EventTimer: Fast and Accurate Event-Based Dynamic Timing Analysis

Zuodong Zhang[†], Zizheng Guo[‡], Yibo Lin [‡⋆], Runsheng Wang[†], and Ru Huang[†]

[†]*Institute of Microelectronics, Peking University, Beijing, China*
[‡]*CECA, CS Department, Peking University, Beijing, China*

***Abstract*—As the transistor shrinks to nanoscale, the overhead of ensuring circuit functionality becomes extremely large due to the increasing timing variations. Thus, better-than-worst-case design (BTWC) has attracted more and more attention. Many of these techniques utilize dynamic timing slack (DTS) and activity information for design optimization and runtime tuning. Existing DTS computation methods are essentially a modification to the worst-case delay information, which cannot guarantee exact DTS and activity simulation, causing performance degradation in timing optimization. Therefore, in this paper, we propose EventTimer, a dynamic timing analysis engine based on event propagation to accurately compute DTS and activity information. We evaluate its accuracy and efficiency on different benchmark circuits. The experimental results show that EventTimer can achieve exact DTS computation with high efficiency. And it also proves that EventTimer has good scalability with the circuit scale and the number of CPU threads, which make it possible to be used in the application-level analysis.**

I. Introduction

With the continuous shrinking of semiconductor technology nodes, non-ideal factors like process variations and transistor aging start to introduce non-negligible static and dynamic timing variations [1]–[4]. To guarantee the timing correctness under a wide range of operating temperatures and voltages, designers have to increase the guardband by lowering the performance, as the widely-adopted static timing analysis (STA) methodology relies on analyzing the worst-case scenarios and imposes conservative constraints to the designs.

To recover the performance, better-than-worst-case design (BTWC) has been proposed to relax the conservative design constraints, e.g., increasing the clock frequency or decreasing the voltage depending on the working scenarios dynamically beyond the requirements from STA. A typical BTWC methodology relies on the circuit- and application-level analysis to predict/detect potential timing errors and trying to avoid/correct such errors at runtime [5]–[9]. Some other studies even exploit inherent application-level resilience to tolerate timing errors [10]–[15], such as neural network accelerators. These BTWC techniques usually go beyond static timing information and leverage dynamic timing analysis (DTA) to guide design optimization strategies like dynamic voltage/frequency scaling (DVFS) and application-level fault tolerance.

To obtain the dynamic timing information, the aforementioned studies usually first perform graph-based STA to obtain the worst-case delay, and then delay-annotated gate-level simulation with a post-processing step to obtain the dynamic information. Eventually, they use the dynamic timing information to guide the runtime optimization strategies. For simplicity, we call this method graph-based DTA (GB-DTA). However, such an approach is still too conservative because graph-based STA yields inaccurate delay annotations due to its pessimistic delay update [16]. Cherupalli *et al* [17] propose a new graph-based algorithm to report the N-worst exercised paths. Several recent works [12], [18], [19] adopt machine learning to predict the timing error or dynamic delay of a particular circuit, which have the potential to improve the runtime of DTA. Garyfallou *et al* [20] propose to recalculate the arrival time of switching activities based on the gate-level simulation results to eliminate the pessimism of graph-based STA. Their algorithm can only compute dynamic timing slacks (DTS) without reporting toggled paths and switching activities, while its dependency on external gate-level simulators also limits the runtime scalability due to the large overhead from reading temporary files (e.g., VCD files from ModelSim).

To effectively guide BTWC optimization, fast and accurate DTA engines are needed. Therefore, in this paper, we propose EventTimer, an event-based DTA engine that can calculate DTS and switching activities fast and accurately. To the best of our knowledge, EventTimer is the first parallel DTA engine that does not suffer from the pessimistic graph-based assumption. The major contributions of this work are as follows:

1) We propose EventTimer, a DTA framework with integrated logic simulation and timing analysis, which does not need any gate-level simulation or external activity information file. This feature makes EventTimer available for large-scale circuits.
2) We propose a new event propagation algorithm, which propagates events cell by cell with accurate timing information in each cycle. The proposed algorithm can provide finer scope to inspect the timing and activity information of the internal nodes.
3) We propose a parallelization scheme across independent simulation cycles, which enables EventTimer to gain performance benefit from multi-core CPUs and complete the analysis much faster. The results show that EventTimer can achieve 18-21× speedup with 32 threads.
4) We also develop a new test pattern generation method. The results show that the proposed method can achieve 82%–100% coverage with a simulation of 100K input vectors, which is up to 37% higher coverage than that of the conventional test pattern generation method.

