Efficient Circuit Partitioning to Extend Cycle Simulation Beyond Synchronous Circuits

Charles J. DeVane*
Viewlogic Systems, Inc.
Marlboro, MA 01752-4615

Abstract

Cycle simulation techniques, such as levelized compiled code, can ordinarily be applied only to synchronous designs. They usually cannot be applied to designs containing circuit features like combinational paths, multiple clock domains, generated clocks, asynchronous resets, and transparent latches. This paper presents a novel partitioning algorithm that partitions a non-cycle-simulatable circuit containing these features into sub-circuits that can be cycle simulated. Cycle simulation techniques can be applied to the individual sub-circuits, and the whole collection of sub-circuits can be simulated together using conventional co-simulation techniques. Empirical results demonstrate that this approach brings the benefits of cycle simulation to circuits that were previously impossible to cycle simulate. The partitioning algorithm requires time and space linear in the size of the circuit, and in practice is very fast. We also discuss how the key ideas presented here can be applied to accelerate HDL simulation.

1 Introduction

The rapidly expanding size and complexity of digital systems has caused simulation to become a major bottleneck in the design flow. To overcome this bottleneck, designers are increasingly willing to abandon traditional event-driven timing simulation and adopt a combination of static timing analysis and functional simulation. Static timing analysis relieves the simulator from the burden of verifying timing, so that only the circuit's logical function needs to be simulated. The circuit can be simulated at a level of abstraction higher than event-driven timing simulation using faster algorithms.

In contexts where simulation of timing is not needed, levelized compiled code (*LCC*) has been used very successfully on large, purely synchronous designs. This technique consumes very little memory and is usually very fast, but it generally supports only a zero delay model of logic and is appropriate only for synchronous circuits. This model of simulation is sometimes called *cycle simulation*, because it focuses on computing circuit values only on clock cycle boundaries.

Unfortunately, most designs are not purely synchronous but include features like combinational paths, asynchronous resets, transparent latches, multiple clock domains, and generated clocks which are not readily supported by the cycle simulation abstraction. It can be feasible to cycle simulate these features in a test environment where the stimulus is a predetermined set of test vectors. However, it is increasingly important to verify circuits in context inside a complete system. Problems arise as soon as a circuit with any of these features is embedded in a system simulation where external feedback may cause the circuit input to be a dynamic function of the circuit outputs.

How can we exploit cycle-simulation algorithms like LCC to simulate circuits with these features within a complete system simulation? If circuits that cannot be cycle-simulated can be automatically and efficiently partitioned into subcircuits that can be cycle-simulated then the benefits of cycle simulation algorithms can be extended to a broader class of circuits. Using conventional co-simulation techniques, a hybrid simulation environment can apply cycle simulation to the subcircuits that allow it, and conventional event-driven simulation to the rest of the circuit (see Figure 1). Hybrid environments similar to this are already widely used to support requirements like mixed-signal simulation, third-party models, mixed-language HDL simulation, etc. The key problem, then, is how to partition circuits into cycle-simulatable sub-circuits.

In this paper we present an algorithm to partition circuits into cycle-simulatable sub-circuits. The algorithm is efficient, generally being linear in both time and space, and very fast in practice. We first establish a notion of cycle simulation as a level of abstraction, and then review prior work in this light. Next we define a class of circuits that can be cycle-simulated. Then we discuss several common circuit features which preclude cycle simulation: combinational paths, multiple

*The author's current address is: The MathWorks, Inc., Natick, MA 01760; cdevane@mathworks.com.

0-89791-993-9/97 $10.00 © 1997 IEEE

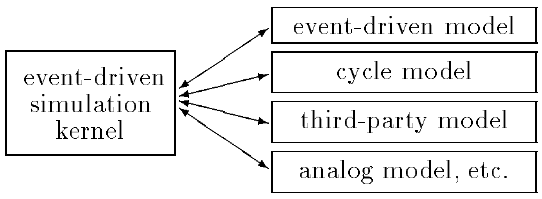

Figure 1: Hybrid simulation environment supporting event models and cycle models as well as other forms of simulation.

clock domains, generated clocks, asynchronous resets, and transparent latches. For each case it is shown how to partition the circuit for efficient simulation. An efficient algorithm is presented that performs the desired circuit partitioning, along with a brief sketch of the proof of correctness and analysis of complexity. Experimental results of applying this partitioning to actual industrial designs are then presented to demonstrate the efficiency and effectiveness of the algorithm. The paper closes with an application of these structural ideas to accelerating HDL simulation.

