A General Method for Compiling Event-Driven Simulations

Robert S. French, Monica S. Lam, Jeremy R. Levitt, Kunle Olukotun
Computer Systems Laboratory
Stanford University, CA 94305-4055

Abstract—We present a new approach to event-driven simulation that does not use a centralized run-time event queue, yet is capable of handling arbitrary models, including those with unclocked feedback and nonunit delay. The elimination of the event queue significantly reduces run-time overhead, resulting in faster simulation. We have implemented our algorithm in a prototype Verilog simulator called VeriSUIF. Using this simulator we demonstrate improved performance vs. a commercial simulator on a small set of programs.

I. Introduction

Modern digital system design relies heavily on simulation to reduce the number of design errors and to improve system efficiency. In large system designs so much time is spent in simulation that it has become a design bottleneck. Event-driven simulation and levelized compiled simulation are two well-known simulation techniques that are currently used in digital system design.

In event-driven simulation, events are managed dynamically by an event scheduler. The main advantage of event-driven scheduling is flexibility; event-driven simulators can simulate both synchronous and asynchronous models with arbitrary timing delays. The disadvantage of event-driven simulation is low simulation performance.

Levelized compiled code logic simulators have the potential to provide much higher simulation performance than event-driven simulators because they eliminate much of the run-time overhead associated with ordering and propagating events [1, 2]. This is done by evaluating all components once each clock cycle in an order that ensures all inputs to a component have their latest value by the time the component is executed. The main disadvantage of levelized compiled simulation techniques is that they are not general. Most levelized compiled logic simulators cannot simulate models with arbitrary delays (RAVEL [3] is a notable exception). Furthermore, these techniques will not work on asynchronous models or models with unclocked feedback. In practice, even though most digital systems are synchronous, asynchronous chip interfaces are common.

In this paper we present a general method for compiling event-driven models called *static simulation* that combines the generality of event-driven simulations and the efficiency of the levelized simulation approach. Like event-driven simulation, our technique applies to all general models, including both synchronous and asynchronous designs. The only restriction is that any specified delays in the simulation must be known constants at compile time. For efficiency, our technique schedules the events at compile time, thus eliminating the need for a run-time event queue and its associated overhead. We replace the event queue with inexpensive run-time tests where necessary. For the models we have tested, these run-time tests incur significantly less overhead than a run-time event queue.

This research was supported in part by ARPA contract DABT63-94-C-0054.

We represent the event-driven behavior with an *event graph*, whose vertices represent events in the simulation and whose edges represent the causal relationships between the events. We apply the general technique of *partial evaluation* to schedule the events as well as possible using statically available information. Specifically, the compiler tries to approximate the dynamic simulation process by keeping track of all the available static information that affects the contents of the run-time event queue in a dynamic simulation. This general method can be applied uniformly to all models, unlike previous approaches such as LECSIM [4], TORTLE [5] and [6].

To test our algorithm, we have implemented a prototype simulator, called VeriSUIF, using the SUIF (Stanford University Intermediate Format) compiler system [7]. We chose Verilog mainly because it is a relatively simple language to implement. The VeriSUIF simulator is particularly useful for long-running regression tests because it produces a faster simulation than other techniques. However, our current implementation is unsuitable for other phases of the design process because it does not support interactive debugging.

The remainder of the paper is organized as follows. First we give a brief overview of Verilog and describe the features of Verilog that we support. Then we describe the event graph representation which underlies our method. Next we describe our mathematical model of traditional event-driven simulation and our static simulation technique. Finally, we discuss some optimizations, experimental results, and our conclusions.

II. A Brief Overview of Verilog

All Verilog programs are composed of *modules*. These modules may be instantiated inside of other modules to create a hierarchy that represents the structure of the hardware system. Modules contain three types of concurrent process statements: *initial* blocks, *always* blocks, and *continuous assignments*. Initial blocks are executed once at the beginning of the simulation, while always blocks are executed repeatedly. Initial and always blocks consist of statements that are executed sequentially, and each can wait on a signal to change value using `wait` or `@` statements. A continuous assignment is an assignment to a wire whose left hand side continuously reflects the current state of the variables on the right hand side. A Verilog simulator simulates a model sequentially by removing events from an event queue, executing the events, and placing new events on the queue as they become activated.

