# Symbolic Timing Simulation Using Cluster Scheduling

Clayton B. McDonald Randal E. Bryant *
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213
{clayton,bryant}@ece.cmu.edu

## ABSTRACT

We recently introduced symbolic timing simulation (STS) using data-dependent delays as a tool for verifying the timing of full-custom transistor-level circuit designs, and for the functional verification of delay-dependent logic. While STS leverages efficient symbolic encodings to yield huge gains over conventional simulation methodologies, it still suffers from a problem known as *event multiplication*. We discuss this problem and present an event-list management technique based on event-clusters, and a new simulator which utilizes this technique. Finally, we demonstrate substantial speedups on a wide range of test cases, including exponential improvement on a simple logic chain.

## 1. INTRODUCTION

As design complexity continues to increase, efficient methods of verifying timing and functionality become both more important and more difficult. Symbolic simulation has been used in the past to greatly increases the effective throughput of simulation-based verification methodologies, and also as a component of a number of formal verification strategies. Symbolic timing simulation is an extension of this technique that correctly accounts for circuit propagation delays.

Symbolic simulation is a form of data-parallel simulation in which Boolean values are used to encode a set of input data patterns. In conventional simulation the user applies a pattern of constant 0's and 1's to each of the circuit inputs, steps the simulator, and verifies that the outputs and state elements have settled to the desired values. With a symbolic simulator, the user may substitute Boolean variables for any of the input values to signify that the input may be either a 0 or a 1. If the user applies $n$ Boolean variables, the symbolic simulator will perform the equivalent of $2^n$ conventional simulations. The outputs and state elements of the circuit will evaluate to Boolean functions of the input variables, which can be verified against the desired behavior.

Symbolic simulation relies on having an efficient means of encoding Boolean functions to represent the values of circuit nodes. Typically, this takes the form of Binary Decision Diagrams(BDDs) [3]. Though the memory required to encode a Boolean function is provably exponential in the worst case, BDDs have been shown to be efficient and easily-manipulated data structures for representing a large number of interesting functions.

*This research was supported by the SRC (contract DC-068)

We recently applied symbolic *timing* simulation to the verification of full-custom transistor-level circuits, as an alternative to static timing analysis. Custom circuits often contain transistor topologies that defy heuristic recognition, causing static analyzers to miss timing checks. A symbolic timing simulator can simulate the timing of all possible input patterns in parallel and infer the correctness of all internal timing by checking for correct functional behavior. This avoids the dependence on correct identification of all possible latches, flip-flops, dynamic gates, self-timed circuitry, etc.

SirSim [9] is a transistor-level symbolic timing simulator based on the delay calculation procedures in IRSIM[10]. It demonstrated the computational feasibility of symbolic timing simulation on a number of reasonable-sized benchmarks. However, SirSim suffers from a major bottleneck which we term the *event multiplication* problem.

In Section 2, we discuss the extension from symbolic simulation to symbolic timing simulation as a natural progression of different delay models. Section 3 discusses event clusters and how they can be used to improve performance. In Section 4 we present a cluster-based event-management scheme and a new symbolic timing simulator, STEED. Section 5 discusses experimental results and demonstrates the advantages of our approach.

## 2. DELAY MODELS

Like conventional simulators, symbolic simulators can be built upon a wide range of delay models. Historically, symbolic simulators have been used for functional verification and have tended to utilize either zero- or unit-delay models. However, assigned delay and data-dependent delay models have been successfully implemented for other applications.

### 2.1 Constant Delay

One of the earliest references to symbolic simulation [8] described a simple zero-delay model. Zero-delay simulation implies a levelized sweep through the circuit going from the inputs to the outputs. At each level, a function is computed that represents the value of each node based on the input variables. While it is extremely efficient, only acyclic circuits can be dealt with in this manner.