The rest of this paper is organized as follows. Section II introduces the background of STA, DTA, and formulates the problem. Section III presents details of the algorithm. Section IV demonstrated the experimental results of our algorithm. Finally, Section V concludes the paper.

II. Preliminaries

A. Static Timing Analysis

There are two common STA methodologies, namely the graph-based STA (GB-STA) and the path-based STA (PB-STA). In GB-STA, the circuit netlist is modeled as a directed acyclic graph (DAG), which is composed of nodes (primary ports and pins) and edges (timing arcs from cells and

⋆Corresponding author: yibolin@pku.edu.cn

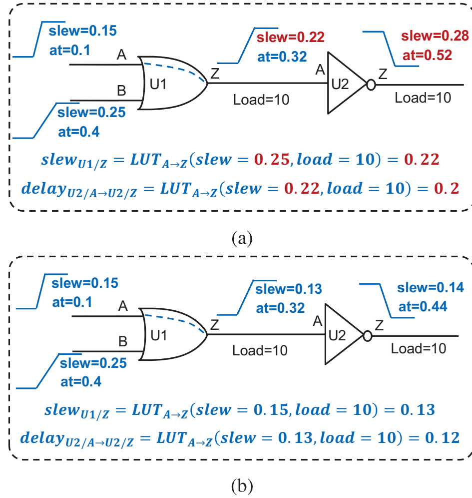

Fig. 1: Input slew merge introduces pessimism in delay calculation of GB-STA. (a) Graph-based static timing analysis. (b) Path-based static timing analysis.

nets). GB-STA typically contains two major phases, forward propagation and backward propagation. Forward propagation computes the timing information such as slew, delay, and arrival time (*at*). Backward propagation computes the required arrival time (*rat*).

GB-STA can calculate the whole timing graph and report the top-1 critical path quickly [21]. However, it can introduce pessimism because of the worst-case slew propagation. For example, the slew propagation in GB-STA utilizes the worst slew of input pins, as shown in Fig. 1(a). During a setup check, the slew of the pin $U1/Z$ would be computed assuming the worst slew, i.e. slew at pin B, thus the slew of the pin $U1/Z$ will be overestimated. More importantly, the delay of all gates in the latter such as $U2$ would be overestimated.

Different from GB-STA, the PB-STA only calculate the delay of a particular path, so there is no need to merge the input slews and the path delay calculated is more accurate (Fig. 1(b)) [22], [23].

B. Dynamic Timing Analysis

No matter GB-STA or PB-STA is adopted, the reported critical paths are only topologically, and STA cannot analyze which input patterns may cause timing errors and estimate the error rate. Therefore, BTWC optimization aims at relaxing the conservative design constraints from STA by analyzing the dynamic information, such as DTS, toggle rates of the critical paths, etc. Fig. 2 shows an example that using DTA to optimize the application-level DVFS strategies [6], [7], [24]. By using DTA to estimate the minimum DTS, designers can find the max frequency or minimum voltage for each application, which can help develop the runtime DVFS strategies for better performance or energy efficiency.

GB-DTA is a commonly adopted technique to estimate such dynamic information. It leverages the annotated delay information extracted from GB-STA together with gate-level simulation to analyze the switching activities of the critical paths. The major drawback for GB-DTA is that it can still introduce pessimism due to the pessimistic delay from GB-STA.

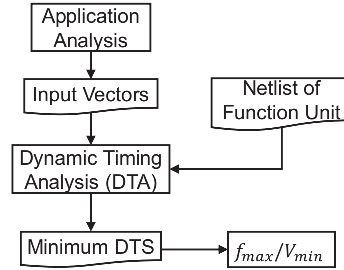

Fig. 2: Using DTA to optimize application-level DVFS strategies.

Recently, there is a growing interest in using machine learning to speedup GB-DTA. A typical approach is to build fast machine learning models to predict the dynamic delay or timing error rate [12], [18], [19]. More than 10× speedup over GB-DTA has been reported with the machine learning techniques.