VeriSUIF supports a subset of Verilog. We do not support tasks, functions, fork/join, or the disable statement. However, we do not foresee any difficulty in extending our system to handle these features.

III. Event Graph Representation

We represent a model using an *event graph*. Event graphs provide a representation for the static simulation algorithm to work on, and transformations on event graphs can be used to improve simulation performance.

An event graph partitions the model into *events*, represented by vertices, and uses directed edges to represent relationships between events. We define an event to be the largest unit of a program that will execute atomically during simulation, which corresponds to the code that would be executed during one step of an event-driven simulation. The semantics of the original language determines precisely how the boundaries between events are determined from the original source. In Verilog, events have boundaries at the start and finish of always and initial blocks, at explicit delays, and at statements that wait on signals (such as `@` and `wait`). Each vertex has associated with it executable code from the Verilog program.

An edge represents a causal relationship between one event and another. Edges can take two forms: *sensitizing* and *triggering.* Event v_1 sensitizes a following event v_2 whenever they are separated by an `@` or `wait` statement. The execution of event v_1 makes v_2 sensitive; v_2 can then be triggered by other events.

Triggering actions cause events to be scheduled for execution. There are three kinds of trigger actions: control flow, data flow, and delays. v_1 can trigger v_2:

- if program control flows from v_1 to v_2 whenever a boolean expression b evaluates to true after executing v_1,
- if the execution of v_1 changes the value of an expression x, awaited by v_2,
- or if the delay statement `#d` separates v_1 and v_2.

Note that not all control flow is represented by control flow edges. Only control flow that crosses an event boundary is represented in this way. All other control flow is contained within a single event.

A Verilog program is represented by an event graph $G = \langle V, v_0, N_0, E_n, E_b, E_x, E_d \rangle$, where

- V is the set of events,
- $v_0 \in V$ is the start event,
- $N_0 \subset V$ is the set of initially sensitive events,
- $E_n \subset V \times V$ represents the sensitizing edges,
- $E_b \subset V \times V \times B$, where B is the set of boolean expressions, represents possible triggering actions due to control flow,
- $E_x \subset V \times V \times X$, where X is the set of expressions, represents possible triggering actions due to change of expression values, and
- $E_d \subset V \times V \times \mathcal{W}$, where $\mathcal{W}$ is the set of whole numbers, represents possible triggering actions due to delay statements.

An example of an event graph for a small Verilog program is shown in Figure 1.

(a)

```
        module example();
        reg clk;
  E1    initial clk=0;
  E2    always begin clk = ~clk; #1; end
  E3    always @(clk) $display("clk");
        endmodule
```

(b)

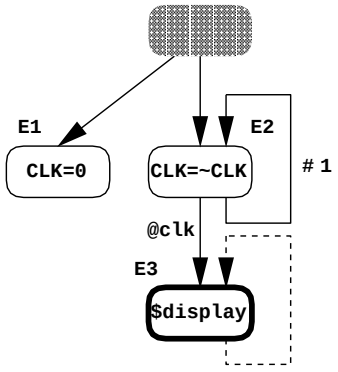

Figure 1: (a) a Verilog program and (b) its event graph. Trigger edges are solid, sensitizing edges are dashed. The start event is shaded, and the initially sensitive events have thick edges.

IV. Dynamic Simulation

Before we describe our compiler algorithm, we first formally describe how we model the traditional dynamic simulation process. During simulation, the state of the computation is captured by a quadruple $s = \langle N, R, D, M \rangle$, where

- $N \subset V$ is the set of sensitive events,
- $R \subset V$ is the set of events ready to be executed,
- $D \subset V \times \mathcal{W}$ is the set of all $\langle v, d \rangle$ such that event v has d time steps left before it can execute,
- M is the memory store that maps each variable to its current value.