2 Background

2.1 The Cycle Simulation Abstraction

A hierarchy of abstraction for circuit simulation is illustrated in Figure 2. At higher levels of abstraction less information is computed, which leads to faster algorithms. However, these faster algorithms are only applicable to circuits for which the abstraction is appropriate. Conversely, at lower levels of abstraction more information is computed, leading to algorithms that are slower but more broadly applicable. For example, SPICE2[7] is much slower than event-driven logic simulation, but can be applied to analog circuits for which event-driven logic simulation is inappropriate.

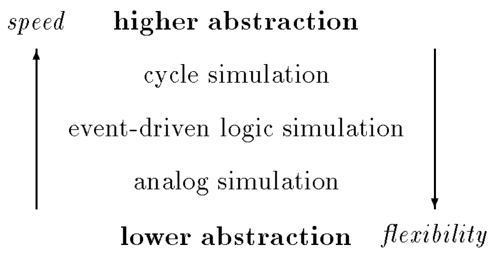

Figure 2: A hierarchy of abstraction for circuit simulation.

The term *cycle simulation* is sometimes equated with LCC simulators [2] [4] [9]. However, this paper uses the term more generally to describe any simulation algorithm that focuses on computing and storing only the values of circuit outputs and storage elements at most once per clock cycle. By this definition cycle simulation operates at a higher level of abstraction than event-driven logic simulation. Because they compute and store fewer values at fewer time points, algorithms that exploit the cycle simulation abstraction can be expected to have an inherent performance advantage for appropriate circuits.

It is important to distinguish between the level of abstraction supported (or exploited) by a simulator and the algorithms used to implement the simulator, because multiple algorithms may be used to support a given level of abstraction. For example, instead of using LCC a cycle simulator can use techniques such as branching code [1] [6] or threaded code [5]. Likewise, an event-driven simulator need not have a run-time event queue [3].

2.2 Previous Work

HSS/3[2], Terse[4], and SSIM[9] all demonstrate the performance potential of software LCC simulators. Terse can also perform static timing analysis during model compilation. However, all these systems were restricted to synchronous designs.

Research continues to improve the performance of event-driven simulation. However, due to the difference in levels of abstraction, the performance of event-driven simulation still has not approached that of cycle simulation. For example, VeriSUIF[3] used compiler optimization techniques to eliminate the run-time event queue while maintaining the event-driven abstraction. It achieved impressive performance for an event-driven simulator, but still falls short of cycle simulation. Although VeriSUIF is a compiled simulator and incorporates a form of levelization, it does not use what is conventionally known as LCC as in [2][4][9]. By comparison, VeriSUIF's levelization optimization is limited in its ability to suppress multiple evaluations. Because VeriSUIF supports the event-driven abstraction, computing intermediate values at intermediate time points, its performance potential is necessarily limited compared to cycle simulation. Another potential performance problem may be compilation time.

Another relevant approach to improving simulation performance is *demand-driven* simulation, as illustrated by BACKSIM[8]. BACKSIM did not support timing analysis and was used essentially as a cycle simulator. During simulation, BACKSIM uses a depth-first search to recursively walk backward through the circuit, collecting only the information required to compute the desired output values. BACKSIM can be viewed as using *dynamic levelization* at run-time instead of the compile-time *static levelization* used by LCC simulators. The general concept of demand driven evaluation can be seen behind our notion of *fanin* gates (Section 3.2).

Several event-driven simulators have sought higher performance by exploiting techniques like LCC at a relatively fine granularity. These simulators could be considered as hybrids between the event-driven abstraction and the cycle abstraction. For example, HSS/4[2] implemented event-driven scheduling of fanout-free trees of gates, where the trees are evaluated obliviously. Similarly, LECSIM[10] reduces scheduling overhead by employing two partitioning strategies. First, it groups each strongly connected component into a single block which is evaluated by LCC inside a small loop that iteratively evaluates the block until it stabilizes. This scheme is much cheaper than using the main scheduling mechanism. Second, LECSIM can group fanout-free trees into single blocks for scheduling. Our approach can be viewed as an extension of this hybridization, where we use a different partitioning that typically results in a far coarser level of granularity.