To handle circuits containing feedback or state-holding nodes, a unit-delay model can be used. Implementation of the unit-delay model typically utilizes event-lists, where an *event* is a change in the value function for a particular node. To start the simulation, the input values are *scheduled* by placing them in the list of events to be executed. The simulator then executes all events in the event list, computes new values for fanouts of the changed nodes, and schedules them back onto the event list. This process is repeated

until the circuit reaches a steady-state, as signalled by an empty event list.

Symbolic simulators which have implemented the unit-delay model include MOSSYM[2] and Cosmos [4]. Both of these simulators actually implement a mixture of zero-delay and unit-delay to gain efficiency when the generality of the full unit-delay model is not required.

The next level of complexity is the assigned-delay model. Here, the delay from one node to the next (typically through a logic gate) has an assigned value $d$. To handle this case, we simply extend the event-driven scheme from the unit-delay model by sorting the event-list by time. At each time-step the event at the head of the event list is removed (along with any others having the same time value) and executed. As before, the effects on its fanout are computed, and scheduled into the now-sorted event list to be updated $d$ time-units later. Devadas et. al. [7] implemented an assigned-delay symbolic simulator to analyze the transition-delay of gate-level circuits.

## 2.2 Data-Dependent Delays

Recently, we demonstrated the feasibility of extending symbolic simulation to a data-dependent delay model[9]. The difficulty lies in the fact that the time of an event is dependent on the values of the input-variables applied to the circuit. Consider the skewed inverter in Figure 1(a). When the input changes value from variable $a$ to variable $b$ (Figure 1(b)), either of which could symbolize a 0 or a 1, it is not clear when the event on node $out$ should be scheduled.

To handle this case properly, we can schedule an event at each of the potential timepoints, along with a mask $M$ that indicates under which input patterns the event will occur. When the event is removed from the event list, we compute the node's new value as:

$$NewValue = (M \wedge EventValue) \vee (\overline{M} \wedge OldValue)$$

For the inverter case, we schedule an event at time 100 with mask $M = \overline{a}b$, and at time 200 with mask $M = a\overline{b}$, both events having the new value $\overline{b}$. At time 100, the new node value will become $(\overline{a}b \wedge \overline{b}) \vee (\overline{\overline{a}b} \wedge \overline{a}) = \overline{a}\overline{b}$, and at time 200 $(a\overline{b} \wedge \overline{b}) \vee (\overline{a\overline{b}} \wedge \overline{\overline{a}\overline{b}}) = \overline{b}$. This resultant series of functions is shown in Figure 1(c). The intermediate value $\overline{a}\overline{b}$ specifies that the output will only be true between times 100 and 200 if both $a$ and $b$ were false, leaving the output true continuously.

This mechanism was implemented in SirSim, a transistor-level symbolic timing simulator based on the IRSIM Elmore delay-calculation scheme. While a substantial speedup was attained over an equivalent exhaustive IRSIM simulation (up to $10^{33}$ for a 64-bit adder), SirSim still suffers from the problem of *event multiplication*. Since each new node value is scheduled several times, one for each potential delay, the effects of a single event multiply at each level. This can cause a total number of events that is exponential in the circuit depth.

# 3. EVENT-CLUSTERS

To address the event-multiplication problem in data-dependent symbolic timing simulation, we first introduce the concept of *event clusters*. An event cluster is a set of events on a single node with mutually disjoint masks. The set of events that result on a fanout node from a single node value change will always form an event cluster.

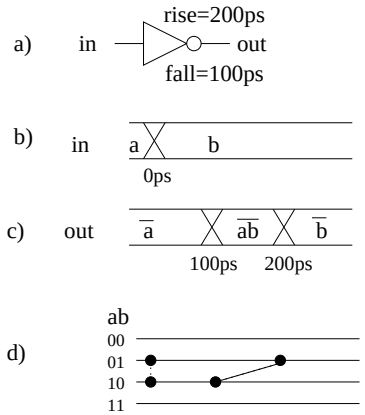


**Figure 1: Skewed Inverter Timing Diagram**

Take for example the events on the output of the skewed inverter in Figure 1(c). Both output events share a common resultant value, but differ in their event times and masks. However, the masks are disjoint ($a\overline{b} \wedge \overline{a}b = 0$). In Figure 1(d) we plot the timeline for all of the possible cases in this simple example, representing every event with a dot. Each dot represents an event which would occur if we were to run a conventional simulation using the input data pattern to the left of that line. We join together events which form clusters with dotted lines.