We assume the existence of the following operations:

- OneOf(R) deterministically chooses an event from the set of ready events R.
- Exec(v, M) applies the code associated with event v to the memory store M and returns a new memory store. It may also produce side effects such as generating output.
- Eval(b, M) evaluates the boolean expression using the memory store M and returns TRUE or FALSE.
- Chgd(v, x, M) indicates if the execution of event v with initial memory store M changes the value of expression x.

We define $s' = \text{Next}(\langle N, R, D, M \rangle, G)$ by $\langle N', R', D', M' \rangle$, where

(Type I Transition) This represents the execution of an event in the current simulation time step:

if $R \neq \emptyset$, then
Let $v = \text{OneOf}(R)$ in

$$\begin{aligned} M' &= \text{Exec}(v, M) \\ N' &= N - \{v\} \cup \{v' | \langle v, v' \rangle \in E_n\} \\ R' &= R - \{v\} \end{aligned}$$

$$\cup \{v' | \langle v, v', b\rangle \in E_b \wedge \text{Eval}(b, M')\}$$
$$\cup \{v' | \ \langle v, v', x\rangle \in E_x \wedge v' \in N \wedge \text{Chgd}(v, x, M)\}$$
$$D' \quad = \quad D \cup \{\langle v', d\rangle | \langle v, v', d\rangle \in E_d\}$$

(Type II Transition) This represents incrementing the simulation time to the next time during which an event can occur.

if $R = \emptyset$ and $D \neq \emptyset$, then
Let

$$d_0 = \min_{\langle v,d\rangle \in D} d,$$
$$A = \{\langle v, d\rangle | \langle v, d\rangle \in D \wedge d = d_0\}$$

in

$$M' = M$$
$$N' = N$$
$$R' = \{v | \langle v, d\rangle \in A\}$$
$$D' = \{\langle v, d\rangle | \langle v, d + d_0\rangle \in (D - A)\}$$

(Type III Transition) This represents the end of simulation:

if $R = \emptyset$ and $D = \emptyset$, then
$s' = \perp$, denoting no next state.

The *dynamic simulation* of an event graph G is a sequence of states $s_0, s_1, \ldots, \perp$, where

- $s_0 = \langle N_0, \{v_0\}, \{\}, M_0\rangle$, where M_0 represents the initial memory store that maps every variable to an appropriate initial value (for Verilog, all variables are initially `x`).
- $s_{i+1} = \text{Next}(s_i, G)$.

V. Static Simulation

Manipulating the event queue is a considerable source of run-time overhead. Our approach to reducing this overhead is to have the compiler perform as much of the simulation as possible at compile time and completely eliminate the run-time event queue. We do this by collecting information about which events *could* be executed at any given point during the simulation and generating code for those events guarded by run-time tests. When possible, we track variable values during compilation so that more decisions about whether an event will execute can be made at compile time instead of run time.

As an example, consider Figure 1. At the start of simulation, the compiler finds the initial event E1 in the event graph and emits the corresponding code. On analyzing the code itself, the compiler determines that the value of `clk` is set to 0. Following the trigger edge from the start event to E2, the compiler emits code for E2 and similarly notes that the value of `clk` is changed to 1. The change of `clk` triggers E3, so the compiler emits the code for E3, and notes that E3 remains sensitive. At this point, the only possible transition to take corresponds to the delay edge from E2 back to itself. The compiler generates code to increment the time. It then repeats this same series with the `clk` value initially set to 1, and arrives at the same state it was in after E1 executed: E2 is ready to execute and `clk` is 0. The compiler simply wraps a loop around that section of code. As there are no other events waiting to be executed, the compilation is complete. The final code is:

```
E1    clk = 0
      forever
E2        clk = ~clk
E3        $display("clk")
          time = time+1
E2        clk = ~clk
E3        $display("clk")
          time = time+1
      end
```

In an event-driven simulator this example would cause two events to be scheduled for each time step. Our approach completely eliminates all of the overhead; there is no run-time event queue and no conditional tests are performed at run time. The generated code is what one expects of a cycle-based compiled-code simulator[2]; however, our technique does not require special treatment of clock signals and is thus more general.