C. Problem Formulation for Dynamic Timing Analysis

DTA usually estimates statistical information of delay and activity in multiple cycles. The problem formulation of DTA for a particular cycle is defined as following.

Problem. Given the state of all pins and a specific input vector, simulate the switching activities of each internal node with timing information, report the delay of a specific path and the critical ones with the largest delay.

Definition 1. *Event:* a digital switching signal on a pin. The *at*(arrive time) of an *event* is the time of occurrence relative to the last clock. The *slew* of an *event* is the time token for signal transition.
Definition 2. *Toggled pin:* a pin is toggled in a particular cycle if the value of the pin changes in that cycle
Definition 3. *Toggled path:* a path is toggled in a particular cycle if all the pins in the path toggle in that cycle
Definition 4. *Critical endpoints:* monitored timing endpoints (flip-flop or output port) specified by users, generally are the endpoints where timing violation may occur (both setup violation and hold violation).
Definition 5. *Critical subgraph:* a subgraph which contain all critical endpoints and all nets/gates related to the critical endpoints.

In practice, we can support multi-cycle simulation by performing the above single-cycle DTA repeatedly. Accurate DTA essentially requires combining gate-level simulation and timing analysis. One straightforward way is to leverage an external gate-level simulator like ModelSim for the activity information [20], but the large output VCD files and IO overhead result in impractical runtime and disk usage.

Hence, a practical DTA tool must have integrated gate-level simulation for the activity information and timing propagation for scalable timing analysis.

III. Algorithm

In this section, we explain the details of the proposed event-based DTA. We first give an overview of the proposed event-driven DTA and then present some details. The overall taskflow of the algorithm is shown in Fig. 3. The proposed algorithm contains two steps: structure initialization and cycle-by-cycle timing analysis.

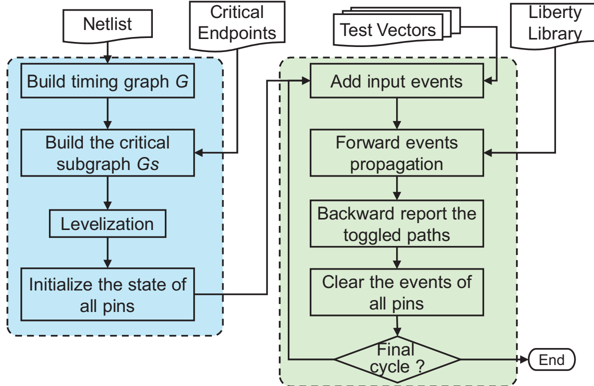

Fig. 3: The overall taskflow of EventTimer.

A. Structure Initialization

The structure initialization step is to prepare the necessary data for timing propagation. In this step, we first read the netlist and build the timing graph. Then, we prune the timing graph and keep only the critical subgraph according to the user-specified critical endpoints. The critical subgraph is the minimum subgraph that contains all nodes on the fan-in cone of the critical endpoints. Using the critical subgraph instead of the whole timing graph can effectively reduce the runtime, especially for hold time check, because it avoids updating the timing information of the unnecessary gates/nets. Algorithm 1 shows the pseudocode of our critical subgraph building algorithm. The algorithm starts at the critical endpoints (line 8), then, adds fan-in nodes of the current nodes to the subgraph recursively until encountering the input nodes (lines 3-5).

Algorithm 1: Build the critical subgraph.

Input: Timing graph G, critical endpoints EPT_c
Output: the critical subgraph G_s

1 **Function** *add_node_to_subgraph* $(node\ n)$**:**
2 $\quad G_s.push(n)$;
3 $\quad$ **if** $n.in_degree! = 0$ **then**
4 $\quad\quad$ **for** $node\ n \in fanin(n)$ **do**
5 $\quad\quad\quad add_node_to_subgraph(n)$;
6 $Gs \leftarrow \{\}$;
7 **for** *all critical endpoint* $e_i \in EPT_c$ **do**
8 $\quad add_node_to_subgraph(e_i)$;

After obtaining the critical subgraph, we levelize the graph to build level-by-level dependencies of the gates for events propagation. Finally, we finish the preparations for event propagation by setting all primary inputs to zero and initializing all pin states accordingly.

