated. In the first case, we have a shadow barrier for each barrier introduced. In the second case, we simulate the same circuit, but without the shadow barriers. In both cases, the speedup measures the increase in concurrency with the number of barriers used. It also allows the measurement of the impact of duplication caused by these overhead register barriers.

As shown in Figure 7, the introduction of 16 barriers capture most of the natural pipeline concurrency of the circuit. The 29% increase in netlist overhead corresponds to a 40% degradation in performance. With more than 16 barriers and no shadow barriers, there is a small increase in concurrency. But with barriers and shadow barriers there is small change in the speedup due to the combined increase in concurrency and netlist overhead. The best measured case is with 24 barriers. With less than 16 barriers, the increase in the netlist duplication is almost hidden by the increase in concurrency.

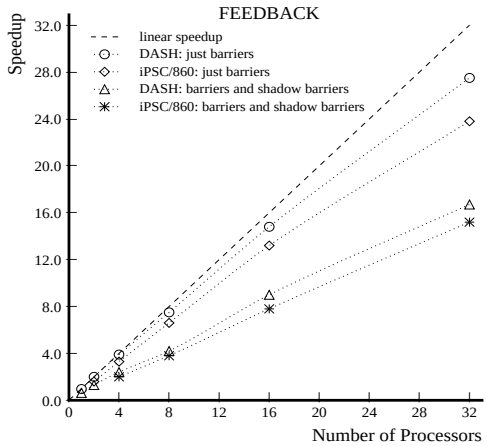

Figure 8. Feedback circuit parallel simulation performance.

Next, we look at the simulation with 24 barriers. In this case, there is a 43% increase in netlist overhead with shadow barriers. The

speedups on 32 processors for the two cases are 28 and 17 respectively when running on DASH, and 24 and 15 respectively when running on the iPSC/860, as shown in Figure 8. As in the acyclic case, the DASH implementation benefits from more concurrency but the differences are not as large. This is a result of using 24 barriers to expose the circuit concurrency. The multiprocessor runs in the iPSC/860 did not fit in the memory of one processor. In the case of shadow barriers it also did not fit with 2 processors. There is a difference of less then 4% between the sequential and the parallel execution in DASH with only barriers in one processor.

4.3 Mixed-Signal Case

The third example is a mixed-signal circuit, implementing a 7-bit CMOS flash A/D converter [16]. The analog portion of the converter consists of 128 comparators. Each comparator uses a differential preamplifier followed by a drain-strobed latch and a completion detection circuit. The circuit has a total of 11,042 transistors. Because of little coupling between comparators, we used multiple instantiations to simulate separately the 128 comparators in SPICE, dominating the total simulation time. The digital part of the circuit is simulated in IRSIM. We ramp up the analog input to generate the complete 128 7-bit outputs. This execution takes approximately 10 1/2 hours in one DASH processor.

The speedup performance is shown in Figure 9. As in the previous examples, the DASH implementation shows more concurrency in parallel simulation, and the multiprocessor runs in the iPSC/860 did not fit in the memory with one processor. The parallel execution in one DASH processor is 20% faster than the sequential execution. This is a consequence of smaller partitioned netlists which results in fewer time steps in the parallel execution.

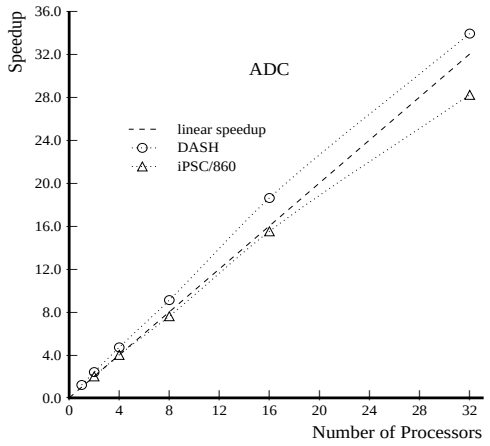

Figure 9. ADC circuit parallel simulation performance.

5 Conclusions

In this paper, we presented a new parallel discrete-event simulation based on the barrier algorithm. Using this algorithm, there is little communication overhead and the barrier simulation avoids state rollbacks. Through the use of barriers it was possible to increase simulation concurrency by taking advantage of the circuit pipeline.