3 Extending Cycle Simulation

This section first defines a class of circuits which can be cycle-simulated. Several common circuit features that preclude cycle simulation are discussed as well as how each feature affects partitioning for efficient simulation.

3.1 Assumptions

For simplicity, assume that all circuit primitives are either combinational gates or registers (storage elements). Initially registers are assumed to be only edge-triggered flip flops but in Section 3.7 registers are extended to include transparent latches. All primitives have some connected inputs and some connected outputs. All signals have uni-directional signal flow; i.e., any bi-directional signals have been converted to uni-directional signals using bus resolution primitives. The circuit contains no combinational feedback (asynchronous state machines).[1] Only zero-delay simulation semantics are of interest.

Throughout this paper the term *gate* designates an arbitrary combinational logic function of any complexity and bit width, assuming only that it contains no state. For example, a gate may be as simple as a 1-bit 2-input AND, or as complex as a 64-bit carry-lookahead adder.

3.2 Circuits Which Can Be Cycle-Simulated

The key idea behind cycle simulation is to compute circuit values only at clock edges. Furthermore, cycle simulation is usually only concerned with the values of registers and circuit outputs.

[1] Combinational cycles can be handled by extracting them into a separate partition which is co-simulated along with the other partitions, similar to LECSIM's handling of strongly connected components [10].

This, however, implies that cycle simulation only makes sense for circuits that are controlled by a clock – synchronous state machines. Other circuits, such as purely combinational logic or asynchronous state machines don't make sense for cycle simulation because they may require evaluation at times other than clock edges.

To define more precisely a class of circuits that can be cycle-simulated begin by considering a single edge-triggered register. Clearly this can be correctly simulated by only evaluating it on the active edge of the clock. Now consider Figure 3. Zero-delay simulation semantics only require evaluating this circuit at the active clock edge to get correct results. This is because the input of the combinational gate will change only in response to the active clock edge. This reasoning can be extended to any number of gates as long as their transitive fanin is derived strictly from the outputs of registers sharing the same clock.

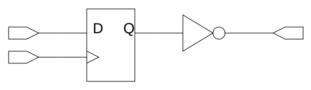

Figure 3: A register with a fanout gate.

Next consider Figure 4. Conventional event-driven techniques will evaluate the combinational gate any time its inputs change, which could be many times per clock cycle. However, its output is used only on the active clock edge. Since only the values of registers and outputs need to be computed, the gate only needs to be evaluated on the active clock edge. As before, this reasoning can be extended to any number of gates as long as their transitive fanout ultimately goes strictly to the inputs of registers sharing the same clock.

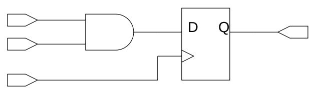

Figure 4: A register with a fanin gate.

A circuit which can be cycle-simulated is defined as a collection of edge-triggered registers sharing the same clock plus all the combinational gates which meet either or both of the following conditions:

1. the gate's entire transitive fanin comes from the registers, or
2. the gate's entire transitive fanout goes to the registers.

Gates meeting condition 1 are called *fanout* gates, because they are the fanout of the registers of the circuit.

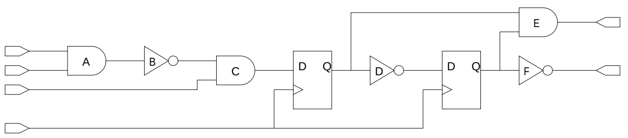

Figure 5: Examples of fanin and fanout gates.

Gates meeting condition 2 are called *fanin* gates, because they are the fanin of the registers of the circuit.

Figure 5 shows an instructive example. Both registers share the same clock. Gates A, B, and C meet condition 1. Gates E, and F, meet condition 2. Gate D meets both conditions.

