A Case against Event-Driven Simulation for Digital System Design

Glenn Jennings
Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 Lund, Sweden
Tel. (+46)-46-1097 63, Fax (+46)-46-1047 14
Electronic Mail: glenn@dit.lth.se

Abstract

We argue that the electronic design community too often tries to apply discrete event simulation technology, such as VHDL, to applications for which it is ill-suited. In particular we consider the simulation of synchronous digital designs, and report two orders of magnitude speed improvement by using compiled simulation instead. The basic design of a simple compiled simulator is outlined.

1 Introduction

Discrete event-driven simulation is currently the design methodology a la mode in the digital electronic design community. This is due partly to the language VHDL [Ins88] [Sha86] and a recent Department of Defense requirement for documentation in the form of VHDL models.1 Several commercial products (not just those for VHDL) are based on event-driven simulation. However, we fear that the design community has lost its sense of balance, in that it reaches for event-driven simulation even when it is not appropriate. When designing synchronous systems, we believe that event-driven simulation can seldom be justified.

The alternative has acquired the name compiled simulation [WHPZ87] [SG87] [Lew89] [WM89] [MW90] and is a methodology in which no dynamic scheduling of the model components occurs during simulation. Instead, a static schedule is determined by a preprocessing step before simulation begins. The majority of all digital designs, especially in VLSI and for large systems, are synchronous (driven by global clocks). We have found that for the synchronous designs at this department [PAR89], compiled simulation is proving to be the superior simulation method.

2 A Case Study

In a recent (and fairly typical) scenario at our department, we were constructing a special-purpose microprocessor [R+90].2 After the architects had devised a circuit (a structure) which they believed functionally correct, the structure (at the register-transfer level) had to be validated by a lengthy simulation. This involved simulating the execution of hundreds of thousands of microprocessor instructions with this very detailed model.

We first created a model of the structure in VHDL. We took full advantage of VHDL's behavioral code, modeling the circuit as functional units. Our first disappointment was slow compilation time, approaching ten to fifteen hours for even small changes. Our second disappointment was the slow execution speed [LCAC88] of our model; we estimated five to six days for a full simulation. This was not workable, so we sought an alternative simulation methodology.

The critical insight was that VHDL was scheduling the

1MIL-STD-454L, paragraph 4.5.1

2We presently understand VERA to be the first example of a special-purpose vector processor, as well as the first example of a vector processor on a chip.
functional component models dynamically for every simulated instruction cycle. The overhead in event-driven simulators for managing the event queue and dispatching the models (dynamic scheduling) is well-known [Ram87] [WM90] and we believed we were paying this price dearly. We then observed the output from the VHDL simulation. It was showing us all the circuit activity from the moment a simulated clock signal changed state until the circuit became stable. We did not need to see all of these "glitches," but only the final stable circuit state. This was because, due to the methodology we use for building synchronous designs, we knew that the circuit would always reach a stable state. All we wanted to see was that stable, settled state of the model between clock pulses, and not all the jitter and noise that happens on all the wires before stability is reached. VHDL was spending its CPU resources producing information of no value to us.

Therefore we set out to see if the component models could be scheduled before simulation began, once and for all. This turned out to be a simple matter, and we modified our simulator to compute such a schedule automatically. The simulator input text was identical to the old VHDL-based code [Jen91] except that we replaced VHDL behavioral code with C behavioral code. There was only a cosmetic, syntactical difference between the two; that is, we made no change in the "abstraction level" at which the model was described. We were not too surprised when we found that compilation times went from fifteen hours down to three or four minutes, since VHDL compiler technology has not had much time to develop. But we were astounded to see about a forty-fold improvement in simulation speed! We could now complete our five-day simulation in only a few hours.

\[3\] Historical Background

The electronic community has a long tradition of using dynamically-scheduled simulators. Timing simulators [Nag75] [Nag80] such as SPICE numerically solve a large set of simultaneous differential equations at the circuit level and give very exact timing information. The size of the simulated time step is dynamically determined by a sort of global "vote" among the components at each simulation cycle; a disagreement means the entire simulation must back up and try again with a smaller time step. Such simulators can only be used for a few hundred transistors at best.

