Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

Compiled Code in Distributed Logic Simulation

Jun Wang and Carl Tropper

School of Computer Science
McGill University
Montreal, Quebec, Canada
jwang90, carl@cs.mcgill.ca

ABSTRACT

A logic simulation approach known as compiled-code event-driven simulation was developed in the past for sequential logic simulation. It improves simulation performance by reducing the logic evaluation and propagation time. In this paper we describe the application of this approach to distributed logic simulation. Our experimental results show that using compiled code can greatly improve the stability and overall performance of a Time-Warp based logic simulator. We also present a technique called fanout aggregation that makes use of information on circuit partitions and considerably improves the run-time performance of our (distributed) compiled code simulator. It does not produce a similar improvement when used in conjunction with an interpreted simulator because of run-time overhead.

1 INTRODUCTION

Logic simulation plays a very important role in the digital integrated circuit (IC) design process. Historically, two primary classes of logic simulators have been developed: compiled-code simulators that create a computer program for each specific circuit, and event-driven simulators that are based on the general event-driven simulation paradigm. While the former eliminates the need for a central scheduler, the latter has the advantage of processing only the circuit elements that have a state change.

Compiled-code event-driven simulation (Lewis 1991)(Wang and Maurer 1990)(Au, Weise, and Seligman 1991) is a hybrid approach that attempts to combine the advantages of both compiled-code and event-driven simulation. Since circuit structure is known at compile-time, compiled-code can be generated to represent the structure so as to reduce logic evaluation and propagation time. At the same time, it uses an event-driven simulation kernel to avoid evaluating all circuit elements indiscriminately.

As circuit size grows, logic simulation has become a bottle-neck in the IC design process. One approach to speeding up the simulation task is distributed logic simulation (Chamberlain 1995), in which the simulation is executed on multiple CPUs. Synchronization and communication among the processes involved in a distributed simulation involves the sending of time-stamped messages between the CPUs executing the simulation. Therefore, lowering the message overhead is very important for better performance. Furthermore, in a Time-Warp based distributed simulation (Jefferson 1985), reducing the number of rollbacks plays a critical role in determining the performance of the simulator.

In this paper, we discuss the idea of using compiled code in distributed event-driven logic simulation. As with its sequential counterpart, we attempt to improve the simulation performance by creating a compiled-code program for the circuit structure, separate from the event-driven simulation engine. Furthermore, while generating code, we can make use of information available at compile-time to optimize the generated code. In particular, we describe a technique called fanout aggregation that combines messages sent from a logic gate to another CPU, thereby reducing message and scheduling overhead. This technique can also be used in an interpreted simulator. However, it has a run time overhead associated with it since the aggregation is performed at run-time by checking the partition of the fan-outs and then aggregating them, thereby producing mixed results in our experiments. Used with compiled code, the technique incurs no run-time overhead and works effectively.

The rest of the paper is organized as follows. Section two gives a brief overview of compiled-code event-driven logic simulation. In section three we describe the utilization of compiled code in distributed event-driven logic simulation, and the optimization technique fanout aggregation. In section four we present test results on some benchmark circuits and our analysis of the results. Finally, section five contains our conclusions.

2 BACKGROUND

From the algorithmic point of view, logic simulators can be broadly classified into two categories: oblivious and event-driven. In oblivious simulation, for each input vector, all the logic elements in the circuit are evaluated, regardless of whether any changes have occurred on their inputs. For good performance, the circuit under simulation is usually translated directly into a straight line of compiled-code that evaluates the logic gates. This is possible because of the oblivious nature of the simulation. To ensure correctness, gates are levelized(Chiang and Palkovic 1986)(Wang and Maurer 1990)(Wang, Hoover, Porter, and Zasio 1987) such that before a gate is evaluated, all its fan-ins would have been evaluated. Simulation is performed by executing the generated program directly and no external simulation engine is required. The advantage of compiled oblivious simulation is that logic evaluation and propagation operations are performed extremely fast. The generated code usually contains very few conditionals and no loops. Also, it is very efficient for repeated simulation runs. The disadvantage is that a lot of useless work is done for gates whose state is not changed. And because scheduling is implicitly implemented in the generated code, compiled-code based simulations are limited to synchronous circuits with zero-delay or unit-delay, although some researchers have tried to extend it to arbitrary delay models(Shriver and Sakallah 1992)(Lee and Maurer 1996).