B. Cycle-by-Cycle Timing Analysis

The central step of EventTimer is the cycle-by-cycle timing analysis, where we perform timing analysis between adjacent cycles based on the test vectors. On each cycle, the task can be divided into three main sub-steps: input events building, event propagation, and path reporting.

1) Input Events Building: The first step is building the input events based on the input vectors and add these events to the corresponding input pins. First, we read the input vectors and compares the value of each bit with the value of the previous

Algorithm 2: Propagate events through the gate.

1 **Function** *propogate_events_gate(gate* g_i*)***:**
2 $\quad PIN_{in} \leftarrow get_input_pins(g_i)$;
3 $\quad PIN_{out} \leftarrow get_output_pins(g_i)$;
4 $\quad$ build a list of time $T(g_i)$;
5 $\quad T(g_i) \leftarrow get_input_event_at(g_i)$;
6 $\quad$ sort $T(g_i)$ and ensure that there are no two input events with the same at;
7 $\quad$ **for** *all time* $t_i \in T(g_i)$ **do**
8 $\quad\quad$ find the corresponding input event $event_in$;
9 $\quad\quad$ get the states of input pins at time t_i, and calculate the output value;
10 $\quad\quad$ **for** *all pin* $p_i \in PIN_{out}$ **do**
11 $\quad\quad\quad$ **if** *state of* p_i *changed* **then**
12 $\quad\quad\quad\quad$ build a output event $event_out$;
13 $\quad\quad\quad\quad event_out.at = event_in.at + Cell_delay(event_in.slew, load)$;
14 $\quad\quad\quad\quad event_out.slew = Cell_slew(event_in.slew, load)$;
15 $\quad\quad\quad\quad$ add $event_out$ to the events list of p_i;
16 $\quad$ filter the improper event pairs

cycle. A changed value indicates that an event occurs on the specific input. The slew of input event is the user defined $input_slew$, and the arrival time (*at*) of the event is the user defined $external_delay$ plus a random disturbance, denoted as $input_uncertainty$. We offset the arrival time of input pins by a random disturbance because of two reasons: 1) Random disturbance imitates the stochastic behavior caused by the input wire delay or the clock skew of the former flip-flops. 2) Random disturbance prevents the situation where two input events of a gate arrive at the same time. Thus, it eliminates the ambiguity in event propagation.

2) Event Propagation: In the second step, we propagate the events level by level to the timing endpoints. Fig. 4 gives an example to better describe the propagation algorithm. We propagate the events through gates and nets level by level. Specifically, we apply the propagation for net arcs and cell arcs alternately.

A net has one input pin and multiple output pins. Nets in effect only offset the arrival time of input events by a propagation delay and do not change their slew values. We first find all output pins of the net, and then calculates the corresponding wire delay, and finally adds the events to the output pins.

Event propagation through gates is more complex as it involves logic simulation and look-up table query for delay and output slew computation. Also, the process has to deal with multi-input gates, where several input events coming from different input pins may trigger output events.

Algorithm 2 presents the gate propagation process. We first build a sorted list of all arrival times of input events (lines 4-6). Then, we simulate the logic behavior of the gate and get the gate output value in chronological order (lines 7-9). A toggled output value indicates a new output event (lines 9-11). For each output event that occurs, we compute the cell delay and output slew based on the look-up table in standard cell libraries, and add the event to the event list of the corresponding output pin (lines 12-14).

After propagating all events in a cycle, we check the events

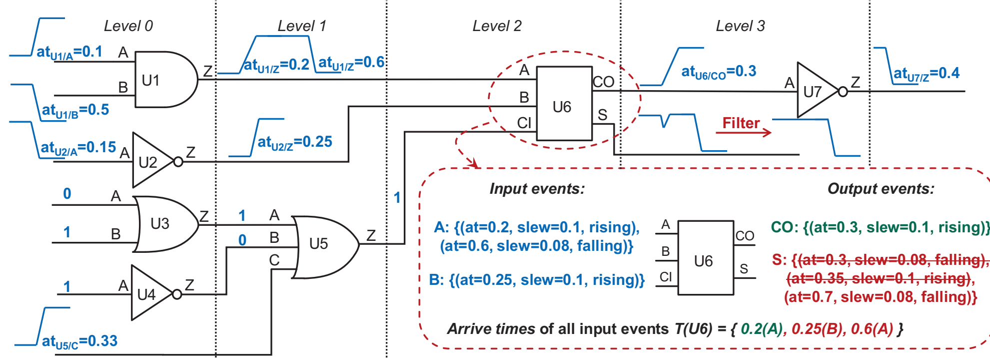

Fig. 4: The schematic of the events propagation. To briefly demonstrate the propagation algorithm, the wire delay is ignored and all cell delay is assumed to be 0.1.