Each event in a cluster specifies a change on the same node, but at a different timepoint for each input pattern. Therefore, we can potentially compute the resultant effects on fanout nodes once for each cluster, rather than recomputing it for each unique event.

## 3.1 Cluster-Queues

Unfortunately, event clusters can interact with each other such that one cluster changes the state of the network during the period covered by another. In conventional timing simulation, we sort the event queue such that earlier events are always executed before later events, thus guaranteeing that the network state is up-to-date. However, since clusters span time ranges, it is not always possible to impose a total ordering based on time.

Consider the example in Figure 2, containing the same skewed inverter but with an edge-triggered flip-flop connected to its output. If the clock transitions high at time 150ps, it should sample the old value for case $\overline{a}b$, and the new value otherwise. Clearly, if we were to execute the event cluster on $out$ before the cluster on $clk$, we would sample the new value for all cases, which is incorrect. Likewise, if we executed the cluster on $clk$ before the cluster on $out$, we would sample the old value for all cases, which is also incorrect. In this case, the only alternative is to split one of the clusters so that the proper ordering can be maintained.

To be clear about how this splitting takes place, we must first formally define a cluster:

$$\begin{aligned} C_A &= \langle N_A, V_A, T_A \rangle \\ N_A &: \text{Node} \\ V_A &: \text{Node value function} \\ T_A &: \text{Set of time-function pairs } \{(t_A^i, m_A^i)\} \\ & \quad t_A^i: \text{ scalar time value} \\ & \quad m_A^i: \text{ mask function for } t_A^i \\ & \quad \text{Note: } \forall i \neq j((m_A^i \wedge m_A^j) = 0) \end{aligned}$$

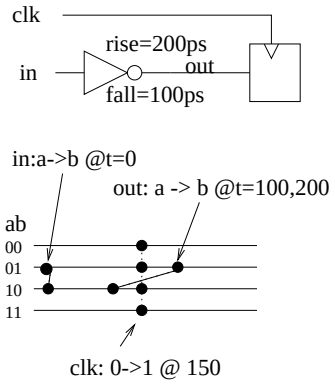


**Figure 2: Event Cluster Interactions**

Based on this definition, we can define a relation $\prec$, such that cluster A can be safely executed before cluster B if and only if $C_A \prec C_B$.

$$C_A \prec C_B \iff \forall i,j\left[(t_A^i < t_B^j) \vee \overline{(m_A^i \wedge m_B^j)}\right]$$

This states that $C_A \prec C_B$ if and only if $C_A$ occurs earlier than $C_B$ for all cases in which they are both defined.

A *cluster queue* is an ordered list of clusters such that:

$$Q = \{C_1, C_2, \cdots, C_k\}$$
$$\forall i < j (C_i \prec C_j)$$

Note that this definition of correctness requires $k^2/2$ comparisons to verify. At the cost of additional splitting, we can reduce the computations required to insert a new cluster into this queue if we define the additional function:

$$mustPrecede(C_A, C_B) = \bigvee_{i,j}(m_A^i \wedge m_B^j \wedge (t_A^i < t_B^j))$$

In other words, $mustPrecede$ returns the Boolean function containing all input assignments for which clusters $C_A$ and $C_B$ are both defined, and in which $C_A$ must occur before $C_B$.