Event-driven simulation, on the other hand, utilizes a central event queue and processes only the state changes occurring in the circuit. This usually implies an interpretive implementation where the simulator creates internal data structures to represent the circuit, and logic evaluation and propagation are performed on these data structures. The advantage of event-driven logic simulation is that it only processes the activities in the circuit. Another advantage is that it is capable of handling all circuit models (synchronous or asynchronous) and timing models (zero-delay, unit-delay, or arbitrary delay). The disadvantage is the scheduling overhead and the fact that it takes longer to evaluate and propagate logic values due to the need to traverse the data structures.

Compiled-code event-driven simulation attempts to take advantage of the strengths of both approaches. The simulation is event-driven in nature, but the circuit structure is translated into compiled-code. However, the compiled-code is not a straight line of code as in oblivious simulation. Instead, it is a collection of chunks of code with each chunk performing the evaluation or propagation for one gate. Each chunk can be implemented as a procedure or simply distinguished by a leading label. The simulation kernel can also be greatly simplified (Lewis 1991)(Wang and Maurer 1990) to be a central dispatcher. The simulation is performed by jumping back and forth between the chunks of code and the dispatcher as directed by the generated events. Figure 1 is an example modified from (Lewis 1991) that shows a simple circuit and a chunk of code to propagate the output of gate A.

The distinction between compiled-code event-driven simulation and purely event-driven simulation can also be described in terms of partial evaluation(Au, Weise, and Seligman 1991), a technique that turns a generic program into a specialized program by combining the generic program with some known constant inputs. The compiled code of the circuit structure is the constant data, and together with the simulation engine forms a specialized event-driven simulation program.

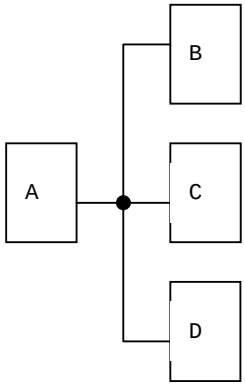
Among the major advantages of compiled event-driven simulation are:

- Fast logic value propagation.
- Efficiency for repeated simulation runs.
- Possibilities of compile-time optimizations.
- Simplified scheduler with less scheduling overhead.

3 DISTRIBUTED COMPILED-CODE EVENT-DRIVEN SIMULATION

3.1 Distributed Logic Simulation

As circuit complexity grows steadily, researchers have been looking at parallel and distributed simulation as a way to speed up simulation of digital ICs. In distributed logic simulation, multiple CPUs are utilized. The circuit is first partitioned into a number of parts and each part is assigned to one of the CPUs, which then carry out the simulation in parallel in an attempt to shorten simulation time. Communication among the CPUs is achieved by exchanging timestamped messages. In order to ensure simulation correctness, synchronization protocols are employed. Two major class of synchronization protocols have been developed: conservative and optimistic. Conservative protocols(Chandy and Misra 1981) achieve synchronization by making use of a blocking protocol, while in an optimistic protocol such as Time Warp(Jefferson 1985), a logical process (LP) proceeds without any concern for other LPs until a message in its past (a straggler message) is received, at which point the LP "rolls back" to a (simulated) time prior to the time of the straggler message. Messages which were sent to other LPs after the straggler are canceled by sending *anti-messages*.




```
fanout_A:
  node_val[A] = next_node_val[A];
  node_active[A] = FALSE;
  if(!gate_active[B]) {
     gate_active[B] = TRUE;
     *gate_active_ptr++ = &simulate_B;
  }
  if(!gate_active[C]) {
     gate_active[C] = TRUE;
     *gate_active_ptr++ = &simulate_C;
  }
  if(!gate_active[D]) {
     gate_active[D] = TRUE;
     *gate_active_ptr++ = &simulate_D;
  }
```

Figure 1: Example circuit and propagation code for gate A.

To date, distributed logic simulators have been implemented as event-driven simulators. As mentioned above, this also means the simulator is interpretive. Figure 2 shows the structure of a distributed interpretive event-driven logic simulator.