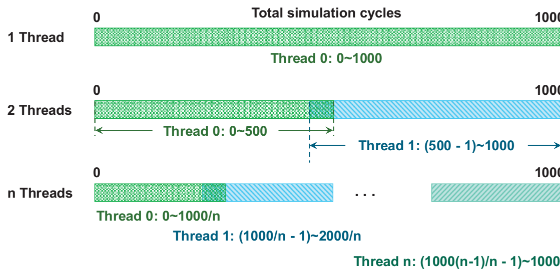

Fig. 5: The cycle parallel scheme of EventTimer.

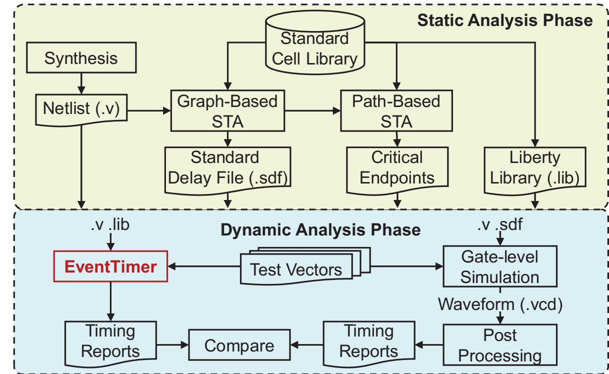

Fig. 6: Workflow to validate EventTimer.

list of each output pin and filter the improper event pairs, e.g., the case that the arrival time of two events is too close. In this case, it is more likely that the output signal will not change at all. For example, the time interval of the first two events of pin $U6/S$ in Fig. 4 is smaller than the $slew$, which means neither of these signal switchings will be completed. Therefore, these output events will be removed from the events list.

3) Path Reporting: The final step is reporting the toggled paths. The algorithm firstly accesses the events list of all critical endpoints, and reports the *at* of the last event as the path delay. Then, find the input pin and event which triggered this event at the former level, and report it. Repeat this operation until reaching the input pins. After reporting all critical endpoints, the algorithm clears all the events and only reserve the state of pins as the initial state for the simulation of the next cycle.

C. Parallel Acceleration

As mentioned above, to analyze the statistical dynamic information of a practical application, the number of cycles to be simulated is extremely large (usually $>$ 10M) [11]. To accelerate the simulation, two speedup methods are proposed: the first is the critical subgraph analysis which is already mentioned in the previous section, and the second is the cycle parallelism. The most time-consuming part of EventTimer is the cycle-by-cycle timing analysis. Therefore, we utilize cycle parallelism for acceleration.

We divide the total number of cycles to be simulated into n equal parts and assign them to n different threads for execution. The rules for cycle division and thread assignment are shown in Fig. 5. Note that there is a one-cycle overlap between different parts, because the circuit takes one cycle to be in the state that it is supposed to be in. Using cycle parallelism avoids complex netlist splitting algorithms, and threads are executed independently, which results in better acceleration.

D. Critical Pattern Generation

Test pattern generation is critical to the coverage of DTA. Inspired by [25], we develop a delay-guided fuzzing-based method to find the *critical patterns* (the input patterns that can toggle the critical path). We define the input pattern as a pair of the current input and the previous input: $\{x[t], x[t-1]\}$. Then, we use EventTimer to calculate the delay of these inputs, and find the top 5% critical patterns to generate new child patterns. For new pattern generation, we perform *random bit flipping* with a certain probability p to each bit in these critical patterns. To speed up convergence while avoiding falling into local optima, we use three different probabilities, each of which will be used to generate one-third of the new patterns.