During insertion into a cluster queue, we may be required to split the new cluster so that the above ordering conditions can be met. To compute the portion of cluster $C$ that intersects with an additional masking function $F$, we introduce the function $Chop(C, F)$, which returns a copy of cluster $C$ where all events that do not intersect $F$ have been removed.

$$\begin{aligned} C_{chop} &\leftarrow Chop(C, F) \\ C_{chop} &= \langle N_{chop}, V_{chop}, T_{chop}\rangle \\ N_{chop} &= N_C \\ V_{chop} &= V_C \\ T_{chop} &= \{(t_{chop}^i, m_{chop}^i)\} \\ & \quad t_{chop}^i = t_C^i \\ & \quad m_{chop}^i = m_C^i \wedge F \end{aligned}$$

When inserting a new cluster $C_A$ into the queue $C_1, \cdots, C_k, k \geq 1$, we will initially split $C_A$ into at most $k+1$ pieces that will be inserted between each of the existing clusters:

$$Q = \{C_{A1}, C_1, C_{A2}, C_2, \cdots C_{Ak}, C_k, C_{A(k+1)}\}$$

```
1   EnqueueCluster( Cluster new, ClusterQueue Q )
2       if( Q empty )
3           Q ← new
4           return
5       Done ← false
6       TBD ← new
7       foreach cluster c in Q
8           MP ← mustPrecede(TBD, c)
9           if( MP = true )
10              insert TBD before c
11              return
12          else if (MP = false )
13              next c
14          P ← Chop(TBD, MP)
15          insert P before c
16          TBD ← Chop(TBD, ¬MP)
17          Done ← Done + MP
18          if( Done = true )
19              return
20      append( TBD, Q )
21
22  Cluster DequeueCluster( ClusterQueue Q )
23      ret = head(Q)
24      Q ← Q − ret
25      return ret
```

**Figure 3: Cluster Queue Operations**

We can recursively compute each of these partial clusters as follows:

$$C_{Ai} = Chop(C_A, P_i), i = 1 \cdots (k+1)$$

where,

$$P_i = mustPrecede(C_A, C_i) \wedge \bigwedge_{j<i} \overline{P_j}, i = 1 \cdots k$$

$$P_{k+1} = \bigwedge_{j=1}^{k} \overline{P_j}$$

Note that in implementing this insertion procedure, we need not insert NULL events (for which $P_i = 0$).

Pseudocode for the cluster enqueue and dequeue operations are shown in Figure 3. EnqueueCluster scans through the queue, splitting off and inserting each portion of the new cluster as required. If $MP$ contains the constant function $true$ in line 9, we can insert the entire remaining cluster and terminate. If $MP$ is $false$, there is no interaction and we can move on to the next cluster. Lines 14-15 enqueue the portion that must be inserted before the current cluster, and lines 16-19 remove that portion from the original and keep track of what's been done so far. If we reach the end of the queue, any remaining portion of the cluster is simply appended. DequeueCluster simply removes the head of the cluster queue and returns it.

Two example cluster insertions are shown in Figure 4. In part (a), the cluster $C_{new}$ can be inserted completely before $C_2$. Despite the fact that the first (leftmost) event in $C_2$ is earlier than the first event in $C_{new}$, $C_{new}$ precedes $C_2$ for all input assignments for which they are both defined. In part (b), we see that $C_{new}$ must be split into 3 parts to preserve a consistent ordering. Notice that only those portions that must precede $C_1$ and $C_2$ are split off, and the remainder is grouped together at the end of the queue.

