Speeding up Look-up-Table Driven Logic Simulation

Rajeev Murgai and Masahiro Fujita
Fujitsu Laboratories of America, Inc.,
Sunnyvale, CA, USA
murgai@fla.fujitsu.com, fujita@fla.fujitsu.com

Fumiyasu Hirose
Cadence Design Systems, Inc.
Yokohama, Japan
hirose@cadence.com

Abstract Logic simulation is one of the most important steps in the design of a digital circuit. Due to the growing complexity of the designs, a large number of test vectors is needed, making simulation a big bottleneck. One way to speed up logic simulation is by designing distributed or parallel hardware architectures that are optimized for simulation. One such hardware accelerator is Fujitsu's TP5000, which was shown to be over 300 times faster than a state-of-the-art software simulator on large circuits. TP5000 uses a memory-based, event-driven simulator [9]. In this paper, we propose logic restructuring techniques to further speed up functional simulation on TP5000. The key idea is to generate a perfectly balanced network logically equivalent to the original network that fits in the TP5000 memory. A perfectly balanced network gets rid of useless evaluations. We use logic decomposition, partial collapsing, and buffer insertion to generate such a network. Experimental results indicate that our techniques reduce the number of events in TP5000 by 33% as compared to a commonly-used technique that optimizes the network and does a straightforward mapping on the simulator, and by 22% as compared to the technique of [7]. On some benchmarks, the reduction is by a factor of 3.

Keywords: Logic simulation, events, logic synthesis.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35498-9_57
L. M. Silveira et al. (eds.), *VLSI: Systems on a Chip*

1. INTRODUCTION

Logic simulation is one of the most important steps in the design of a digital circuit, and is used to verify the correctness of the designed circuit. Test inputs are applied to see if the circuit responds to these inputs as it was supposed to. For large circuits, typically millions of such inputs are applied. Simulation can be done at either behavioral, logic, or circuit level. Also, it can be performed either in **software** or **hardware**. In software simulation, models for the basic circuit components or **primitives** (such as inverter, two-input AND, two-input OR gates in logic level simulation) are used. An AND gate, for instance, is modeled either with its truth table or using the bit-wise AND instruction of the processor on which the software runs. The circuit is converted into these basic primitives by the simulator. These primitives are evaluated either in a statically determined, topological order starting from the inputs (**compiled code simulation**) or dynamically in an **event-driven** manner. Since these primitives are two-input gates, their number is very high. This, along with a large input-vector set and the software nature of testing, makes the simulation process rather slow. **Hardware accelerators** have been proposed to speed up the simulation process [1, 3]. The circuit to be simulated is mapped on to the **hardware primitives** of the accelerator that are designed specifically for simulation and have short evaluation times. In addition, accelerators such as Fujitsu's Thread Processor TP5000 [9] divide the algorithm into fragments that can store and access data independently, and can be pipelined, thus achieving a higher throughput.

In this paper, we focus on an event-driven, logic simulator implemented on TP5000. This simulator was shown to be 300 times faster than the state-of-the-art software simulator (Cadence's Leapfrog^{TM}) [9] on circuits having more than half a million gates. Our goal in this work is to further speed up the TP5000 simulator. The paper is organized as follows. Section 2 describes the working of the TP5000 simulator. The problem statement and related work are in Section 3. Section 4 establishes a connection between useless evaluations in the simulator and glitches in a circuit under unit delay model. This leads us to propose synthesis techniques for removing glitches, specifically for generating a perfectly balanced network under the TP5000 memory constraint; these are described in Section 5. In Section 6, we present experimental results that demonstrate effectiveness of the proposed techniques in reducing the number of events during simulation. The conclusions and possible extensions are presented in Section 7.

2. USING TP5000 FOR SIMULATION

Given a design (network, circuit) to be simulated on TP5000, the first step is to map it onto the logic primitives provided by TP5000. One such logic primitive is a truth table L, which is actually the memory associated with one of the TP5000 processors. L has an 18-bit wide address bus and hence a **memory capacity** $M = 2^{18}$. L can be partitioned arbitrarily into smaller chunks, each chunk implementing a logic function. For example, L can implement either one function of 18 inputs, or one function of 17 inputs and two functions of 16 inputs, etc. As illustrated in Figure 1 (a), the truth tables of two functions $f(x_1, x_2, x_3)$ and $g(y_1, y_2)$ are stored together in the memory. Given an input combination, the value of the function (f or g) can be read from the appropriate location of the memory. Mapping of a design on TP5000 involves generating a circuit, all of whose functions can be packed together in L. This typically entails some restructuring of the design.