IV. Experimental Results

We implemented EventTimer in C++ and conducted all the experiments on a Linux machine with Intel Xeon E5-2650 at 2.20GHz and 64 GB RAM. AxBench benchmarks [26] are adopted to validate the algorithm, containing function units that are commonly used in embedded systems, from medium-scale ALUs to large-scale multipliers [28]. The workflow of the experiments is shown in Fig. 6, which consisting of two phases, enabling us to compare with GB-STA, PB-STA, and GB-DTA. In the static analysis phase, we use Synopsys Design Compiler to synthesize the benchmark circuits. We then use

TABLE I: Top-1 path delay calculated by GB-STA, PB-STA, GB-DTA and EventTimer. In principle, the delay values from EventTimer should not be larger than those from PB-STA, and should be less than those from GB-DTA. The results are consistent with such theoretical analysis.

Benchmark [26]	# Gates	# I/O	Top 1 Path Delay (ns)				
			STA [27]		DTA		
			GB-STA	PB-STA	GB-DTA [7], [8]	EventTimer	Pessimism Eliminated
`brent.kung.16b`	55	32/17	297.677	296.204	296.996	296.204	0.792
`brent.kung.32b`	201	64/33	534.335	531.189	534.338	531.189	3.149
`kogge.stone.32b`	151	64/33	569.107	567.157	568.974	567.158	1.816
`multiplier.8b`	227	16/16	336.076	335.258	334.459	333.13	1.329
`mac.8b`	251	33/17	402.894	401.567	402.892	401.566	1.326
`multiplier.16b`	882	32/32	684.941	682.895	623.712	611.299	12.413
`multiplier.32b`	3395	64/64	1263.871	1258.956	1054.098	1036.895	17.203

TABLE II: Coverage improvement of the proposed fuzzing-based method.

Benchmark [26]	Coverage		Improv.
	w/o Fuzzing	w/ Fuzzing	
`brent.kung.16b`	100%	100%	-
`brent.kung.32b`	77%	100%	23%
`kogge.stone.32b`	80%	100%	20%
`multiplier.8b`	97%	99%	2%
`mac.8b`	100%	100%	-
`multiplier.16b`	78%	90%	12%
`multiplier.32b`	45%	82%	37%

Synopsys PrimeTime [27] to perform GB-STA, report the top-N critical paths, and generate the standard delay files (SDF). Next, we turn on the path-based analysis option in PrimeTime (later referred as PB-STA) to recalculate the path delay of the top-N critical paths and report the top-N critical endpoints as the monitored endpoints in the dynamic analysis phase. We employ the open-source Nangate 15nm standard cell library [29] recharacterized with a commercial 16/14nm FinFET modelcard as the standard cell library.

In the dynamic analysis phase, we use two methods to generate the input patterns: 1) randomly generate 100K test vectors; 2) fuzzing-based method, i.e., each iteration produces 1K input vectors, and the maximum iteration is 100. The three flipping probabilities are 2.5%, 5%, and 7.5%, respectively. Then, we can have two methods to report the delay of each cycle and the top-N critical paths: 1) adopt the GB-DTA methodology [7], [8] that requires a post-processing program (Python) to extract the delay from the VCD files generated by the gate-level simulation (ModelSim); 2) use EventTimer. In this way, we can compare the timing reports for the accuracy of path delay calculation, and compare the logic outputs with gate-level simulation to verify the logic functionality. In all the experiments, we assume zero wire delays, since wire delay models are orthogonal to this work and wire delays can be precomputed from commercial tools like PrimeTime.

A. The Top-1 Path Delay

Table I shows the top-1 path delay of GB-STA, PB-STA, GB-DTA and EventTimer. PB-STA is in general more accurate than GB-STA. We expect the top-1 path delay calculated by GB-DTA equal to or less than the result of GB-STA, depending on whether the critical path is exercised during the simulation. We also expect the top-1 path delay calculated by EventTimer equal to or less than the result of PB-STA, as there should be no pessimistic estimation in EventTimer.

From Table I, we can see that the results match our expectations. The top-1 path delay values from GB-DTA are no larger than those from GB-STA. The delay values from EventTimer are no larger than those from PB-STA. These results indicate that EventTimer produces reasonable delay values. We need to mention that the pessimism of GB-STA does not seem to be large in the experiments. This is because the results are extracted from the front-end analysis ignoring the wire delay, so the large slews in internal nodes of the circuits are underestimated. For a real design with high fan-outs and long wires, the pessimism introduced by GB-STA will be much more severe, when taking the back-end information into account.