3.3 Combinational Paths

What about gates that meet neither condition? Assume for the moment that all registers in the circuit share the same clock. If a gate does not meet condition 1 then its transitive combinational fanin must include the circuit's input. Similarly, if the gate does not meet condition 2 then its transitive combinational fanout must include the circuit's output. So there must be some combinational path from a circuit input through the gate and on to a circuit output. Figure 6 illustrates this situation. Gate G is neither a fanin nor a fanout gate. The combinational path through gate G is shown in bold.

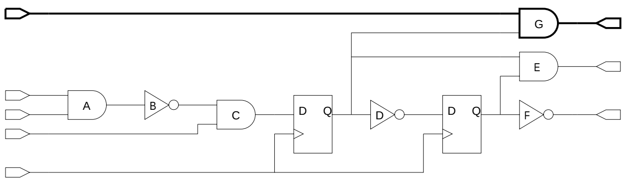

Figure 6: Gate G cannot be cycle-simulated due to a combinational path.

How should these gates be simulated? If the simulation environment simply exercises the circuit with vectors in a relatively synchronous fashion, it may be acceptable to evaluate these gates once per clock cycle (or once per vector). However, an event-driven simulation environment has no *a priori* knowledge of when the gate's inputs change, or when its output is needed. Hence, for correctness, the gate should be evaluated whenever its inputs change. These gates are called *input-triggered* gates.

If all the registers in the circuit share a clock, the circuit can be subdivided into two partitions: a *clock-triggered partition* containing all the registers plus all the fanin and fanout gates and an *input-triggered partition* containing all the remaining gates. The clock-triggered partition need be evaluated only once per

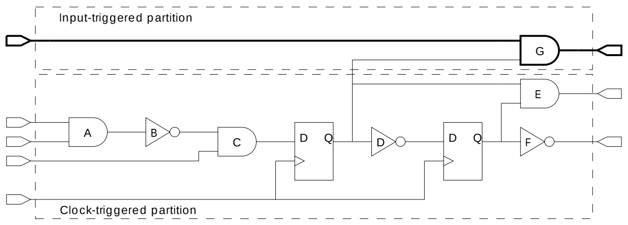

Figure 7: Partitioning into *clock-triggered* and *input-triggered* partitions.

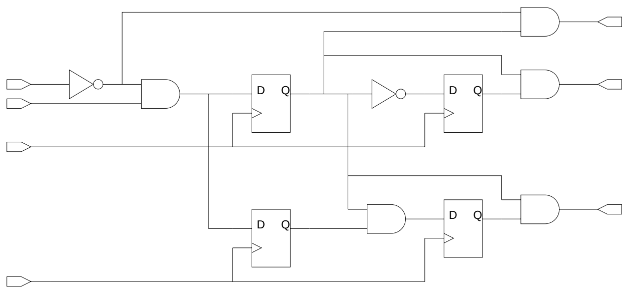

Figure 8: A circuit with multiple clock domains.

clock cycle, while the input-triggered partition must be evaluated whenever its inputs change. Figure 7 illustrates the partitioning of Figure 6.

Nothing has been said about how the two partitions should be evaluated. Any evaluation technique can be used for either partition, including LCC, threaded code [5], branching code [1] [6], etc.

3.4 Multiple Clock Domains

Introducing registers with different clocks (i.e., multiple clock domains), as illustrated in Figure 8, requires strengthening the definition of fanin and fanout gates.

It is not enough simply to restrict the transitive combinational fanout of fanin gates only to registers: all the registers must share the same clock. Clearly, such a fanin gate can be associated with the registers in its fanout and evaluated only when those registers are clocked. Similarly the transitive combinational fanin of fanout gates must be restricted to registers sharing the same clock. Such a fanout gate should be associated with the registers in its fanin and only evaluated when those registers are clocked.

This introduces an ambiguity in the partitioning. Some gates can be both a fanout gate of one clock domain, and a fanin gate of a different clock domain. The choice of which domain the gate is associated with is arbitrary, and may be made according to other expedients. For example, the choice could be made to minimize signals that cross domains.