The above example shows how variable values can be tracked at compile time. In general, not all variables have known values at compile time, and even if they do, the compiler cannot afford to track all of them. For example, it is intractable to record all the values generated by an increment to an initially known value within a loop. Our compiler only tracks values due to assignments of constants and simple boolean expressions. Without knowing the exact values, the compiler may not be able to determine if an event will definitely execute. For these cases, the compiler generates run-time tests to ensure that the simulation is correct.

We now discuss our static simulation technique in more detail.

A. Static Simulation State

The compiler runs through a static simulation of the program at compile time. A static simulation state $\overline{s} = \langle \overline{N}, \overline{R}, \overline{D}, \overline{M}\rangle$ captures a conservative approximation of the corresponding dynamic simulation state $s = \langle N, R, D, M\rangle$ as follows:

- $\overline{N} \subset V \times \{\text{MAY}, \text{MUST}\}$ contains all events that may or must be sensitized ($v \in N \Rightarrow \langle v, m\rangle \in \overline{N}$ and $\langle v, \text{MUST}\rangle \in \overline{N} \Rightarrow v \in N$). An event can not be paired with MAY and MUST simultaneously.
- $\overline{R} \subset V \times \{\text{MAY}, \text{MUST}\}$ contains all events that may or must be ready to be executed currently ($v \in R \Rightarrow \langle v, m\rangle \in \overline{R}$ and $\langle v, \text{MUST}\rangle \in \overline{R} \Rightarrow v \in R$). An event can not be paired with MAY and MUST simultaneously.
- $\overline{D} \subset V \times \mathcal{W} \times \{\text{MAY}, \text{MUST}\}$ contains all events that may or must be waiting to be executed in a future simulation time ($\langle v, d\rangle \in D \Rightarrow \langle v, d, m\rangle \in \overline{D}$ and $\langle v, d, \text{MUST}\rangle \in \overline{D} \Rightarrow \langle v, d\rangle \in D$). An event can not be paired with MAY and MUST simultaneously.
- $\overline{M}$ is the memory store that maps each variable to its current value. If we do not know the value of a variable at compile time, it maps to $\perp$. (For each mapping var $\mapsto$ val in $\overline{M}$ such that val $\neq \perp$, var $\mapsto$ val is in M.)

We modify the functions from the dynamic simulation as follows:

- $\overline{\text{OneOf}}(\overline{R})$ deterministically chooses a tuple $\langle v, m\rangle$ from the set of ready events $\overline{R}$.
- $\overline{\text{Exec}}(v, m, \overline{M})$ applies the code associated with event v to the memory store $\overline{M}$ and returns a new memory store. If $m =$ MUST, any variable written by v can be stored in $\overline{M}$ if it is known to be a constant at the end of v, subject to the restrictions outlined earlier. Otherwise, any variable written by v must map to $\perp$ in $\overline{M}$.

- $\overline{\text{Eval}}(b, \overline{M})$ evaluates the boolean expression using the memory store M and returns MUST (TRUE), MAY (some variable in b maps to $\perp$ in $\overline{M}$), or MUST NOT (FALSE).
- $\overline{\text{Chgd}}(v, x, \overline{M})$ indicates if the execution of event v with initial memory store $\overline{M}$ changes the value of expression x and returns MUST (TRUE), MAY (some variable in x maps to $\perp$ in $\overline{M}$) or MUST NOT (FALSE).