Moreover, although not shown in the table, when comparing EventTimer with GB-DTA, we observe that even though with different delay values, the cycles in which the top-1 path is toggled are the same, and the outputs of each cycle match, which proves the correctness of EventTimer.

To evaluate the fuzzing-based method, we define the coverage of EventTimer as:

$$\text{Coverage} = \frac{\text{Top delay of EventTimer}}{\text{Top delay of PB-STA}}. \tag{1}$$

Table II shows that the proposed fuzzing-based method can achieve 82%–100% coverage, which is up to 37% higher than that of the conventional method. It should be noted that the proposed fuzzing-based method will only increase the runtime by less than 10%, because the total number of simulation cycles is the same.

It can also be seen that the fuzzing-based method has found the critical patterns for most benchmarks, except for the `multiplier.16b` and `multiplier.32b`. Even after a very long search, the fuzzing-based approach still cannot find an input pattern that could toggle the static critical path reported by STA. This result is consistent with previous experimental observations [6]. It may be because that the static critical path cannot actually be toggled by any input pattern, especially in complex circuits.

B. Parallel Acceleration and Runtime Comparison

Fig. 7(a) shows the multi-threading results of EventTimer. The plot shows that EventTimer is almost linearly scalable with the number of CPU threads. The results on other bench-

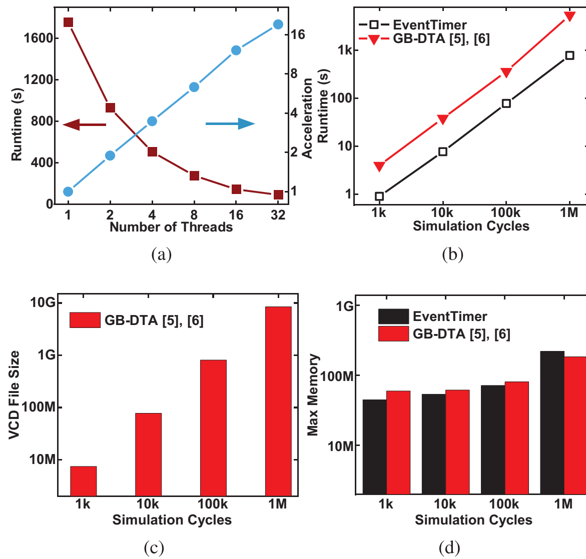

Fig. 7: (a) Multi-threaded acceleration of EventTimer. (b) Total runtime of EventTimer and GB-DTA for different number of simulation cycles. (c) VCD file size at different simulation cycles in GB-DTA. (d) Maximum memory used by EventTimer and GB-DTA for different number of simulation cycles. The benchmark is `multiplier.16b`.

marks are similar. In general, EventTimer can achieve 18-21× speedup with 32 threads.

Then, we compare the runtime of EventTimer with that of GB-DTA (Fig. 7(b)). The EventTimer is executed in parallel with 32 threads. GB-DTA requires a gate-level simulation with a post-processing program. The runtime of gate-level simulation is comparable to that of EventTimer, while the post-processing is very time-consuming, because generating and analyzing the large VCD file introduces a significant runtime overhead (Fig. 7(c)). Eventually, EventTimer is 6.9× faster than GB-DTA. Fig. 7(d) shows that the maximum memory used by EventTimer is similar to GB-DTA, because EventTimer only records the states of the last cycle.

V. Conclusion

In this paper, we present EventTimer, an integrated simulation and timing analysis framework. EventTimer simulates the switching activities by forward events propagation, and performs timing analysis by backward path reporting. EventTimer adopts cycle parallelism, which makes it well scalable with the number of CPU threads. We compare the DTA results from EventTimer and the conventional GB-DTA, proving that EventTimer does not suffer from the pessimistic graph-based assumption and can calculate the path delay accurately. Our future work includes taking account timing variation caused by process variation into EventTimer, and estimating the timing error rate for low-power near-threshold applications.

Acknowledgements

This work is supported in part by the National Key R&D Program (2020YFB2205500), NSFC (62034007, 62004006, 62125401), and 111 Project (B18001).