When the goal was to simulate the circuit's logical behavior, SPICE was spending too much time showing timing (a physical behavior). This led to switch-level simulators [Bry81] [Bry84] such as MOSSIM in which the component scheduling is determined each simulation cycle by forcing rooted paths into a switch network model, and resolving conflicts dynamically according to strength rules. This allowed simulation of tens of thousands of transistors, at the expense of detailed timing information.

Discrete event simulation (a methodology already known to the simulation community) then became popular [BP81] [CM81] [VL84] [DC+87] especially when coupled with behavioral modeling, as we see today with VHDL. It was now possible to model larger systems in the hundreds of thousands of transistors. Asynchronous designs posed no real problem to event-driven simulation. The design community was even further enticed (perhaps the word should be "seduced") by the apparent added gain that approximate timing information could still be obtained, since propagational delays could easily be represented as scheduling delays in the simulator's event-queue mechanism. We simply refer the reader to Chapter 12 of [Ins88] (the so-called "Semantics" of VHDL) to show how discrete-event simulation has been packaged for the electronics design community.

So we seem to have come full circle again. Instead of trying to use SPICE in applications where MOSSIM would have been more appropriate, we are now using VHDL and other event-driven simulators where compiled simulation is more appropriate. The error in both cases is roughly the same: trying to observe logical behavior with a simulator whose time semantics are geared to displaying physical behavior. It is not enough to say that we simply set all delays in our VHDL model to zero: the simulator's scheduling mechanism still grinds on its event queues in the same way, with delta-delays instead of nanosecond delays. It may be argued that it is only VHDL which "abuses" event-driven simulation, but the evidence of a number of other simulator projects [Cot90] [NV89] [PL90] [VR87] seems to speak for the event-driven mechanism being especially helpful for physical-behavior simulation.

It may be that compiled simulation has not yet made a breakthrough in digital system design because there are no simulators that are readily available, or perhaps they are not sufficiently general to assist the digital circuit design community. We have developed what we hope to be such
Figure 1: "Werner diagram" of a circuit with two combinational components, four D-flipflops, and eight named "nets." Data flow through all boxes is left to right. Multiplexer "Mn" is controlled by signal "choose." Signal "muxout" achieves a synchronous feedback loop. Note the combinational path from "choose" to "muxout."

4 Objection Addressed

It may be argued that compiled simulation does not allow the designer to see "glitches" in his circuit before it settles, so that he cannot know whether or not he is clocking his circuit too quickly. We point out that one can seldom be certain he has found the maximum clock rate by using simulation. We could never picture ourselves using an event-driven simulator for this purpose.

What has changed in the past few years, and what (at least in our case) has made timing simulation obsolete, is the breakthrough of timing analysis. Timing analysis (timing verification) [Hit82] [Ous85] [DR86] [Szy86] [JM87] [Jou87] [Che88] [WS88] [YGD88] [RA89] is a static analysis method which analyses the circuit itself, deducing longest paths and taking into account other electrical effects, and produces complete and very accurate data [Sh88] on the expected timing behavior of the design. Timing characteristics such as the maximum clocking speed, setup and hold constraints, and output delays are readily obtained. Once we know the maximum clock speed from the timing analysis, then we are interested only in the behavior of the "settled" circuit from our simulator, and have no interest in the "noisy" part of the simulation trace. But it is only the "noise" that is lost when choosing compiled simulation over event-driven simulation; the settled ("logical") behavior is simulated correctly.

5 Static Scheduling

The method for building a simple\textsuperscript{5} compiled simulator is outlined here. We start with a circuit description as in Figure 1 where we assume a single system clock, and that all clocked components are D-flipflops. All other components must be combinational, that is, unclocked. For now we can also assume that only the D flipflops are capable of retaining any internal state, that is, there is "memory" only in the D flipflops. Every component (D flipflop or combinational unit) has a clear directionality.