We applied the barrier algorithm to an extensible simulator integration paradigm suitable for parallel multi-level mixed-mode co-simulation. The feasibility of these concepts was demonstrated with a simulation backplane called Symphony, and the integration of SPICE3, IRSIM and THOR to Symphony. This resulting parallel co-simulator was capable of concurrent multiple circuit-level, switch-level, gate-level, RTL-level and behavioral-level simulations. Symphony has been ported on an iPSC/860 message-passing machine and the DASH shared-memory multiprocessor. The three benchmark circuits used in this paper showed good parallel performance with medium-size multiprocessor machines and little integration overhead. Our work suggests that parallel multi-level co-simulation is a viable means to efficiently verify large-scale mixed-mode systems on general-purpose multiprocessor machines.

References

[1] E. L. Acuna, et al., “Simulation Techniques for Mixed- Analog/Digital Circuits”, *IEEE Journal of Solid-State Circuits*, April 1990.

[2] R. Alverson, et al., *THOR user’s manual: Tutorial and commands*. Technical Report CSL-TR-88-348, Stanford, January 1988.

[3] D. Becker, et al., “An Engineering Environment for Hardware/Software Co-Simulation”, *Proc. of DAC*, 129-134, June 1992.

[4] J. V. Briner Jr., et al., “Breaking the Barrier of Parallel Simulation of Digital Systems”, *Proc. of DAC*, 223-226, June 1991.

[5] R.E. Bryant, “Simulation of Packet Communication Architecture Computer Systems”, *Technical Report MIT-LCS-TR-188*, MIT, July 1977.

[6] K.M. Chandy and J. Misra, “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs”, *IEEE Transactions of Software Engineering*, SE-5(5):440-452, September 1979.

[7] K.M. Chandy and J. Misra, “Asynchronous Distributed Simulation via a Sequence of Parallel Computations”, *Communications of the ACM*, 24(11):198-206, April 1981.

[8] E. DeBenedictis, et at., “A Novel Algorithm for Discrete-Event Simulation”, *IEEE Computer*, 24(6):21-33, June 1991.

[9] R.M. Fujimoto, “Parallel Discrete Event Simulation”, *Communication of the ACM*, 33(10):30-53, October 1990.

[10] R.K. Gupta, “Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components”, *Proc. of DAC*, 225-230, June 1992.

[11] D.R. Jefferson, “Virtual Time”, *ACM Transactions on Programming Languages and Systems*, 7(3):404-425, July 1985.

[12] A. Kalavade and E.A. Lee, “A Hardware/Software Codesign Methodology for DSP Applications”, *IEEE Design and Test of Computers*, 16-28, 1993.

[13] D. Kumar, “Systems with Low Distributed Simulation Overhead”, *IEEE Transactions on Parallel and Distributed Systems*, 3(2): 155-165, March 1992.

[14] D. Lenoski et al. “The DASH Prototype: Implementation and Performance”, *Proc. of ISCA*, May 1992.

[15] J. Misra, “Distributed Discrete-Event Simulation”, *ACM Computing Surveys*, 18(1), 39-65, March 1986.

[16] C. L. Portmann and T. Meng, “Power-efficient Metastability Error Reduction in CMOS Flash A/D Converters”, to appear in *IEEE Journal of Solid-States Circuits*, July 1996.

[17] R.A. Saleh and A.R. Newton. *Mixed-Mode Simulation*, Kluwer Academic Publishers, 1990.

[18] A. Salz and M. Horowitz, “Irsim: An Incremental MOS Switch-Level Simulator”, *Proc. of DAC*, 173-178, June 1989.

[19] J. Singh and R. Saleh. “iMACSIM: A Program for Multi-level Analog Circuit Simulation”, *Proc. of ICCAD*, 16-19, 1991.

[20] L. P. Soule, “Parallel logic simulation: an evaluation of centralized-time and distributed-time algorithms”, *Technical Report CSL-TR-92-527*, Stanford, June 1992.

[21] R. J. Spier. “A preliminary evaluation of the SPICE3 simulation package”, *IECEC*, vol. 1, 189-194, 1991.

[22] D.E. Thomas, et al., “A Model and Methodology for Hardware-Software Codesign”, *IEEE Design and Test of Computers*, 6-15, 1993.

[23] C.-P. Wen and K. Yelick. “Parallel Timing Simulation on a Distributed Memory Multiprocessor”, *Proc. of ICCAD*, 130-105, 1993.

[24] LSIM Manuals, Mentor Graphics.