Some gates may fail to be fanin or fanout gates of any partition. It is no longer the case that such gates must

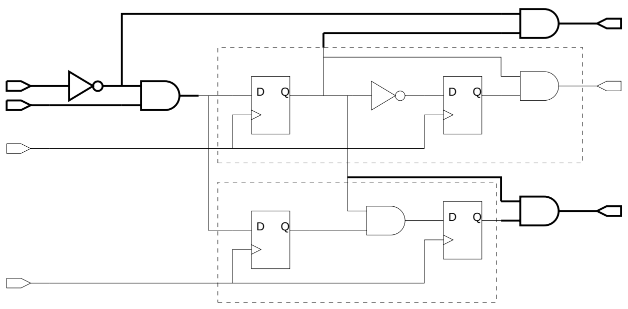

Figure 9: Partitioning of the circuit from Figure 8.

lie on a combinational path from circuit input to circuit output. A gate that fails to be a fanout gate must have transitive combinational fanin either from two or more clock domains, or from a circuit input plus one or more clock domains. Likewise, a gate that fails to be a fanin gate must have transitive combinational fanout either to two or more clock domains, or to a circuit output plus one or more clock domains.

To simulate these circuits we extend the previous partitioning strategy to construct one clock-triggered partition for each clock, and one input-triggered partition. All the registers that share a clock are placed into that clock's clock-triggered partition together with the associated fanin and fanout gates. All the gates that are neither fanin nor fanout gates are placed into the input triggered partition. Figure 9 shows the result of partitioning on the circuit of Figure 8. The gates in the input-triggered partition are shown in bold.

3.5 Generated Clocks

So far the discussion has assumed that all the clock signals are inputs to the circuit. What if some clocks are generated within the circuit? Consider Figure 10. Gate A is a fanin gate since it only drives the data input of the register. But gate B, which drives a register's clock input, is different. If gate B is made a fanin gate associated with the register and evaluated only when the register is clocked, neither it nor the register will ever be evaluated. Rather, the partition must be evaluated whenever gate B's inputs change. But this may increase the number of inputs the partition is sensitive to and complicate the partition evaluation code by forcing it to detect the rising edge of the internal signal.

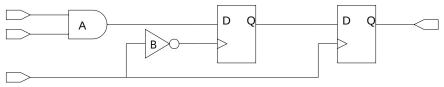

Figure 10: A circuit with a generated clock.

A simpler approach, illustrated in Figure 11, makes the register's clock input an input to the register's partition. This forces B into another partition as either a fanout gate or an input-triggered gate, as determined by the rest of B's context. The simulation environment can provide the necessary clock edge detection, as it must be doing anyway to support co-simulation of multiple circuit partitions. The desired partitioning is achieved by treating the clock inputs of registers exactly as if they were circuit outputs. This will prevent the driving gate from being a fanin gate, thus forcing it to be a fanout gate or an input-triggered gate and forcing the clock signal to be an input to the register's partition.

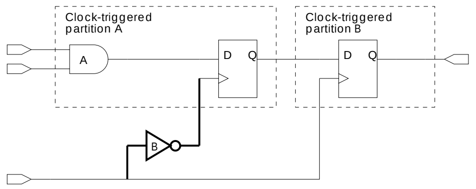

Figure 11: Partitioning of the circuit from Figure 10.

3.6 Asynchronous Resets

So far all partitions have been sensitive only to a single input, the clock, or else sensitive to all inputs. Asynchronously reset registers present an issue because they must be evaluated not only on the active clock edge but also when the reset is asserted. This is readily handled by extending the notion of clock domains into *trigger domains*. A *trigger* is any $<signal, event>$ tuple which causes a register to require evaluation. A *trigger domain* is all the logic that is sensitive to a set of triggers. For example, all the registers that are clocked by $<risingedge, CLK>$ and reset by $<fallingedge, \overline{RESET}>$ together with their fanin and fanout gates form a single trigger domain.

3.7 Transparent Latches

So far all storage primitives have been edge-triggered flip flops, which need to be evaluated only on the asserting edge of the clock (or reset). Transparent latches present a problem because they must be evaluated not only on clock edges, but also on data edges when the clock is asserted. Their behavior is essentially combinational during the active clock phase.