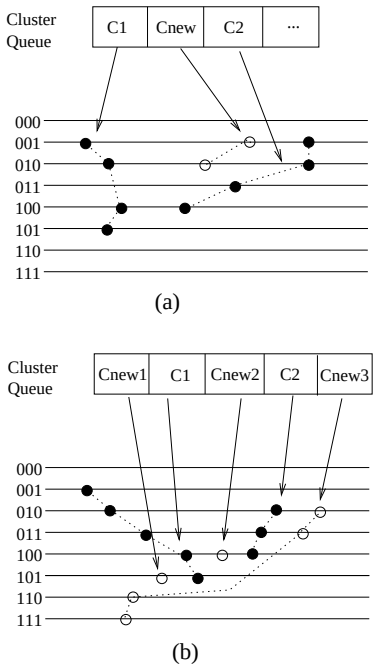


**Figure 4: Cluster Queue Insertion**

## 3.2 Cluster Scheduling

It is possible to construct a data-dependent symbolic timing simulator with a single cluster-queue for all events. However, upon doing so, we discovered it to be highly inefficient due to interactions between independent clusters. This results in the majority of clusters being split several times to maintain proper ordering, which fails to alleviate the event-multiplication problem. Even worse, it incurs the additional overhead of managing the cluster-queues.

The solution lies in realizing that a strict time ordering across the entire circuit is not required. We simply need to make sure that all circuit state which can affect the computation of a certain event has been accounted for when that event is executed.

For clarity, we will describe the case of a gate-level logic network, but the same principles apply to transistor-level networks subdivided into channel-connected regions. For a cluster of events $C$ to be ready for execution at the input to gate $G$, we must satisfy the following conservative safety conditions:

1. There may be no pending events on the other inputs to $G$ that precede events in $C$.
2. There may be no pending events on the output of $G$ that precede events in $C$.
3. There may be no pending events upstream of the other inputs to $G$ that could propagate to $G$ before the completion of $C$.

Conditions 1 and 2 can be satisfied by creating one cluster-queue for each gate. Events are scheduled into the cluster-queue for the driving gate and all receiving gates. When events are dequeued from the driving gate's cluster-queue, only node state is updated. When events are dequeued from the receiving gate's cluster-queue, node state is updated and the effects of that node transition are computed in the receiving gate. Consequently, each node has multiple current states, one for each gate it is connected to. We denote the value of node $N$ relative to gate $G$ as $V_N^G$.

Condition 3 requires knowledge of the minimum propagation delay between any two gates. If the output of one gate cannot propagate to the output of another, the minimum propagation delay is defined as $\infty$. Once the minimum delays are known, we can guarantee safety if, for all other events $O$:

$$safe(C) \leftarrow \forall O(Earliest_O + MinDelay_{O,G} > Latest_C)$$

$$where$$

$$Earliest_O = min_i(t_O^i)$$

$$Latest_O = max_i(t_O^i)$$

The minimum propagation delay between any two nodes can be computed by determining a conservative minimum delay per gate, and computing All-Pairs-Shortest-Paths via the Floyd-Warshall or Johnson's algorithm [6]. Johnson's algorithm requires $O(V \cdot E \cdot lgV)$ time, which is $O(V^3 \cdot lgV)$ for a fully-connected graph. However, since the maximum fanout of each gate is limited to a small integer, usually $\approx 4$, the number of edges is effectively $O(V)$, bringing the overall complexity to $O(V^2 \cdot lgV)$. Furthermore, this computation can be done as a pre-processing step and re-used from one simulation run to the next.

If no pending clusters satisfy the safety conditions, it is always safe to effectively revert to the non-cluster methodology by splitting off the earliest event contained in all clusters. While this hurts efficiency, it guarantees forward progress. Hopefully, by removing several singular events, we will again be able to guarantee the safety of one or more clusters and return to full-speed operation.

The improved cluster scheduling algorithm is presented in Figure 5. It makes use of the *EnqueueCluster* and *DequeueCluster* operations from Figure 3.

Simulate() forms the main body of the simulator. As long as pending events remain, it obtains them via GetNext(). Recall that each event cluster is scheduled into the cluster queue for its driving gate and for all receivers. If the cluster returned by GetNext() was from its driver's queue, only that node state is updated. However, if the cluster was removed from a receiver's queue, then the results of that node transition in the receiving gate are computed and scheduled as a future event.