We define the $\wedge$ operator for MAY, MUST, and MUST NOT as follows:

	MUST NOT	MAY	MUST
MUST NOT	MUST NOT	MUST NOT	MUST NOT
MAY	MUST NOT	MAY	MAY
MUST	MUST NOT	MAY	MUST

We define a function Merge(S_1, S_2), where S_1 and S_2 are sets of tuples $\langle v, m\rangle$. We assume the existence of a similar function for $\langle v, d, m\rangle$.

$$\begin{aligned}\text{Merge}(S_1, S_2) \quad = \quad & S_1 \cup S_2 \\ & -\{\langle v, m\rangle |\ \langle v, m\rangle \in S_1 \cup S_2 \\ & \qquad \wedge (\ \ m = \text{MUST NOT} \\ & \qquad\quad \vee \ \ m = \text{MAY} \\ & \qquad\qquad \wedge \langle v, \text{MUST}\rangle \in S_1 \cup S_2)\}\end{aligned}$$

Finally, we extend the state transition function Next to static states. The definition of $\overline{\text{Next}}$ ensures that the set of sensitized events, currently waiting events and delayed events are a superset of the corresponding event sets in dynamic simulation. We define $\overline{s}' = \overline{\text{Next}}(\langle \overline{N}, \overline{R}, \overline{D}, \overline{M}\rangle, G)$ by $\langle \overline{N}', \overline{R}', \overline{D}', \overline{M}'\rangle$, where

(Type I Transition) This represents the execution of an event in the current simulation time step:

if $\overline{R} \neq \emptyset$, then
Let $\langle v, m\rangle = \overline{\text{OneOf}}(\overline{R})$ in

$$\begin{aligned}\overline{M}' \quad = \quad & \overline{\text{Exec}}(v, m, \overline{M}) \\ \overline{N}' \quad = \quad & \text{Merge}(\ \overline{N} - \{\langle v, m\rangle \text{ if } m = \text{MUST}\}, \\ & \qquad \{\langle v', m\rangle | \langle v, v'\rangle \in E_n\}) \\ \overline{R}' \quad = \quad & \text{Merge}(\overline{R} - \{\langle v, m\rangle\}, \\ & \qquad \{\langle v', m'\rangle |\ \langle v, v', b\rangle \in E_b \\ & \qquad\qquad \wedge m' = (m \wedge \overline{\text{Eval}}(b, \overline{M}'))\} \\ & \qquad \cup \{\langle v', m''\rangle |\ \langle v, v', x\rangle \in E_x \\ & \qquad\qquad \wedge \langle v', m'\rangle \in \overline{N} \\ & \qquad\qquad \wedge m'' = (\ \ m \wedge m' \\ & \qquad\qquad\qquad \wedge \overline{\text{Chgd}}(v, x, \overline{M}))\}) \\ \overline{D}' \quad = \quad & \text{Merge}(\overline{D}, \{\langle v', d, m\rangle | \langle v, v', d\rangle \in E_d\})\end{aligned}$$

(Type II Transition) This represents incrementing the simulation time to the next time during which an event can occur.

if $\overline{R} = \emptyset$ and $\overline{D} \neq \emptyset$, then
Let

$$\begin{aligned} d_0 \quad = \quad & \min_{\langle v, d, m\rangle \in \overline{D}} d, \\ A \quad = \quad & \{\langle v, d, m\rangle | \langle v, d, m\rangle \in \overline{D} \wedge d = d_0\}\end{aligned}$$

in

$$\begin{aligned}\overline{M}' \quad = \quad & \overline{M} \\ \overline{N}' \quad = \quad & \overline{N} \\ \overline{R}' \quad = \quad & \{\langle v, m\rangle | \langle v, d, m\rangle \in A\} \\ \overline{D}' \quad = \quad & \{\langle v, d, m\rangle | \langle v, d + d_0, m\rangle \in (\overline{D} - A)\}\end{aligned}$$

(Type III Transition) This represents the end of simulation:

if $\overline{R} = \emptyset$ and $\overline{D} = \emptyset$, then
$\overline{s}' = \overline{\perp}$

The *static simulation* of an event graph G is a sequence of states, $\overline{s_0}, \overline{s_1}, \ldots, \overline{\perp}$, where

- $\overline{s}_0 = \langle N_0, \{v_0\}, \{\}, M_0\rangle$, and
- $\overline{s}_{i+1} = \text{Next}(\overline{s}_i, G)$.