3.2 Compiled Code in Distributed Logic Simulation

As an alternative, we create a distributed compiled-code event-driven simulator in which the event-driven simulation kernel is retained, but the data structures representing the circuit to be simulated are created as separate programs. First, we partition the circuit into desired number of parts. Then for each part, we create a C file that contains code for the logic propagation of each gate belonging to that part. The C files are then compiled into shared objects and dynamically loaded and linked to the simulation kernel at run-time, as shown in Figure 3. The overhead of compiling the C files can be greatly reduced by compiling them in parallel, since they are completely independent of one another (each file corresponds to a partition).

It should be noted that translating a circuit description into C code works not only for a structural description, but also for a behavioral description which is common in modern hardware description languages (HDL) such as Verilog (IEEE 2001). In interpretive simulators, a behavioral description is usually translated into internal data structures and executed in an interpretive way. With compiled code, a behavioral description can be directly translated into C code, which is more efficient at run-time than interpreted code. In addition, the C code can be further optimized with an optimizing C compiler.

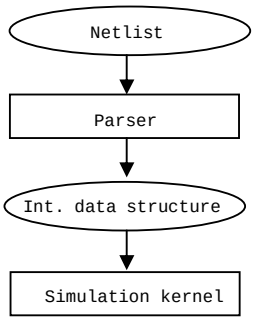


Figure 2: Typical interpretive event-driven simulator.

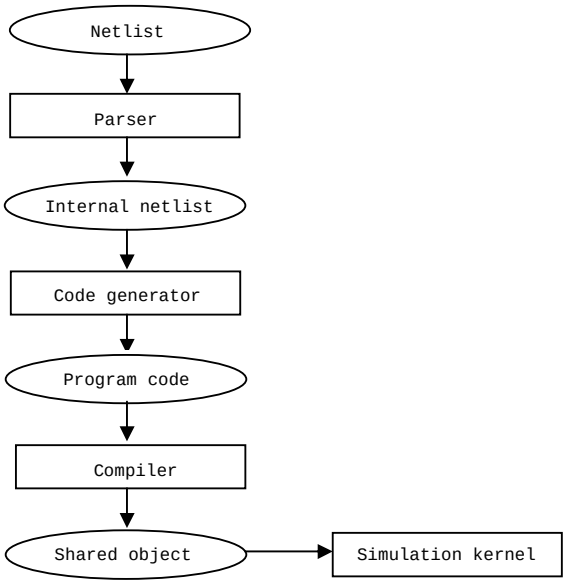


Figure 3: Compiled-code event-driven simulator.

As pointed out above, a direct effect of compiled code is more efficient processing of logic propagation. Furthermore, it also allows compile-time optimization techniques to be easily applied. For example, since the circuit structure is known at compile-time, all of the propagation loops are unrolled. Other possible compile-time optimizations include:

1. Elimination of buffers and inverters, which essentially reduces the circuit size(Maurer 1997).
2. Gate grouping, which reduces run-time scheduling overhead(Maurer 1997).
3. Gate levelization.
4. Identifying of gate trees or strongly-connected components.

One major difference from sequential compiled-code event-driven simulation is the necessity to do partitioning. With partitioning information available at compile-time, more optimizations can be performed. For example, when a fanout gate f of a gate g is in another part, we need to send a message to that part when the output of g is changed. With the part of f known at compile-time, we can generate code for g to directly send a message to that specific part, thus eliminating the need to check the part of f at run-time. Figure 4(a) shows code generated for gate A in the example circuit in Figure 1. Suppose gate B and C are in part 2 while gate A is in another part, when gate A propagates its value, the generated code sends messages directly to part 2 without any run-time checking.

3.3 Fanout Aggregation

A simple yet very effective optimization technique called fanout aggregation also makes use of partitioning information. Using the circuit in Figure 1 as an example, assume both B and C are in the same part, say part 2, while A is in another part. When the output of A changes, we send one message for each of B and C, as shown in Figure 4(a). The receiver of the messages will schedule one event for each message. Obviously, since B and C are in the same partition, we can combine the two messages into one, thereby reducing the number of messages. Reducing the number of messages causes a reduction of the probability of rollbacks in Time Warp. Furthermore, when an aggregation message is received, only one event needs to be scheduled. Thus, we also reduce the number of scheduled events.

Example code generated with fanout aggregation is shown in Figure 4(b). One aggregation message is sent for both B and C instead of two messages in Figure 4(a).