After the design is mapped, TP5000 starts simulating it on a set of test vectors. The event-driven simulation algorithm implemented on top of TP5000 [9] evaluates one logic function at a time. Its operation is shown in Figure 1 (b). When a new test vector is applied, an **event** (a transition from 0 to 1 or from 1 to 0) is recorded at each primary input that changed its value from the last vector. The logic functions to which such primary inputs fan out are put into the

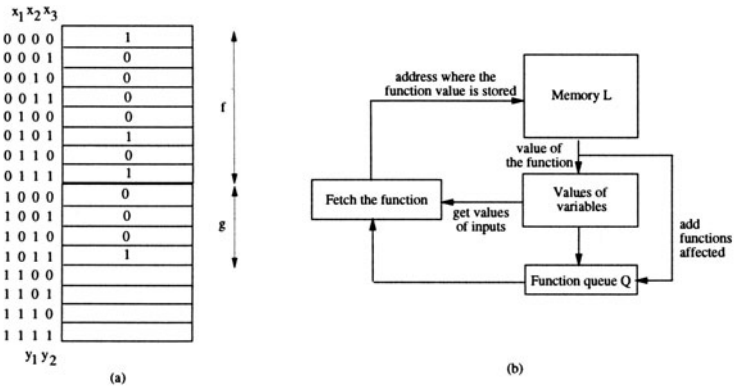

Figure 1 (a) Storing two functions f & g in an SRAM with $M = 16$; (b) Working principle of the event-driven simulator of TP5000

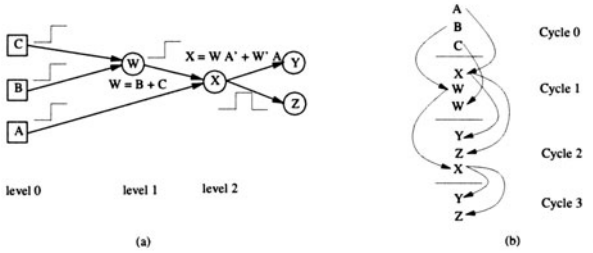

Figure 2 (a) Event-driven simulation of TP5000 in action; (b) Causality among evaluations

function queue Q for **evaluation**. *Evaluating a function f means determining the value of f, given the values of its fanins and its representation (i.e., the truth table).* The first function, say f, is fetched from the head of the queue Q for evaluation. The values of fanins of f determine the memory address that should be accessed for evaluating f. If the value stored at this location is different from the previous value of f, the value of f is updated and an event is recorded at f, in which case the fanouts of f are inserted at the end of Q for future evaluation. Evaluation of a circuit output does not cause further evaluations. This process continues until Q is empty. The following example illustrates the event-driven simulation of TP5000.

Example 1.1 *Figure 2 (a) shows a part of a network η with A, B, and C as primary inputs. The truth tables of W, X, Y, and Z are stored in the memory L of* TP5000. *Let $(A, B, C) = (0, 0, 0)$ initially, resulting in $W = X = 0$. Now, $(A, B, C) = (1, 1, 1)$ is applied. First, all the primary inputs A, B, and C are placed on the queue Q. A being at the head of Q is evaluated first. An event occurs since A goes from 0 to 1. Hence the fanout X of A is placed at the end of Q. Figure 2 (b) shows the causality between various events and evaluations (ignore the cycle numbers). For instance, the arrow from A to X means that an event took place on A, causing an evaluation of X. Now $Q = (B, C, X)$. Next B and then C are evaluated, causing W, the fanout of B and C, to be placed on Q, which becomes (X, W, W). Note that one input vector can cause a function (in this case W) to be evaluated more than once. Next X is fetched from Q. Its evaluation uses the updated value of A (i.e., 1) and $W = 0$. As a result, $X = WA' + W'A$ switches from 0 to 1. The new value of X is*

recorded and an event is generated at X. X's fanouts, Y and Z, are placed at the end of Q, which becomes (W, W, Y, Z). W is fetched next. Using $B = C = 1$, $W = B + C = 1$. This is an event at W, causing X to be put on Q. Now $Q = (W, Y, Z, X)$. The next evaluation of W does not result in any events, since W's fanins have not changed since W was evaluated last.

As mentioned earlier, not all evaluations cause further evaluations. This happens when the value of a node does not change. For example, the second evaluation of W.

Two-valued vs. four-valued simulation: In a two-valued simulation, each signal can be either 0 or 1. Then an n-input function takes 2^n bits in the memory. TP5000 actually implements a four-valued simulation, where each signal can take values 0, 1, X, or Z. X means unknown and Z means high impedance. Two bits are needed to encode a signal. An n-input function then uses $2 \times 4^n = 2^{(2n+1)}$ bits of storage. The factor 2 refers to the two-bit output value, and 4^n is the total number of input combinations. In our experiments, the memory capacity $M = 2^{18}$. So no mapped logic function can have more than 8 inputs.