B. Code Generation

When an event v is chosen via the $\overline{\text{OneOf}}$ function during a type I transition, the compiler emits the code corresponding to that event. To ensure that the code associated with a MAY event is executed only when the dynamic conditions are right, the code is predicated by a condition that is evaluated at run time. The compiler introduces a run-time boolean variable for each vertex in the event graph that is maintained throughout the program execution such that it reflects whether the corresponding event is sensitized. Likewise, for each static simulation state a trigger variable is introduced to indicate if the event to be executed during that state has been triggered. Code is generated to set or reset these variables after each event. When a type II transition is taken, code can be emitted to increment the global simulation clock.

We observe that the naive static simulation described above may generate an infinite list of states on some event graphs. This may happen under two circumstances:

1. The static simulation may never reach a state whose set of ready events $\overline{R}$ is empty. The algorithm described so far could keep taking type I transitions forever. This will occur, for example, when simulating circuits with unclocked feedback.
2. Similarly, the static simulation may never reach a state whose delayed events $\overline{D}$ is empty. This occurs in synchronous designs as the clock signals change continuously until some dynamic condition occurs (see Figure 1).

We use the general method of finding fixed points to solve both of these problems. The technique is based on the observation that the set of possible static simulation states is finite.

Theorem 1 *The number of possible static simulation states for any event graph G is finite.*

Proof: For a given $G = \langle V, v_0, N_0, E_n, E_b, E_x, E_d\rangle$, the sets V and E_d are finite. The following relations must hold: $|\overline{N}| \leq |V|$, $|\overline{R}| \leq |V|$, and $|\overline{D}| \leq |E_d| \cdot d_m$ where $d_m = \max_{\langle v, v', d\rangle \in E_d} d$. The latter equation is derived by observing that for all delayed events $\langle v', d', m'\rangle \in \overline{D}$ triggered by edge $\langle v, v', d\rangle$, $d' \leq d$. (That is, the remaining time to wait cannot be greater than the original delay in the program.) Thus there are only a finite number of possible $\overline{N}$, $\overline{R}$, and $\overline{D}$ sets. As discussed earlier, we limit the possible values in $\overline{M}$ to constants that appear in the program and unary operators on these

constants. Thus there are a finite number of possible $\overline{M}$ sets, and a finite number of possible static simulation states.

Our compiler algorithm is as follows. Whenever the compiler generates a new state $\overline{s}_i$, it compares $\overline{s}_i$ with all the previously generated states. We are guaranteed by Theorem 1 that either the simulation will eventually terminate or we will find two matching states. If there is no match, the static simulation proceeds as discussed above. Otherwise, let $\overline{s}_j$ be the matched state. The sequence of static simulation states following $\overline{s}_i$ must be exactly the same sequence that follows $\overline{s}_j$. Thus, it is not necessary to continue to simulate statically, since the compiler can produce the equivalent code sequence by inserting a branch operation from $\overline{s}_{i-1}$ to $\overline{s}_j$, thus creating a loop consisting of events $\overline{s}_j, \ldots, \overline{s}_{i-1}$.

Once a loop has been found, it is necessary to remove elements from the current $\overline{R}$ and $\overline{D}$ so that static simulation may continue. We remove the elements corresponding to events executed during the loop. This set of events also allows us to construct the exit condition for the loop, since it is only when none of these events are ready to be triggered that the loop may exit.

After finding a loop we continue simulation until we find another loop or until simulation is complete (a type III transition is taken). In general, loops consisting only of type I transitions will be generated first (unclocked feedback) and enclosed by loops consisting type I and II transitions (clocked feedback).