GetNext() is responsible for finding an event-cluster that can be safely executed. It first checks if any of the clusters at the heads of any of the gate queues are "safe", and returns them if they are. Otherwise, it creates a new event cluster containing only the earliest event present in all clusters. That earliest event is then deleted from the original cluster by eliminating that case from its mask.

Safe() checks condition 3 specified above. It visits event clusters in order of increasing earliest timepoints, which allows for the early termination case in line 9. If it finds any event which could arrive at $G$ before $C$ completes, it returns false. If it finds no clusters which could conflict, it returns true.

Schedule() simply schedules an event cluster into the cluster-queues for all gates connected to $N_C$.

# 4. IMPLEMENTATION

The algorithms from the preceding section have been implemented in STEED (Symbolic Timing Engine for Electronic Design), a transistor-level symbolic timing simulator. STEED computes data-dependent

```
1   Schedule( Cluster C )
2       foreach Gate G connected to N_C
3           EnqueueCluster( C, Q_G )
4
5   Boolean Safe( Cluster C, Gate G )
6       ∀O ∈ Q_G sorted by increasing earliest timepoint
7           if( Earliest_O + MinDelay_{O,G} < Latest_C )
8               return false
9           if( Earliest_O > Latest_C )
10              return true
11      return true
12
13  ⟨Cluster, Gate⟩ GetNext()
14      foreach Gate G
15          if( Safe( head(Q_G), G ) )
16              return ⟨ DequeueCluster(Q_G), G⟩
17      find Cluster E that contains earliest event
18      m ← m^i_E, such that t^i_E = Earliest_E
19      E' ← Chop(E, m)
20      E ← Chop(E, m̄)
21      return E'
22
23  Simulate()
24      while( ⟨C, G⟩ ← GetNext() )
25          V^G_N ← V_C
26          if( N_C ∈ inputs(G) )
27              N_Z ← output(G)
28              V_Z ← new value for N_Z
29              T_Z ← output transition time of N_Z
30              Z ← ⟨N_Z, V_Z, T_Z⟩
31              m ← V^G_{output(G)} ⊕ V_Z
32              Schedule( Chop(Z, m) )
```

**Figure 5: Symbolic Simulation Using Cluster Scheduling**

delays based on the Elmore approximation (RC products), utilizing the methodology implemented in IRSIM[5]. It borrows the node value and delay-calculation engines from SirSim [9], an earlier symbolic timing simulator not based on cluster scheduling.

As transistor-level simulators, both STEED and SirSim operate on channel-connected regions (CCRs) rather than on gates. An input to a CCR is any node connected to the gate of a transistor in that CCR. An output of a CCR is any node connected to the source or drain of a transistor in that CCR. Note that internal nodes are considered outputs for the purposes of the scheduling algorithms.

When an input to a CCR changes, the new value of all output nodes is computed based on voltage-divider or on charge-sharing models. This new value is returned as a BDD, representing the Boolean function of that node relative to the applied input variables.

Delays are computed symbolically using Multi-Terminal Binary Decision Diagrams (MTBDDs)[1]. MTBDDs are an extension of BDDs that may contain an arbitrary number of real-valued terminal nodes, rather than the binary terminals 0 and 1. The delay MTBDDs returned by the delay calculation engine encode the delay value due to the current node transition, as well as the mask information. Recall that the mask specifies under what logical conditions the output transition will occur. If no transition will occur, this is encoded as an infinite delay value in the delay MTBDD.