For the purposes of cycle simulation, some design methodologies can model transparent latches as edge-triggered flip flops. For example, this is frequently the case for multi-phase clock methodologies that use transparent latches as the primary storage element. In these cases latches can be converted into edge-triggered flip flops. The flip flops can be clocked on the opening or closing edge of the clock, whichever is most appropriate for the individual design.

$$\begin{tikzcd}
 & & \top \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
label_i \arrow[drr] & label_{i+1} \arrow[dr] & & \cdots \arrow[dl] & label_{i+N} \arrow[dll] \\
 & & \perp & &
\end{tikzcd}$$

Figure 12: Lattice of values for representing fanin and fanout information.

However, some designs will not simulate correctly without genuinely transparent latch behavior. This case can be handled by treating both the clock and data inputs to the latch as sensitive inputs. The latch must be evaluated on the opening edge of the latch clock, and whenever the data changes.

4 The Partitioning Algorithm

The input to our partitioning algorithm is a network of registers and combinational gates connected by unidirectional signals.

Recall that *gate* refers to an arbitrary combinational logic function of any complexity or width, and *register* refers to either edge-triggered flip flops or level-sensitive latches. Gates and registers are collectively referred to as *primitives*. For convenience, the network also contains primitives representing the circuit inputs and outputs.

The circuit must contain no cycles of combinational gates; i.e., any cycles in the network must be broken by registers.

The output is the same network with all primitives labeled according to their assigned partition; i.e., all primitives with the same label are in the same partition.

For computing transitive combinational fanin and transitive combinational fanout and labeling primitives we use a lattice of values as in Figure 12. The lattice includes a label for each partition. Registers are labeled according to their trigger sets. Gate are initially labeled with $\top$ to indicate that no fanin (fanout) information is known about the gate. If fanin from (fanout to) only one partition i is discovered the gate will be relabeled $label_i$. If fanin from (fanout to) multiple partitions is discovered the gate will be relabeled $\perp$.

Each gate has the following fields:

gate.fanin the lattice value for the gate's transitive fanin;

gate.fanout the lattice value for the gate's transitive fanout;

gate.partition the final partition assignment for the gate.

Each register has a field `reg.partition` to hold the register's label. All of these fields are initialized to $\top$.

Conceptually, the algorithm has four phases. First, the registers are partitioned according to trigger sets such that all the registers that share a given set of triggers are contained in their own partition. These partitions are the kernels of the clock-triggered partitions. The final circuit partitioning will have these partitions plus the (possibly empty) input-triggered partition. Each partition (including the input-triggered partition) is assigned a unique label from the lattice. Each register is labeled with its partition's label.

Second, fanin information is propagated forward from register outputs and circuit inputs to the gates, leaving each gate labeled according to its transitive combinational fanin. This is accomplished using a depth-first search which first recursively visits all of a gate's fanin before labeling the gate. The recursive search stops at registers, which have already been labeled, and circuit inputs, which are treated as if labeled with the input-triggered partition's label. If all the gate's fanin are labeled identically then the gate receives that label; otherwise the gate is labeled with $\perp$. This label is stored in `gate.fanin`.

Third, fanout information is propagated backward from register inputs and circuit outputs to the gates, leaving each gate labeled according to its transitive combinational fanout. This is accomplished using a depth-first search which first recursively visits all of the gate's fanout before labeling the gate. The recursive search stops at registers, which have already been labeled, and circuit outputs, which are treated as if labeled with the input-triggered partition's label. As a special case, sensitive register inputs (e.g., clock, asynchronous reset, or latch data inputs) are handled as if they were circuit outputs. If all the gate's fanouts are labeled identically then the gate receives that label; otherwise the gate is labeled with $\perp$. This label is stored in `gate.fanout`.

Finally each gate is classified as fanin, fanout or input-triggered and assigned its final label. If `gate.fanout` is neither $\perp$ nor the input-triggered partition's label then the gate is a fanin gate and `gate.fanout` is assigned to `gate.partition`. Otherwise, if `gate.fanin` is neither $\perp$ nor the input-triggered partition's label then the gate is a fanout gate and `gate.fanin` is assigned to `gate.partition`. If the gate can be neither a fanin gate nor a fanout gate, the gate belongs in the input-triggered partition and that partition's label is assigned to `gate.partition`.