VI. Optimizations

While the algorithm presented in the previous section generates working code, we have found a number of optimizations that are useful to produce a more efficient simulation.

- *Continuous assignment optimizations* – Continuous assignments represent assignments to wires where the left hand side continuously reflects the current state of the variables on the right hand side. Rather than treating them as separate events we can inline them directly into the code, thus drastically reducing the number of events we need to schedule. Such inlining must be done in moderation, however, to limit the increase in code size. This optimization is performed on the event graph before scheduling takes place.

- *Sensitization optimization* – If an event immediately sensitizes itself after execution, and is sensitive at the beginning of simulation, it will always be sensitive. Thus it is not necessary to test for sensitivity in the generated code. In many models, this is the case for the majority of events.

- *Levelization* – During the course of event-driven simulation, it is possible to execute events multiple times as values propagate through the model. These multiple executions are not required for the correct answer and adversely impact simulation performance. Through proper ordering of event execution unnecessary events can be eliminated. This process is generally called *levelization* and is used in levelized compiled code simulators [1, 2].

  One can add a form of levelization to an event-driven simulator by being intelligent about which events are retrieved from the event queue. We implement this in our compiler by assigning a *level* to each event in the graph based on its maximum length path from the start event. Then, when the $\overline{\text{OneOf}}$ function chooses events from $\overline{R}$, it chooses the event with the lowest level number.

VII. Preliminary Experimental Results

We used six benchmarks to test our implementation. Five of these are from Coumeri and Thomas [8]. This suite consists of small benchmarks that vary in hierarchy depth, partitioning, and operators used. The benchmarks include a series of test vectors and the vectors are run up to 1,000,000 times to produce measurable run times. The arms_counter benchmark is particularly interesting because it contains unclocked feedback. This suite also contains five other benchmarks that we are not using. They are all variations on a 64-bit bit-wise adder, and our prototype does not yet support efficient 64-bit operations. The final benchmark, MIPS-Lite, is a simple behavioral description of a MIPS-compatible processor.

We compared the performance of our simulator with VCS 2.3 from Chronologic Simulation, a state-of-the-art commercial simulator. Both VCS and VeriSUIF generate C code. In all cases, the generated code was compiled with the MIPS C compiler v3.19 with –O2. The simulations were executed on a Silicon Graphics Indigo with 64MB of memory and a 100 MHz R4000 processor. Scheduling overhead was measured using the *pixie* tool. The results are shown in Table 1.

The overhead we show for VCS indicates the amount of time spent in the run-time library manipulating the event queue. VCS does some optimizations to reduce run-time overhead by bypassing the event queue in some cases, such as simple signal propagation. In these cases it uses function calls to activate events. The time spent performing these function calls is not included in our measurements. The overhead for VeriSUIF shows the amount of time spent in the code that sets and tests triggers.

The absolute run time is shown as well as the time spent performing scheduling tasks. When looking at the absolute times, it is important to realize that VCS has a highly tuned code generator, and in some cases generates better code than VeriSUIF. This is apparent on the gcd benchmarks where our poorer computational code is the sole reason for the reduced performance. However, even with this disadvantage, we still run almost two times faster on average. More interesting is the time spent performing scheduling tasks. Here we can see that the time spent performing event queue management in VCS can be substantial. On average, we spend only 4% as much time as VCS in scheduling overhead.

It is also interesting to measure the benefit of tracking variable values during static simulation. A comparison is presented in Table 2. The percentage of scheduling states that do not require a run-time test nearly double with variable tracking, from 40% to 76%, with several requiring no tests for the entire simulation (the 40% of states that are marked MUST even without variable tracking are due to unconditional control flow and delays). As a result of eliminating these run-time tests, variable tracking was able to reduce scheduling overhead on all the benchmarks by an average of 1.9 times. In many cases this is because the benchmark used a series of constant test vectors and the compiler was able to determine which events must trigger during each test. Also, in the arms_counter and MIPS-Lite benchmarks there is a top level clock and the compiler can eliminate run-time tests for events that depend on the clock edges. However, there is a disadvantage to using variable tracking: the number of static states can increase significantly. This is because it takes longer to detect a cycle when variable values are taken into account.