The delay MTBDD for the skewed inverter case is shown in Figure 6. To interpret this MTBDD, we follow the solid arc when the

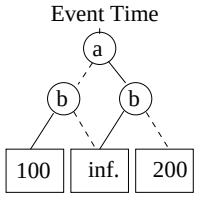


**Figure 6: Event Time MTBDD**

**Table 1: Runtimes**

| | SirSim | | | STEED | | | |
|---|---|---|---|---|---|---|---|
| Name | MB | sec. | Ev | MB | sec. | Ev | SU |
| adder16 | 22 | 20 | 4015 | 5.5 | 5.2 | 2802 | 3.5 |
| adder32 | 102 | 107 | 15194 | 17 | 22 | 6525 | 4.8 |
| adder64 | 280 | 784 | 58692 | 61 | 147 | 16459 | 5.3 |
| byp_add16 | 33 | 26 | | 13 | 14 | 4013 | 2.0 |
| byp_add32 | 250 | 213 | | 241 | 397 | 19267 | 0.5 |
| adder32.r | 230 | 207 | 15616 | 36 | 39 | 6832 | 5.3 |
| adder64.r | 298 | 2078 | 61289 | 188 | 267 | 17331 | 7.8 |
| s298 | 2.7 | 2.2 | 2350 | 2.5 | 3.0 | 3178 | 0.9 |
| s382 | 7.2 | 10.0 | 5485 | 6.4 | 9.0 | 3178 | 1.1 |
| s444 | 6.4 | 9.5 | 6324 | 7.1 | 11.6 | 7017 | 0.8 |

associated variable is true, and the dashed arc when it is false. In this case, we see that no event will occur (delay = $\infty$) when both $a$ and $b$ are true, or when both are false. When $a$ is true and $b$ is false, we get a rising transition on the output having a delay of 200. When $a$ is false and $b$ is true, we get a falling transition on the output having a delay of 100.

All of the operations in the cluster-queue and cluster-scheduling algorithms can be implemented easily using standard MTBDD functions. The CUDD 2.2.0 [11] decision-diagram package was used for all symbolic computation.

## 5. RESULTS

STEED was run for most of the benchmarks reported in [9] as well as several others. STEED achieved a speedup in nearly every case, up to 7.8X. Table 1 shows runtimes, peak BDD memory usage, and event counts for STEED and Sirsim, as well as the speedup attained. Recall that, in STEED, events are inserted into the queues for both driver and receiver, increasing their apparent number. The effective event count is typically about half the total number which appears in Table 1.

The lowest speedups occur on the carry-bypass adders, byp_add16 and byp_add32. These test cases contain especially large channel-connected regions in their carry-chains, meaning more nodes and events per cluster-queue. When large numbers of events are inserted into a single cluster-queue, fragmentation of the events in-

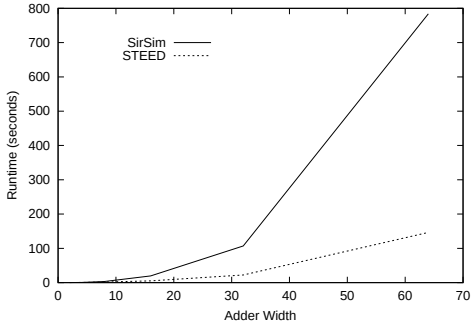


**Figure 7: Comparison of Adder Runtimes**

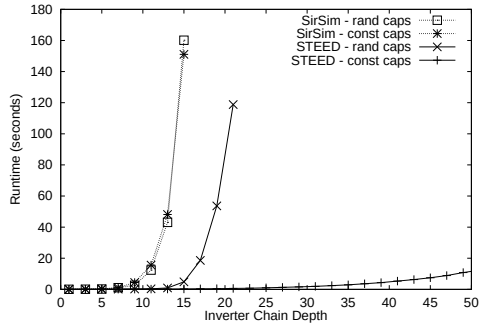


**Figure 8: Inverter Chain Performance**

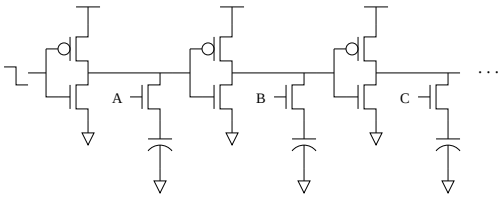


**Figure 9: Inverter Chain with Data-Dependent Load Elements**

creases, and the computational overhead of queue management becomes costly. These test cases suggest that a hybrid scheduling mechanism might be beneficial.