3. SYNTHESIS FOR SIMULATION

Assume we are given a network η that is to be simulated on TP5000 for functional correctness. Since we are interested only in functional correctness (and not in timing correctness or glitch analysis), we can restructure η (i.e., change its intermediate logic functions) as long as its global logic functionality does not change. Since different structures can result in different simulation times on a given set of simulation vectors, **our goal is to transform η into $\tilde{\eta}$ such that $\tilde{\eta}$ is logically equivalent to η, $\tilde{\eta}$ fits in L using the available memory M, and the total simulation time for $\tilde{\eta}$ on a given set of vectors is minimized.** One constraint was that we could not change the simulator. Examine the requirement of minimizing the total simulation time. The simulation time depends on the number of function evaluations and the time taken for each evaluation. Since a function evaluation involves a memory read, the time taken for an evaluation is essentially independent of the complexity of the function. So the requirement reduces to minimizing the number of evaluations. Since each function evaluation corresponds to an event at an input of the function, we wish to minimize the number of events in the network. It is not easy to model the number of events. Since simulation is serial in TP5000 (i.e., only one function is evaluated at a time), the following approximation was made in [7]: *model the total number of events by the number of functions (primitives) in the network mapped on L.* Thus, the goal was to minimize the number of functions in the network subject to the memory constraint M. Decomposition and partial collapsing techniques were investigated. The following example illustrates partial collapsing.

Example 1.2 *In Figure 2 (a), the total space used by W and X is 8 bits (assuming 2-valued simulation). Collapsing W into X, we obtain a new three-input function at X, which also uses 8 bits. This transformation does not increase the memory usage and also reduces the number of network primitives by one. So it will be applied in [7].*

In [7], it was observed empirically that a 2-input decomposition followed by partial collapsing targeted for minimum primitive-count subject to the memory constraint yielded the minimum number of primitives. This technique (which we shall call min-fn) reduced the number of primitives of many benchmarks by a factor of 2 to 4. Although the number of events also decreased significantly, it did not go down in the same proportion as the primitives. *In this paper, we will describe techniques that target minimizing the number of events directly instead of the number of primitives.*

To speed up cycle-based simulation, a decision-diagram based method was proposed by McGeer *et al.* [4]. Although their technique is not for event-driven simulation, our problem and

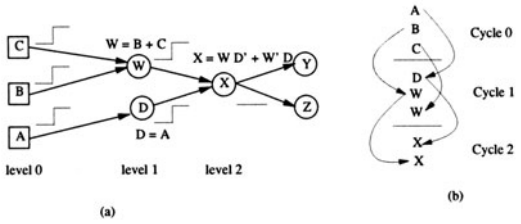

Figure 3 (a) Adding a buffer to reduce events, (b) Evaluations after buffer insertion

their problem are similar in essence. The solution techniques are, however, different. Whereas McGeer *et al.* use partitioned decision diagrams to represent logic functions and circuits, the architecture of TP5000 forces us to store logic functions as truth tables in the memory. Also, while their method is based on evaluating a node of the decision diagram, in our technique evaluating a complex logic function is an atomic operation – a memory read, which can lead to potential speed-ups in simulation. A direct comparison is, however, not possible, since the simulation targets are completely different.

4. USELESS SIMULATOR EVALUATIONS AND GLITCHES IN THE CIRCUIT

In Figure 2 (a), each node N is assigned a **level** $\ell(N)$ as follows. $\ell(N) = 0$ if N is a primary input. Otherwise, $\ell(N) = 1 + \max_{i \text{ fanin of } N}\{\ell(i)\}$. $\ell(A) = \ell(B) = \ell(C) = 0$ and $\ell(W) = 1, \ell(X) = 2$. Similarly, we can divide evaluations into **cycles**. All primary input evaluations happen in cycle 0. An evaluation directly caused by an evaluation (event) in cycle i is placed in cycle $(i + 1)$. In Figure 2 (b), the evaluations of A, B, and C are placed in cycle 0. The evaluation of A causes an evaluation of X, which is placed in cycle 1. Note that *the latest cycle in which a node can be evaluated is the same as the level of the node.*

In Figure 2 (b) X was evaluated twice, in cycles 1 and 2. The cycle 1 evaluation was caused by the event at A in cycle 0, and the cycle 2 evaluation by the event at W in cycle 1. Both evaluations resulted in events at X and caused the fanouts Y and Z to be evaluated, once in cycle 2 and then in cycle 3. The cycle 2 evaluations of Y and Z are **useless** for functional simulation, since Y and Z will be evaluated in cycle 3 in any case. If we could reduce the number of events at X from two to one, Y and Z will not be evaluated twice, thus reducing the total number of evaluations and the simulation time. This can be done, for instance, by *delaying the arrival of the signal A at the input of X by one cycle* by inserting a buffer D between A and X, as shown in Figure 3 (a). The resulting evaluations and events are shown in Figure 3 (b). The event at A reaches X in cycle 1. So X is evaluated only in cycle 2, although twice (which is same as before): once due to the event at D and then due to the event at W. However, when X is evaluated the first time in cycle 2, both D and W have already switched from 0 to 1. The new value of $X = WD' + W'D = 0$, the same as before. The second evaluation also does not change the value of X, since the fanins of X did not change values since the first evaluation. Thus, Y and Z are not evaluated at all, thus saving 4 evaluations. Recall that in Figure 2 (b), each of them was evaluated twice. However, we did incur an extra transition at the buffer D. Thus, the net reduction in the number of evaluations is 3.