This technique can also be used in an interpreted simulator. At run-time, the partition information of the fan-outs are checked, and those belonging to the same partition are combined. This run-time overhead, however, would to a certain extent offsets the benefits. With compiled code, as demonstrated in Figure 4(b), the aggregation is done at compile-time, therefore, there is no run-time overhead.

```
void propagate(int index)
{
  switch(index) {
     case A:
       send_assign_msg(2, B, 1, val);
       send_assign_msg(2, C, 1, val);
       gates[D]→set(1, val);
     break;
     ...
  }
}
```

(a)

```
void propagate(int index)
{
  agg_item aggs[MAX_ITEMS];
  switch(index) {
     case A:
       aggs[0].index = B;
       aggs[0].pin = 1;
       aggs[1].index = C;
       aggs[1].pin = 1;
       send_agg_assign_msg(2, aggs, 2, val);
       gates[D]→set(1, val);
     break;
     ...
  }
}
```

(b)

Figure 4: (a)Code for propagation of gate A (b) Code with fanout aggregation.

4 EXPERIMENTAL RESULTS

We have implemented a distributed compiled-code event-driven logic simulator based on the Distributed Verilog Simulator (DVS) as described in (Li and Tropper 2003). A separate program accepts netlists as input, partitions the internal netlist, and produces C files. A command-line option directs whether or not the code generator should perform fanout aggregation. The C files are then compiled into shared objects and are loaded by the simulator proper at run-time.

We tested the simulator on some of the largest ISCAS-89 benchmark circuits(Brglez, D.Bryan, and Kozminski 1989). Each circuit was supplied with 100 random test vectors. Table 1 shows the number of gates, D flip-flops, and primary inputs and outputs. We compared the performance of the simulator to that of an interpreted simulator. The interpreted simulator differs from the compiled simulator only in that logic propagation is performed by traversing data structures. We made use of two versions of the interpreted simulator as well. One performed fanout aggregation at run time while the other did not.

All experiments were conducted on a four node network of AMD Athlon 64 computers running the Linux operating system. The computers were connected by a fast ethernet switch. MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/) was the underlying messaging system.

Table 1: Circuit profile.

Circuit	Gates	DFFs	Inputs	Outputs
s15850	9772	534	77	150
s35932	16065	1728	35	320
s38417	22179	1636	28	106
s38584	19253	1426	38	304

Table 2: Running time in seconds.

Circuits	iagg	cagg	reduction
s15850	13.75	12.02	12.6%
s35932	9.69	6.11	37.0%
s38417	20.85	15.86	24.0%
s38584	NA	20.29	NA

For each circuit, we performed partitioning with an implementation of the Fiduccia-Mattheyses algorithm (Fiduccia and Mattheyses 1982).

We choose to exhibit results for a two node network because neither version of the interpreted simulator could complete a simulation on four nodes. This was a consequence of the communication overhead of the fast ethernet relative to the CPU speed of the Athlon 64 CPU's. While the compiled code version did complete on four nodes, its execution times were larger than on two nodes for much the same reasons.

Tables 2- 5 depicts the running time, the total number of messages sent during the simulation, the total number of scheduled events and the total number of rollbacks, respectively for a two node network. The column labeled "iagg" contains results for the interpreted simulator with fanout aggregation while the "cagg" column contains results for the compiled simulator with fanout aggregation. A third column shows the reduction in percentage of each performance measure for "cagg" compared with the same measures for "iagg". Each entry in the tables is the average value of five simulation runs.

The interpreted version without fanout aggregation did not complete on two nodes. When fanout aggregation was added to the picture, simulations of all of the circuits with the exception of s38584 completed their executions.

These observations suggest that compiled code helps to stabilize optimistic logic simulators.

Comparing the results obtained for the interpreted and compiled code simulators with fanout aggregation, we see that the compiled code version results in far better simulation performance than the interpreted version. Fewer events are scheduled, fewer messages are sent, the number of rollbacks is significantly smaller and the execution time is much smaller. The results for s38417, for example, show 6.8% fewer scheduled events, 11.2% fewer messages, 27.4% fewer rollbacks and a running time which is 24.0% smaller.

Table 3: Messages sent during simulation.

Circuits	iagg	cagg	reduction
s15850	103501	93130	10.0%
s35932	42228	26469	37.4%
s38417	182598	162166	11.2%
s38584	NA	96687	NA

Table 4: Scheduled events.