On the smaller combinational benchmarks, we see very little difference between SirSim and STEED. Apparently, the shallower logic cones ($O(\log n)$ versus $O(n)$ for the adders), mitigate the event multiplication problem, such that cluster-queue management offsets the gains in event count.

STEED's performance on the standard adders is exceptional, obtaining a speedups as high as 5.3X. As shown in Figure 7, STEED's speedup over SirSim grows with the size of the adder.

In order to demonstrate a further advantage over non-cluster-based scheduling, we generated two test-cases, adder32.r and adder64.r, with additional small randomized capacitance values on every node. These capacitances break up much of the symmetry in delay values on multi-input gates. For SirSim, this aggravates the event multiplication problem, increasing the time required to complete the simulation. However, for STEED, the event time MTBDDs simply grow a little larger, impacting performance less profoundly. The result of these additional capacitances is a larger speedup due to cluster scheduling, reaching 7.8X for the 64-bit circuit. Furthermore, these test cases model the real-life situation in which back-annotated capacitance values are included in the simulation, especially when the circuit being simulated has irregular routing above it.

To isolate the effects of event multiplication, we constructed increasing lengths of inverter chains with data-dependent load elements (Figure 9). Under SirSim's scheduling algorithm, a transition on the input will generate two possible delay values through the first stage, dependent on the state of signal A. At each successive stage, the number of events at the output doubles, resulting in an exponential number of events. This is responsible for the exponential runtime shown by the top two lines of Figure 8.

STEED encodes the events on each stage's output as an event cluster. If the capacitors connected to the data-dependent load elements are randomized, there will still be a unique delay case for every possible input assignment, so the cluster event-time MTBDDs ($T_C$) will grow exponentially. Note however, that the exponential runtime is of lower order than in SirSim. If the load-element capacitors are are constant or take on some small number of distinct values, the cluster event-time MTBDDs have a mesh structure and quadratic size. In this case STEED achieves polynomial runtime, as shown by the bottom line on Figure 8.

# 6. CONCLUSION

We have identified a major bottleneck in the data-dependent scheduling mechanism found in SirSim, which we term the event multiplication problem. To address this issue, we introduced an event-management methodology based on event clusters, and implemented it in the simulator STEED. We obtained a substantial speedup in nearly all test-cases attempted, demonstrating the advantages of cluster-based scheduling.

# 7. REFERENCES

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. *ACM/IEEE International Conference on Computer Aided Design*, pages 38–43, October 1992.

[2] R. E. Bryant. Symbolic Verification of MOS Circuits. *1985 Chapel Hill Conference on VLSI*, pages 418–438, 1985.

[3] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):79–85, August 1986.

[4] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, CAD-6(4):634–639, July 1987.

[5] C. Y. Chu. *Improved Models for Switch-Level Simulation*. PhD thesis, Stanford University, October 1988.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1997.

[7] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Certified Timing Verification and the Transition Delay of a Logic Circuit. *Proceedings of the Design Automation Conference*, 1992.

[8] I. Hajj and D. Saab. Symbolic Logic Simulation of MOS Circuits. *International Conference on Circuits and Systems*, pages 249–249, 1983.

[9] C. B. McDonald and R. E. Bryant. Symbolic Functional and Timing Verification of Transistor Level Circuits. *ACM/IEEE International Conference on Computer Aided Design*, 1999.

[10] A. Salz and M. A. Horowitz. IRSIM: An Incremental MOS Switch-level Simulator. *Proceedings of the Design Automation Conference*, pages 173–178, June 1989.

[11] F. Somenzi. *CUDD: CU Decision Diagram Package - Release 2.2.0, Online User Manual*, May 1998.