Once a gate is classified, `gate.fanin` is no longer needed, so in practice it may share the same storage with `gate.partition`.

5 Proof and Analysis

The proof of correctness is straightforward and the details are omitted for brevity. Correct propagation of fanin and fanout information can be proven by induction on the length of combinational paths. The gate classification and labeling is constructed from the definitions of fanin and fanout gates.

Analysis of space and time complexity is also straightforward. The space and effort of collecting trigger sets is proportional to the number of registers multiplied by the number of sensitive inputs per register, which is ordinarily a small constant (usually at most 3). This is clearly bounded by the size of the network. Propagation of fanin and fanout information are each performed by depth-first searches, which require time and space proportional to the size of the network. Thus the complete algorithm requires space and time linearly proportional to the size of the network.

6 Empirical Results

6.1 Implementation

The partitioning algorithm has been incorporated in a VHDL hybrid event-driven/cycle simulator based on a commercially available VHDL simulator. The portion of a design that conforms to a synthesizable subset of VHDL (for example, an ASIC design) can be simulated as a cycle model or in normal event-driven mode. The non-synthesizable portion (typically a testbench requiring file I/O) is simulated in event-driven mode.

The cycle models are constructed by first synthesizing and optimizing a network of RTL-level primitives, such as wide logic gates, adders, multipliers, encoded multiplexers, etc. Note that each RTL primitive is the equivalent of many conventional gates. The model compiler's circuit optimizations can selectively transform transparent latches into edge-triggered flip-flops triggered on either the opening or closing edge of the clock (Section 3.7), and can also transform asynchronously reset registers into synchronously reset registers. The circuit is then partitioned and a 2-state or 4-state LCC model, using native code, is generated for each partition. The partition models are combined with a highly efficient "microkernel" to form a single model which interfaces to the host VHDL simulator through its C language interface. The microkernel manages all interaction amongst the partitions, and between the partitions and the host simulator.

6.2 Measurements

To demonstrate the utility of this approach, Table 1 gives empirical measurements taken on three "industrial strength" test cases. The first of these, case M, is a model of the MIPS R3000 microprocessor executing an instruction diagnostic. The other two cases are ASIC designs from industry. Each design includes a testbench running in event-driven mode.

RTL primitives is the primitive count for each design after optimizations. *Total partitions* is the number of partitions produced for each design, followed by a breakdown of partition types. *Clock-triggered (1 signal)* is the number of partitions that were sensitive to only a single signal; *clock-triggered (2 signals)* is the number of partitions sensitive to two signals. *input-triggered* indicates whether an input-triggered partition was generated. *Partitioning time* is the approximate time in seconds which the cycle model compiler spent in the partitioning algorithm running on a Sun Sparc 20 with 256MB of memory. *Partitioning memory* is the approximate memory required by the partitioning algorithm in addition to the basic network data structures. Due to a refinement in the implementation only 4 bytes per primitive are required.

A detailed discussion of the simulator's architecture and performance is beyond the scope of this paper, but some performance numbers are given to compare the hybrid simulation to the same design simulated purely in the event-driven simulator. *Hybrid acceleration* is the directly measured overall speedup gained by using hybrid simulation versus purely event-driven simulation. *Cycle fraction* is a conservative estimate of how much of the design was accelerated using cycle simulation, measured as a fraction of the purely event-driven simulation time. *Cycle acceleration* is a conservative estimate of how much the cycle fraction was accelerated. *Event density* is an estimate of the fraction of signals that changed value during a cycle model evaluation.

test case	M	S	N
RTL primitives	2,051	42,949	96,063
total partitions	1	4	11
clock-triggered (1 signal)	1	1	5
clock-triggered (2 signals)	0	2	5
input-triggered	no	yes	yes
partitioning time (secs)	.18	2.3	6.5
partitioning memory (KB)	8	168	375
hybrid acceleration	6.8x	9.9x	1.8x
cycle fraction	88%	91%	75%
cycle acceleration	34x	55x	3.8x
event density	17%	16%	1%

Table 1: Partitioning and simulation performance for three large designs.