Circuits	iagg	cagg	reduction
s15850	899124	839739	6.7%
s35932	876028	637484	27.3%
s38417	2006682	1870297	6.8%
s38584	NA	1207710	NA

Table 5: Number of rollbacks.

Circuits	iagg	cagg	reduction
s15850	4607	4059	11.9%
s35932	505	506	0.0%
s38417	3892	2827	27.4%
s38584	NA	2545	NA

The reason for the success of the compiled code simulation lies in its speeding up of logic evaluation and propagation. As a consequence messages are delivered more quickly between processors, resulting in a decreased number of rollbacks. Fanout aggregation causes events to be delivered between processors more quickly and also results in the scheduling of fewer events.

5 CONCLUSIONS

Combining compiled-code with event-driven logic simulation has the advantage of shortening the processing time for logic propagation events as well as opening the door to compile-time optimizations such as fanout aggregation.

In this paper we compared the performance of a distributed compiled code logic simulator to that of a distributed interpreted simulator with and without fanout aggregation. The compiled code simulator exhibited a

far superior performance to the interpreted simulator on a two node network with fanout aggregation. In fact, Its performance without fanout aggregation was either better then or at worst close to that of the interpreted one with fanout aggregation.

Perhaps more significant is the stability which compiled code lends to distributed optimistic logic simulation. The interpreted code simulator could not complete on 4 nodes with or without fanout aggregation and could not complete its execution on two nodes without fanout aggregation. In contrast, the compiled code simulator always completed its execution on either two or four nodes, with or without fanout aggregation.

Our future work will focus on reducing the overhead of the scheduler, as well as identifying new compile-time optimization techniques that can further improve simulation performance. We also plan to investigate the scalability of compiled code algorithms on larger circuits. The ISCAS circuits which we used to run the experiments described in this paper are the largest of the publicly available circuits.

REFERENCES

Au, W., D. Weise, and S. Seligman. 1991. Automatic generation of compiled simulation through program specialization. *Proceedings of the 28th ACM/IEEE Design Automation Conference*.

Brglez, F., D.Bryan, and K. Kozminski. 1989. Combinational profiles of sequential benchmark circuits. *Proceedings of IEEE Symposium on Circuits and Systems*.

Chamberlain, R. 1995. Parallel logic simulation of vlsi systems. *Proceedings of the 32nd ACM/IEEE Design Automation Conference*.

Chandy, K., and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM* 24 (11): 198–206.

Chiang, M., and R. Palkovic. 1986, March. Lcc simulators speed development of synchronous hardware. *Computer Design* 25 (5): 87–92.

Fiduccia, C., and R. Mattheyses. 1982. A linear-time heuristic for improving network partitions. *Proceedings of the 19th Design Automation Conference*.

IEEE 2001. *IEEE std. 1364-2001, IEEE standard verilog hardware description language*.

Jefferson, D. 1985, July. Virtual time. *ACM Transactions on Programming Languages and Systems* 7 (3): 404–425.

Lee, Y., and P. Maurer. 1996, Dec.. Bit-parallel multi-delay simulation. *IEEE Transactions on CAD of Integrated Circuits and Systems* 15 (12): 1547–1554.

Lewis, D. 1991, June. A hierarchical compiled code event-driven logic simulator. *IEEE Transactions on Computer-Aided Design* 10 (6): 726–737.

Li, L., and C. Tropper. 2003. Dvs: An object-oriented framework for distributed verilog simulation. *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*.

Maurer, P. 1997, July. The inversion algorithm for digital simulation. *IEEE Transactions on CAD of Integrated Circuits and Systems* 16 (7): 762–769.

Shriver, E., and K. Sakallah. 1992. Ravel: Assigned-delay compiled code logic simulation. *Proceedings of International Conference on Computer-Aided Design*.

Wang, L.-T., N. Hoover, E. Porter, and J. Zasio. 1987. Ssim: a software levelized compiled-code simulator. *Proceedings of the 24th ACM/IEEE Design Automation Conference*.

Wang, Z., and P. Maurer. 1990. Lecsim: A levelized event driven compiled logic simulator. *Proceedings of the 27th ACM/IEEE Design Automation Conference*.