In this analysis, we did not consider transitive fanouts of Y and Z. Had we, the savings in evaluations would have been potentially higher. Also note that even if X were not an XOR, at most one event could have occurred at X, still one less than Figure 2.

The above situation can be viewed in terms of **glitches** in the circuit. *Glitches are spurious transitions (events) at the outputs of gates before they settle down to their final values.* Since a glitch at a gate g can propagate to its transitive fanout gates, it causes useless evaluations of g and its transitive fanouts. If we can get rid of the glitches, we will get rid of most of the useless evaluations in an event-driven simulator, thus speeding up simulation. This is what we did with X, by removing the glitch in cycle 1. *We would like to synthesize a network that does not have glitches.*

One way to get rid of glitches is to obtain a **perfectly balanced network**. *A perfectly balanced network is one in which all the fanins of each node (or gate) are at the same level.* Thus all the paths to a node are of the same length (we call such a node a **perfectly balanced node**). This ensures that each node N is evaluated at most only in cycle $\ell(N)$, where $\ell(N)$ is the level of node N. Although N may be evaluated as many times as the number of fanins it has, only at most one evaluation can lead to an event. Next, we describe synthesis techniques for obtaining a perfectly balanced network.

5. SYNTHESIZING A PERFECTLY BALANCED NETWORK

Logic synthesis consists of two phases: **optimization** followed by **mapping**. In optimization, a minimal representation of the logic network is sought for area, delay, or power. In mapping, the optimized circuit is mapped on to a target technology (e.g., standard-cell library, FPGAs) minimizing some cost function. In this paper, we focus only on the mapping problem. In the mapping problem for the TP5000 processor, *the target technology is the processor memory and the cost function is the simulation time* (approximated by the number of evaluations or the number of events). Thus the goal is to convert a given network into a logically equivalent network that fits in the processor memory and has minimum number of evaluations or events. Intuitively, since perfectly balanced networks are free of useless events, we will restrict our problem to that of generating a perfectly balanced network under the memory constraint.

5.1 BUFFER INSERTION

Given an arbitrary network η, a simple and effective way to make it perfectly balanced is by inserting buffers appropriately at the primary inputs and gate outputs of η. The following example illustrates the idea.

Example 1.3 *Figure 4 (a) shows part of a network, with node N, at level 10, fanning out to nodes P and Q at levels 12 and 14 respectively. Then, add a series chain of 3 buffers at the output of N, where the final buffer feeds Q. This is shown in Figure 4 (b). Each buffer corresponds to one level. This ensures that if there is a path from a primary input to the output of Q that goes through N, it is of length 14. If each fanin of Q is buffered appropriately in the same manner, all paths to the output of Q will be of length 14. Then Q will be perfectly balanced. Note that the connection (N, P) is replaced by the connection from the output of the first buffer after N, thus saving buffers. The corresponding input of P arrives at time unit 11.*

The algorithm to make a network η perfectly balanced by buffering is straightforward and follows from Example 1.3. First levelize η, i.e., compute $\ell(N)$ for each node N of η. Traverse η such that a node is visited only after all its fanins. Let the current node be N. Let $FO(N)$ denote the set of fanouts of N. Compute $\ell_{max} = \max_{f \in FO(N)}\{\ell(f)\}$. Add $\ell_{max} - 1 - \ell(N)$ buffers at the output of N. This makes all the paths to the output of each fanout f of N that go through N of length $\ell(f)$. Add these buffers in a series structure to the output of N. For each

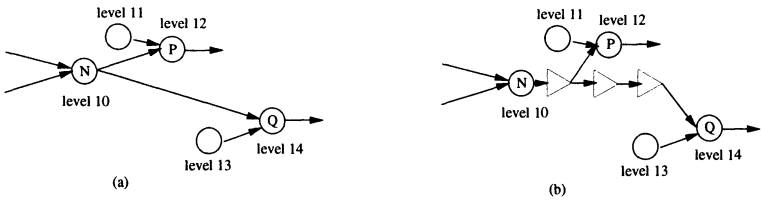

Figure 4 Adding buffers at N to balance path lengths for P and Q

$f \in FO(N)$, replace the (N, f) edge appropriately by an edge from a suitable buffer (as was illustrated in Example 1.3).

We must mention that buffering is a standard technique used in logic synthesis for reducing the delay due to fanout loading and for wave pipelining [11].

As we will see in Section 6, buffering is a powerful technique and works extremely well. However, each buffer comes with a cost, which is the extra transitions at its output. Therefore care must be taken before adding buffers, especially if the initial network is extremely unbalanced. In our approach, we use other synthesis techniques (such as decomposition and partial collapse) to balance the network as much as possible, and then use buffering as the last step to obtain a perfectly balanced network.

5.2 DECOMPOSITION

Decomposition is the process of expressing a given logic function in terms of simpler functions. If the goal is to have the fewest literals, a simple function should have fewer literals.