6.3 Discussion

The partitioning algorithm runs very fast and consumes very little memory. In fact, in all cases the time spent on partitioning was less than 1% of the total model compilation time. Memory consumption was a similarly negligible fraction.

Case M is the archetypical purely synchronous cycle-simulatable circuit. The entire model falls into one partition, and is accelerated substantially.

Case S is more typical of real designs. It contains one clock domain that is entirely asynchronously reset. It contains another clock domain which is partly asynchronously reset, so the reset portion goes in one partition and the remainder in another partition. It also includes a substantial amount of logic in combinational paths.

Case N is a very large and very complex design. Its relatively poor acceleration is accounted for by the low event density. Using threaded compiled code might have produced better performance than LCC for this design. It would not be difficult for a simulator to incorporate both algorithms, offering the user a choice, or even dynamically switching between them during simulation by periodically measuring the event density.

The most important point about this data is that real industrial designs containing multiple clock domains, asynchronous resets, generated clocks, combinational paths, etc. were simulated *correctly* using a cycle simulation algorithm within a system simulation context.

7 Accelerating HDL designs

The ideas beneath the notions of fanin and fanout gates can be generalized into an "optimizing" transformation to accelerate conventional HDL simulation.

Suppose an HDL program contains *clocked processes* which are sensitive to only a few signals (e.g., CLK and $RESET$), and *combinational processes* which are sensitive to all their inputs. If the values computed by a combinational process are used only by a clocked process, then the combinational process can be merged into the clocked process. This is analogous to grouping fanin gates with their registers. Similarly, if all the input values used by a combinational process come from a clocked process, then the combinational process can be merged into the clocked process. This is analogous to grouping fanout gates with their registers.

Merging processes in this fashion has two benefits. First, reducing the sensitivity list of the combinational processes may reduce the frequency of evaluation. Second, reducing the interprocess data flow reduces the number of queued events, which are expensive.

8 Conclusions

We have presented a novel circuit partitioning algorithm that extends the benefits of cycle simulation to circuits that were previously not cycle-simulatable. The algorithm is efficient, and fast in practice. Results showed that this approach does indeed yield substantial performance improvements for very large industrial designs. We also showed how the key ideas underlying this structural algorithm can be applied to accelerating HDL simulations as well.

Acknowledgements

Thanks to Rick Sullivan and Mark Coiley for introducing me to this problem, for the numerous ideas they contributed, and especially for many lively and enjoyable debates. Thanks also to John Bond and Tim McBrayer for their proofreading and very helpful suggestions.

References

[1] Pranav Ashar and Sharad Malik. Fast functional simulation using branching programs. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1995.

[2] Zeev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joseph D. Rutledge. Hss–a high speed simulator. *IEEE Transactions on Computer-Aided Design*, pages 601–617, July 1987.

[3] Robert M. French, Monica S. Lam, Jeremy R. Levitt, and Kunle Olukotun. A general method for compiling event-driven simulations. In *Proceedings of the Design Automation Conference*, pages 151–156, 1995.

[4] Craig Hansen. Hardware logic simulation by compilation. In *Proceedings of the Design Automation Conference*, pages 712–715, 1987.

[5] David M. Lewis. Hierarchical compiled event-driven logic simulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1989.

[6] Patrick C. McGeer, Kenneth L. McMillan, Alexander Saldanha, Alberto L. Sangiovanni-Vincentelli, and Patrick Scaglia. Fast discrete function evaluation using decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1995.

[7] L. W. Nagel. Spice2: A comuter program to simulate semiconductor circuits. Technical Report Rep. No ERL-M520, Electronics Research Laboratory, University of California, Berkeley, CA, May 1975.

[8] Steven P. Smith, M. Ray Mercer, and Bishop Brock. Demand driven simulation: Backsim. In *Proceedings of the Design Automation Conference*, pages 181–187, 1987.

[9] Laung-Terng Wang, Nathan E. Hoover, Edwin H. Porter, and John J. Zasio. Ssim: A software levelized compiled-code simulator. In *Proceedings of the Design Automation Conference*, 1987.

[10] Z. Wang and P. M. Maurer. Lecsim: A levelized event driven compiled logic simulator. In *Proceedings of the Design Automation Conference*, pages 491–496, 1990.