VIII. Conclusions

In this paper, we introduce the static simulation technique as a general method for compiling event-driven models into efficient simulation code. The method has two innovations. First, we use a general event graph that succinctly captures the semantics of an event-driven simulation. Second, we use the general technique of partial evaluation to schedule the events as well as possible using statically available

Benchmark Name	Absolute Runtime (sec)			Scheduling Overhead (sec)		
	VCS	VeriSUIF	Speedup	VCS	VeriSUIF	Speedup
2901/alg	124.35	67.62	1.84	43.65	4.67	9.35
2901/block	235.83	263.44	0.90	65.80	25.47	2.58
arms_counter	42.60	10.77	3.96	36.68	3.37	10.88
diffeq	20.91	5.24	3.99	15.72	0.18	87.33
gcd	9.30	10.62	0.88	1.43	0.05	28.60
MIPS-Lite	16.35	10.06	1.63	4.66	2.46	1.89
Average			1.90			23.44

Table 1: Comparison of the run-time speed and scheduling overhead of VCS and VeriSUIF.

	Without variable tracking			With variable tracking			
Benchmark Name	Schedule Size	% MUST	Scheduling Overhead	Schedule Size	% MUST	Scheduling Overhead	Overhead Speedup
2901/alg	2591	67%	11.28	4315	100%	4.67	2.42
2901/block	7332	24%	62.11	10321	74%	25.47	2.44
arms_counter	1744	28%	5.37	2225	57%	3.37	1.59
diffeq	121	52%	0.42	237	100%	0.18	2.33
gcd	54	55%	0.08	97	100%	0.05	1.50
MIPS-Lite	103	11%	2.87	186	23%	2.46	1.17
Average		40%			76%		1.91

Table 2: Comparison of scheduling overhead with and without variable tracking. All times are in seconds.

information. This general technique can be applied uniformly to optimize the simulation of arbitrary models including those containing unclocked feedback and nonunit delay.

Our prototype implementation of the simulator uses the SUIF compiler system. We achieve an average speedup of about two when compared to VCS 2.3 on six benchmarks. More importantly, our average scheduling overhead amounts to only 4% of that found in the VCS code.

Acknowledgments

We would like to thank John Sanguinetti and Randy Allen of Chronologic for their help with VCS, and Brian Murphy for help with this document.

References

[1] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge, "HSS - a high-speed simulator," *IEEE Transactions on Computer-Aided Design*, vol. 6, pp. 601–616, July 1987.

[2] C. Hansen, "Hardware logic simulation by compilation," in *25th ACM/IEEE Design Automation Conference*, pp. 712–715, 1988.

[3] E. J. Shriver and K. A. Sakallah, "RAVEL: assigned-delay compiled-code logic simulation," in *Proceedings of the 1992 IEEE International Conference on Computer-Aided Design*, pp. 364–368, Nov. 1992.

[4] Z. Wang and P. M. Maurer, "LECSIM: A levelized event driven compiled logic simulator," in *27th ACM/IEEE Design Automation Conference*, pp. 491–496, 1990.

[5] D. M. Lewis, "A hierarchical compiled code event-driven logic simulator," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 726–737, June 1991.

[6] E. G. Ulrich and D. Herbert, "Speed and accuracy in digital network simulation based on structural modeling," in *19th ACM/IEEE Design Automation Conference*, 1982.

[7] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, pp. 31–37, Dec. 1994.

[8] S. L. Coumeri and D. E. Thomas, "Benchmark descriptions for comparing the performance of Verilog and VHDL simulators," in *Proceedings of the 1994 International Verilog HDL Conference*, pp. 14–16, Mar. 1994.