Example 1.4 *Let* $f = abc + abd + a'c'd' + b'c'd'$. *One way to decompose* f *is as follows.* $f = xy + x'y'$, $x = ab$, $y = c + d$. *An alternate way of decomposing* f *is:* $f = w + x + y + z$, $w = abc$, $x = abd$, $y = a'c'd'$, $z = b'c'd'$. *If fewest literals is the goal, clearly the first decomposition is preferable.*

We need decomposition in our application, because for instance, if there is a 9-input function f in the network, and the simulation is 4-valued, f will not fit in the memory of size $M = 2^{18}$. In our decomposition, we restrict each node function to have no more than k fanins, where k is a fixed constant. It was empirically found that $k = 6$ yields the best results (Section 6). If the resulting network is still too large to fit in the memory L, smaller values of k are used.

It follows then that in the context of TP5000, simplicity is determined by the total space the function takes up in the memory L. So one goal is to *decompose a function into sub-functions whose total space requirement is the minimum.* In Example 1.4, for a 2-valued simulation, the original f takes up 16 locations, the first decomposition (using XNOR) 4 + 4 + 4 = 12, and the second one 16 + 8 + 8 + 8 = 40 locations. Clearly, the first decomposition is the desired one for minimum memory.

We now come to the second goal: *the decomposition of a node should yield a perfectly balanced sub-network.* We first propose an algorithm that decomposes a multi-input AND node into a balanced network. It sorts the fanins of the node with the minimum-level fanin first. It *combines* inputs that have identical level until either k inputs have been combined or an input is reached whose level is strictly greater than the last input's. In either case, an intermediate node is added. In the second case, appropriate number of buffers may also be added to balance the fanin. We continue until all the fanins have been processed. The following example explains it.

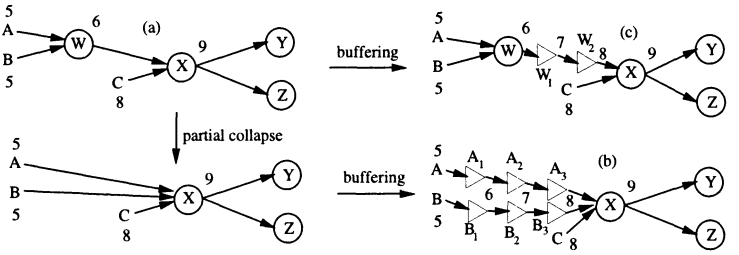

Figure 5 Collapsing vs. buffering: case 1

Example 1.5 *Let k be 4. Let $f = abcde$. Let $\ell(a) = \ell(b) = 1$, $\ell(c) = \ell(d) = \ell(e) = 2$. We first combine a and b (but not c, since $\ell(c) > \ell(a)$). This results in $x = ab$, $f = xcde$, with $\ell(x) = 2$. f need not be decomposed any further, since it satisfies the k constraint and the levels of all the fanins are identical. Both f and x are perfectly balanced and have at most k inputs.*

If $k = 3$ instead, still $x = ab$, $f = xcde$. However, since f has more than 3 fanins, it is decomposed further. Continuing the same procedure and combining inputs x, c, and d (all with level 2), we obtain node $y = xcd$, with $\ell(y) = 3$. Now $f = ye$. Node e has the minimum level. Since $\ell(y) \neq \ell(e)$, e is separated as $z = e$, and then $f = zy$. Note that z is a buffer, with $\ell(z) = 3 = \ell(y)$. Thus f is balanced. So the final decomposition is $x = ab$; $y = xcd$; $z = e$; $f = yz$. Each function has at most 3 inputs and is perfectly balanced.

Note that the procedure may use buffers ($z = e$ for $k = 3$) to ensure a perfectly balanced decomposition. However, it attempts to minimize the number of buffers. An OR node is decomposed similarly. An arbitrary node $f = c_1 + c_2 + \ldots c_p$ (where c_is are product terms) is first decomposed into p AND nodes, and an OR node at the top (f itself). The above procedure is first applied to the AND nodes one by one, the levels of the AND nodes are updated, and finally the procedure is applied to the root OR node. To apply this decomposition on a network, traverse the network topologically from primary inputs to outputs. This ensures that the levels of the fanins are correct when a node is visited and decomposed. This yields a perfectly balanced decomposition of the network.

5.3 PARTIAL COLLAPSE (ELIMINATION)

This step corresponds to tree-covering of conventional technology mappers. We start with the network obtained from the decomposition step. The goal is to transform the network by combining nodes so that the resulting network has potentially fewer evaluations.

Example 1.6 *Consider the network of Figure 2. We proposed earlier that one way to get rid of the glitch at X is to add a buffer at A. Another way is to collapse W into X. Then all the three fanins to X will have the same level 0 and X will be perfectly balanced. However, if W were fanning out to other nodes besides X, collapsing W into X alone may not help. Any events on B and C will still cause W to be evaluated. At the same time, B and C will cause their new fanout X to be evaluated as well. This means a possible increase in the number of evaluations and events.*

The last example underscores the need for deciding if we should collapse or wait until the end to add the required number of buffers, in order to balance the node. We explain how we resolve this by means of an example.