We then transform the circuit into an internal computer data structure as shown in Figure 2. This is called a directed cyclic bipartite graph. The components (boxes) become one type of vertex, while the nets become the other type of vertex. There is never a direct path from one type of vertex to another of its own type. The vertices (both components and nets) are joined by arrows which we will call "edges." We see in Figure 2 that all components (boxes) have only one edge leaving them, but in general there can be any number, even zero. A box with no input edge is called a constant box (it outputs a fixed value, since it is unclocked). Finally, we see that all nets (ovals) have at most one input edge. If a net has two or more input edges, then it must be a bus driven by (for example) tristate drivers, and we will not deal with such nets in this paper. Nets with no input edges are circuit inputs, and nets with no output edges can only be circuit outputs. In the process of creating this graph we keep track

\textsuperscript{5}This (implicitly) levelized zero-delay compiled simulator is the simplest variant. Advanced researchers in this field would point out that we are confusing two orthogonal issues: event-driven vs. oblivious simulation, with interpretive vs. compiled simulation; but we consider this appropriate for this paper. Advanced treatment is found in [Lew89] [WM90] [WM90].
Figure 2: Decomposition of circuit in Figure 1 into its bipartite graph. Ovals are nets, and boxes are components (both clocked flipflops and combinational units are boxes). The lines with arrowheads are called “edges.” The algorithm described in this paper effectively breaks the cyclic graph at the flipflops (marked “D”) and the breadth-first scheduling algorithm requires and checks that the resulting “opened” graph is acyclic.

of all of the D flipflops, all of the constant boxes, and all of the input nets, since they have a privileged role in the scheduling algorithm.

Components will become subroutines in our simulator, and nets will become static variables. “Scheduling” means generating a component subroutine call with the relevant nets passed as parameters. The resulting ordered list of subroutine calls is the simulator’s static schedule and the entire schedule is executed, unconditionally, once each simulation cycle. Each subroutine may only modify its output nets. All nets are initialized to some appropriate undefined marking before commencing simulation.

The D flipflop model (subroutine) also has state memory (per latch instance) and behaves as follows during simulation: when commanded to read, it copies its internal slave memory onto its output net. When commanded to store, it stores the value on its input net into its internal master memory; and when commanded to shift, it copies its master memory into its slave memory. For the algorithm below, it is possible for a D flipflop to be scheduled to store before it is scheduled to read. This is no problem, because D flipflops behave in this master-slave fashion. In addition to the “main” schedule produced by the three routines immediately below, there is a separate “shift” schedule which must be generated as well. Finally, we show the steps for executing one complete simulation cycle.

The main schedule is determined with the help of two recursive procedures together with a global routine that invokes them.

(Visiting a component.)
for each input edge coming into this box do
    if this edge has not yet been traversed then
        Cannot yet schedule this box; return to caller.
    endif.
end_for.
if this box is a D flipflop then
    if this procedure invoked from global routine then
        Schedule this flipflop for reading.
    else
        Schedule this flipflop for storing, and then
        Return to caller.
    end_if.
else
    Schedule this combinational box.
end_if.
for each output edge leaving this box do
    Mark this edge as having been traversed.
    Invoke “Visiting a net” for the net on this edge.
end_for.
Return to caller.
(Visiting a net.)
Mark this net as having been visited.
for each output edge leaving this net do
  Mark this edge as having been traversed.
  Invoke "Visiting a component" for the box on this edge.
end for.
Return to caller.

(Global routine.)
for each constant box do
  Invoke "Visiting a component" for this box.
end for.
for each flipflop do
  Invoke "Visiting a component" for this D-flipflop.
end for.
for each input net do
  Invoke "Visiting a net" for this net.
end for.
Scheduling should be complete.
for each D flipflop do
  if not scheduled for storing then
    Scheduling failure; stop.
  endif.
end for.
for each combinational box do
  if not scheduled then
    Scheduling failure; stop.
  endif.
end for.
Scheduling succeeded; stop.