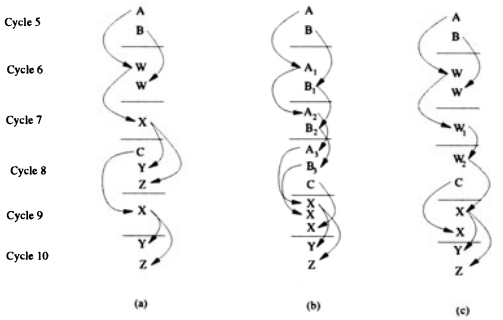

Figure 6 Collapsing vs. buffering: causality

Figure 7 Collapsing vs. buffering: case 2

Example 1.7 *Consider a perfectly balanced node W in Figure 5 (a); the numbers represent the node levels. Let W be collapsed into X. X is not perfectly balanced; 3 buffers need to be added to A and B to balance X: A_1, A_2, A_3 and B_1, B_2, B_3 respectively (Figure 5 (b)). The corresponding event causality diagram is shown in Figure 6 (b). Assuming the worst case scenario that every evaluation leads to an event, the number of evaluations is 12 + $|TFO(X)|$, where $TFO(X)$ denotes the transitive fanout of X. $TFO(X) = \{Y, Z\}$.*

Had we not collapsed W into X, we would need two buffers W_1 and W_2 to make X balanced (Figure 5 (c)). The corresponding event causality diagram is shown in Figure 6 (c). The worst case evaluations are 9 + $|TFO(X)|$.

Finally consider the case when we neither collapse W nor add buffers, i.e., implement the sub-network of Figure 5 (a) as such on the simulator. The corresponding event causality diagram is in Figure 6 (a). The worst case evaluations are 7 + $2|TFO(X)|$. There is a factor 2 with $TFO(X)$, since each of the two evaluations of X can lead to evaluation of its TFO.

We conclude that (c) is strictly better than (b). Also, (c) is better than (a) as long as $|TFO(X)| > 2$. So the best decision (almost always) is not to collapse W into X, but instead to add two buffers at the output of W.

In the last example, our decision was not to collapse W into X. Figure 7 shows a case where it is beneficial to collapse W into X.

For each node, the *partial collapse* algorithm considers which case is applicable (there are more cases than the two described here), and if collapsing would be appropriate in that case. Of course, the memory increase resulting from the collapse should not be more than the leftover memory capacity. The algorithm then evaluates all the node collapses, assigns a cost to each collapse (in the current implementation, the cost is the memory increase resulting from the collapse; in future, it may also reflect change in the number of evaluations), and greedily selects the collapse with the minimum cost. This step is repeated until either no collapse is feasible or each feasible collapse has a memory overhead higher than the remaining memory.

example	pi	po	nodes	edges	lits (fac)	vec.
5xp1	7	10	18	77	114	128
9sym	9	1	12	46	145	512
alu2	10	6	54	273	347	1000
b9	41	21	30	115	124	512
bw	5	28	35	155	160	32
clip	9	5	16	82	117	512
duke2	22	29	81	392	428	1000
e64	65	65	116	253	253	1000
f51m	8	8	16	50	80	256
misex2	25	18	25	103	104	1000
rd84	8	4	18	71	148	256
sao2	10	4	18	93	131	1000
vg2	25	8	10	65	88	1000
rot	135	107	167	597	664	1000
C1355	41	32	162	344	552	1000
apex2	39	3	43	219	268	1000
C1908	33	25	146	383	535	1000

Table 1 Statistics of benchmark circuits

6. EXPERIMENTAL RESULTS

We selected a set of standard MCNC combinational benchmarks and optimized each of them by running twice the optimization script *script.rugged* of sis [2]. We will denote an optimized benchmark by $\widehat{\eta}$. Table 1 provides some information about these benchmarks. The columns *pi* and *po* denote the numbers of primary inputs and outputs respectively. The last four columns refer to numbers of nodes, edges, literals in the factored form in the area-optimized networks $\widehat{\eta}$, and the number of vectors used for simulation. We generated pseudo-random 0-1 vectors to simulate the networks. The probability of each input being a 0 was set to 0.5. For each benchmark, one thousand vectors were generated, unless the number of primary inputs, *pi*, is too small, in which case 2^{pi} vectors were generated. The same set of vectors is used to simulate a benchmark in all the experiments.

The experimental set-up used is shown in Figure 8. Given the optimized network $\widehat{\eta}$, a synthesis technique or script converts it into a network that is more suitable for the TP5000 simulator. For each synthesis technique, the memory constraint M was set to $2^{18} = 262,144$. We do not report the memory used by the final network implementation, since it is always at most M. Once the network is stored in the TP5000 memory, input vectors are applied and the simulation results are compiled. The TP5000 simulator keeps track of the number of events over the entire input vector set, which is what we report in Table 2.