The "main" schedule determined above is combined with the "shift" schedule determined below. The complete simulation cycle can then be given.

(Shift schedule routine.)
for each D flipflop do
  Schedule this D flipflop to shift.
end for.
Return to caller.

(Complete simulation cycle.)
Establish the value for each input net.
Execute the main schedule.
Execute the shift schedule.

The schedule is found in linear time, proportional to the number of components plus nets. The scheduling algorithm is nondeterministic, but all schedules will produce the same result. Failure to find a complete schedule occurs when there is an asynchronous loop in the design, as in Figure 3. We simply define a correct circuit to be one for which a schedule can be found. This is the same as requiring that there be no asynchronous (purely combinational) cyclic paths in the circuit. This is a reasonable restriction, if not a clear requirement, for synchronous circuit design. Note also that circuit inputs become implicitly synchronous, as one would expect for a simulator which does not model asynchronous effects.

We point out that since combinational boxes are scheduled only once each cycle, then one can use this fact to model memories by including an internal state in such boxes. This must be done carefully however, since in doing so we cross the thin line that separates structural simulation from behavioral simulation [Jen91].

6 Two-Phase Clocking

The tool in [Jen91] actually models two-phase nonoverlapping clocking, so instead of D flipflops it uses gated and transparent latches. In this case two schedules are required; one for when the first ("white") clock is high, and one for when the opposite ("black") clock is high. We must then distinguish between "white" latches and "black" latches during scheduling. The "white" schedule is used for the half-cycle when all the white latches are open. To create it, scheduling starts from all black latches, and proceeds through all white latches up to the inputs of the black latches. For that schedule, the white latches are commanded both to store and to propagate its input value to its output. Definition of a correct model for two-phase clocking is beyond the scope of this report, the reader is referred to [Jen90]. Also, as discussed in [Jen90], the method has the surprising side-benefit of allowing the simulator to run in reverse.

7 Discussion

During the design process, the electronics designer works with different models [Phi90], each serving a different purpose, as he transforms his original problem statement into a hardware solution. It has long been a frustration that each model seems to need its own special simulator. For the microprocessor mentioned in Section 2 we wrote a behavioral model in ISPS [Bar74], a structural model in the

Unlike [WM89] we assume that no component contains multiple independent paths.
Figure 3: Scheduling failure. A slight modification was made to the circuit of Figure 1 (the flipflop between signals “muxout” and “feedback” was removed, joining the two signals directly). An asynchronous loop is apparent: signal “feedback” through box “plus” through signal “sum” through box “Mn” back to “feedback.”

compiled simulator [Jen91], and finally, electrical models (managed by a silicon compiler [Joh84]). An even more involved scenario is given in [DB87]. This complexity helps explain the search for the ultimate “multi-level simulator.” It continues to elude us, and we find that designers, as all other craftsmen, are still forced to choose the right tool for the task at hand.

Even the designers of VHDL had no such utopia in mind [Sha86] but knew VHDL would be appropriate in the neighborhood of gate models, and warned that its discrete-event semantics would only be suitable for a particular class of electrical circuits. It is therefore surprising to see VHDL modeling efforts in the literature [Leu89] [ST+89] [SB89] [MB90] discussing some very curious, if not doubtful, applications of event-driven simulation technology. The rationalization is usually some supposed potential for “multi-level simulation” in VHDL of course.

However, compiled simulation is not an all-or-nothing proposition. It is yet another simulation technique that could well be combined with other mechanisms to advantage. As evidence we cite [BB+87] [WM90] [Lew89] as examples of integrating the method with other known techniques. With its help we may actually come one step closer to a truly practical multi-level simulator.

Acknowledgements

This work was supported by the Swedish National Board for Technical Development (STU) under contract number 87-02427.

References