We compare various synthesis techniques discussed in the paper by reporting in Table 2 the number of events generated in the resulting networks during their simulation by TP5000. For each benchmark, the bold entry represents the minimum event count.

The column 2ip-ao corresponds to applying on $\widehat{\eta}$ a 2-input AND-OR decomposition, which decomposes the network into 2-input AND, OR, and inverter gates. The motivation for this was that 2-input gates are the primitives that a software simulator typically uses. So 2ip-ao attempts to mimic the performance of software simulators.

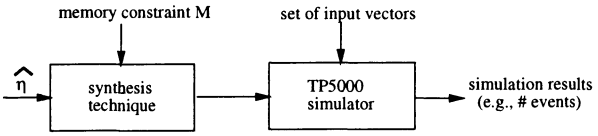

Figure 8 The experimental set-up

The column min-fn corresponds to the technique of [7], which minimizes the total number of functions in the network subject to the memory constraint M (see Section 3). In [7], no simulation results were presented for min-fn. Comparing 2ip-ao with min-fn, we find that in almost all examples, minimizing the number of functions helped reduce the number of events, sometimes by more than a factor of 2. However, in two benchmarks, the number of events went up. For *C1355*, it more than doubled! Although min-fn incorporates a good cost function, it is not always accurate.

To see the effect of buffer insertion, on each optimized network $\hat{\eta}$ we applied min-fn followed by the buffer insertion algorithm of Section 5.1. This generates a perfectly balanced network. The corresponding simulation results are shown in the column min-fn + buf. Although min-fn + buf can add a lot of buffers (on average, 3.4 buffers per function), 11 out of 17 benchmarks had fewer events as a result of buffer addition as compared to min-fn. In some examples such as *9sym*, the reduction in events is by more than a factor of 3, although the number of functions in the buffered network is 3 times! Over the entire benchmark suite, buffering reduced the average number of events by 12.6% over min-fn. However, not all benchmarks benefited from buffer insertion. For instance, *e64, rot,* and *C1908*. This, as mentioned in Section 5.1, is because there are transitions at the buffer outputs and adding too many buffers can backfire. On these three benchmarks, the buffers added were 4 to 5.5 times the number of functions!

Since our goal of reducing the number of events and glitches in a circuit is covered by the power minimization research, we decided to borrow and evaluate techniques that generate low-power circuits [8, 10]. Unfortunately, we could not obtain a tool that minimizes glitches. The power minimization tools currently available in the public-domain use a zero delay model, which precludes consideration of glitching activity. However, we could get our hands on a power estimation tool that considers unit delay model and glitching-activity as well [5]. We used it as follows.

1. Given the area-optimized network $\hat{\eta}$, use the technique proposed in [6] to generate a network $\tilde{\eta}$, which is in terms of 2-input AND and OR gates and has possibly much fewer transitions. The technique of [6] specifically considers glitch minimization.

2. Using power-estimation of [5], identify the node of $\tilde{\eta}$ with the highest switching-activity that can be collapsed into all its fanouts without violating the memory constraint. The idea is to get rid of nodes with high switching activity. Collapse this node into all its fanouts. Repeat this step until memory constraint can no longer be satisfied.

3. Use buffer insertion to balance the network.

We call this technique min-pwr; the corresponding simulation results are shown in the column min-pwr of Table 2. The results are presented only on a subset of the benchmarks, since the power estimation tool could not finish on others. Comparing with min-fn + buf, it is evident that min-pwr is not effective except on *f51m*. We believe one reason for its poor showing is that collapsing a node into its fanouts changes the switching activity of the transitive fanout nodes. This change is difficult to model and is ignored in our technique.

The last two columns 2ip + bal and bal-dec + bal use the techniques of Sections 5.2 and 5.3. Both have three steps: decomposition, partial collapse (of Section 5.3) and buffer insertion.

ex	2ip-ao	min-fn	min-fn + buf	min-pwr	2ip + bal	bal-dec + bal
5xp1	14.2 K	6.8 K	3.2 K	5.2 K	**2.3 K**	2.4 K
9sym	77.2 K	34.9 K	**10.0 K**	36.1 K	12.7 K	45.3 K
alu2	177.8 K	125.4 K	92.9 K	196.3 K	**73.6 K**	87.1 K
b9	64.4 K	36.7 K	37.8 K	51.6 K	**35.8 K**	40.6 K
bw	5.7 K	2.5 K	1.2 K	1.1 K	**0.8 K**	1.1 K
clip	58.4 K	29.0 K	14.6 K	44.6 K	**13.5 K**	24.8 K
duke2	262.9 K	218.6 K	223.6 K	247.8 K	234.0 K	**180.3 K**
e64	**121.8 K**	155.8 K	164.7 K		160.1 K	176.6 K
f51m	20.7 K	14.5 K	15.9 K	**4.4 K**	15.9 K	12.1 K
misex2	74.8 K	60.3 K	**49.7 K**	59.2 K	50.6 K	67.6 K
rd84	47.8 K	15.3 K	9.0 K	19.3 K	**5.1 K**	10.2 K
sao2	110.7 K	61.9 K	**41.2 K**	109.7 K	53.7 K	46.7 K
vg2	78.5 K	61.1 K	60.1 K	51.7 K	56.2 K	**37.0 K**
rot	73.0 K	**59.0 K**	74.7 K		71.4 K	68.5 K
C1355	266.5 K	567.2 K	394.5 K		424.1 K	**189.4 K**
apex2	205.0 K	121.4 K	**114.7 K**		121.4 K	156.6 K
C1908	450.4 K	253.0 K	311.2 K		**161.9 K**	276.2 K
total	2109.8 K	1823.4 K	1619.0 K		1493.1 K	1422.5 K

Table 2 Comparing # events for all synthesis techniques

The first technique, 2ip + bal, uses two-input FPGA decomposition, the one used in min-fn. The second technique, bal-dec + bal, instead, employs the balanced decomposition of Section 5.2. The default is a 6-input decomposition, i.e., $k = 6$. We experimented with several values of k; $k = 6$ yielded the best results. If the resulting network does not satisfy the memory constraint, a 4-input balanced decomposition is done. For no benchmark in our suite did we need to decompose any more. Partial collapse and buffer insertion steps are identical for 2ip + bal and bal-dec + bal.

In terms of the total number of events, bal-dec + bal is the best among all the six techniques of Table 2, closely followed by 2ip + bal. bal-dec + bal is 5.0% better than 2ip + bal, 22.0% better than min-fn, and 32.6% better than 2ip-ao. Out of the 17 benchmarks, 2ip + bal has the minimum events in 7, min-fn + buf in 4, bal-dec + bal in 3, and min-fn, 2ip-ao, and min-pwr in 1 each. Finally, we observed that all the synthesis techniques except min-pwr are really fast; none took more than 50 seconds on a Sparcstation 20 for any benchmark. min-pwr is slow because of repeated invocations of the power estimation tool.

We conclude that *generating a perfectly balanced network by* bal-dec + bal *and* 2ip + bal *is good for minimizing the event count.* Since no technique wins all the time, we believe that *predicting switching activity accurately in a logic network is difficult.*

7. CONCLUSIONS

In this paper, we investigated the problem of speeding up an event-driven simulator implemented on a lookup-table based hardware accelerator TP5000. However, the techniques we proposed are general and can be applied to other domains such as software functional simulation and synthesis of glitch-free, low-power circuits.

Our methodology has a few drawbacks. First, it only considers the worst case behavior, that each evaluation leads to an event. It will be better to take into account the transition probability of each node when considering the propagation of evaluations and make transformation decisions accordingly. Second, it only addresses the mapping problem. We should also solve the optimization problem for simulation speed-up. E.g., we will like to address the *simplification* problem for TP5000: *Given a node N with the associated logic function f and a corresponding sum-of-products representation (SOP), derive another SOP for f that makes N more balanced.* Finally, we wish to push large industrial designs through our techniques.

Acknowledgments

The authors thank Satoshi Kowatari for helping with the TP5000 interface and Ankur Srivastava for proof-reading the paper.

References

[1] M. M. Denneau. The Yorktown Simulation Engine. In *Proceedings of the Design Automation Conference*, June 1982.

[2] Ellen M. Sentovich et. al. *SIS: A System for Sequential Circuit Synthesis.* Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[3] F. Hirose. Simulation Processor SP. In *Proceedings of the International Conference on Computer-Aided Design*, November 1987.

[4] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia. Fast discrete function evaluation using decision diagrams. In *Proceedings of the International Workshop on Logic Synthesis*, May 1995.

[5] J. Monteiro, S. Devadas, and B. Lin. A Methodology for Efficient Estimation of Switching Activity in Sequential Logic Circuits. In *Proceedings of the Design Automation Conference*, pages 12–17, June 1994.

[6] R. Murgai, R. K Brayton, and A. S. Vincentelli. Decomposition of Logic Functions for Minimum Transition Activity. In *European Design and Test Conference*, 1995.

[7] R. Murgai, M. Fujita, and F. Hirose. Logic Synthesis for a Single Large Look-up Table. In *Proceedings of the International Conference on Computer Design*, 1995.

[8] A. Shen, S. Devadas, J. White, A. Ghosh, and K. Keutzer. A Combinational Logic Design Methodology Targeting Low Power Applications. In *MIT Technical Report (available from the authors)*, February 1993.

[9] K. Takayama, M. Shoji, S. Shimogori, and F. Hirose. VHDL Simulator Using TP5000. In *Fujitsu Scientific and Technical Journal*, volume 31, December 1995.

[10] V. Tiwari, P. Ashar, and S. Malik. Technology Mapping for Low Power. In *Proceedings of the* 30^{th} *Design Automation Conference*, pages 74–79, June 1993.

[11] D. Wong, G. De Micheli, and M Flynn. Inserting active delay elements to achieve wave pipelining. In *Proceedings of the International Conference on Computer-Aided Design*, pages 270–273, November 1989.