References

[1] R. Huang, X. Jiang, S. Guo, P. Ren, P. Hao, Z. Yu, Z. Zhang, Y. Wang, and R. Wang, “Variability-and reliability-aware design for 16/14nm and beyond technology,” in *Proc. IEDM*, 2017, pp. 12.4.1–12.4.4.
[2] Z. Ji, H. Chen, and X. Li, “Design for reliability with the advanced integrated circuit (ic) technology: challenges and opportunities,” *Science China Information Sciences*, vol. 12, 2019.
[3] Z. Zhang, R. Wang, X. Shen, D. Wu, J. Zhang, Z. Zhang, J. Wang, and R. Huang, “Aging-aware gate-level modeling for circuit reliability analysis,” *IEEE TED*, vol. 68, no. 9, pp. 4201–4207, 2021.
[4] J. Chen, H. Kando, T. Kanamoto, C. Zhuo, and M. Hashimoto, “A multicore chip load model for pdn analysis considering voltage–current-timing interdependency and operation mode transitions,” *IEEE TCPMT*, vol. 9, no. 9, pp. 1669–1679, 2019.
[5] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, “Razor: a low-power pipeline based on circuit-level timing speculation,” in *Proc. MICRO*, 2003, pp. 7–18.
[6] H. Cherupalli, R. Kumar, and J. Sartori, “Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems,” in *Proc. ISCA*, 2016, p. 671–681.
[7] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg, “Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment,” in *Proc. DATE*, 2015, pp. 381–386.
[8] I. Tsiokanos, L. Mukhanov, and G. Karakonstantis, “Low-power variation-aware cores based on dynamic data-dependent bitwidth truncation,” in *Proc. DATE*, 2019, pp. 698–703.
[9] K. Wang, Z. Xiong, L. Chen, P. Zhou, and H. Shin, “Joint time delay and energy optimization with intelligent overclocking in edge computing,” *Science China Information Sciences*, vol. 63, pp. 1–16, 2020.
[10] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, “Analysis and characterization of inherent application resilience for approximate computing,” in *Proc. DAC*, 2013.
[11] O. Assare and R. Gupta, “Accurate estimation of program error rate for timing-speculative processors,” in *Proc. DAC*, 2019.
[12] J. Zhang and S. Garg, “Fate: Fast and accurate timing error prediction framework for low power dnn accelerator design,” in *Proc. ICCAD*, 2018, pp. 1–8.
[13] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, “Thundervolt: Enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators,” in *Proc. DAC*, 2018.
[14] Z. Zhang, R. Wang, Z. Zhang, R. Huang, C. Meng, W. Qian, and Z. Zhou, “Reliability-enhanced circuit design flow based on approximate logic synthesis,” in *Proc. GLSVLSI*, 2020, p. 71–76.
[15] X. Li, G. Yan, J. Ye, and Y. Wang, “Fault tolerance on-chip: a reliable computing paradigm using self-test, self-diagnosis, and self-repair (3s) approach,” *Science China Information Sciences*, vol. 61, pp. 1–17, 2018.
[16] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media, 2009.
[17] H. Cherupalli and J. Sartori, “Scalable n-worst algorithms for dynamic timing and activity analysis,” in *Proc. ICCAD*, 2017, pp. 585–592.
[18] X. Jiao, A. Rahimi, Y. Jiang, J. Wang, H. Fatemi, J. P. de Gyvez, and R. K. Gupta, “Clim: A cross-level workload-aware timing error prediction model for functional units,” *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 771–783, 2018.
[19] X. Jiao, D. Ma, W. Chang, and Y. Jiang, “Tevot: Timing error modeling of functional units under dynamic voltage and temperature variations,” in *Proc. DAC*, 2020, pp. 1–6.
[20] D. Garyfallou, I. Tsiokanos, N. Evmorfopoulos, G. Stamoulis, and G. Karakonstantis, “Accurate estimation of dynamic timing slacks using event-driven simulation,” in *Proc. ISQED*, 2020, pp. 225–230.
[21] Z. Guo, T.-W. Huang, and Y. Lin, “Gpu-accelerated static timing analysis,” in *Proc. ICCAD*. IEEE Press, November 2020.
[22] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, “Gpu-accelerated critical path generation with path constraints,” in *Proc. ICCAD*, Virtual Conference, November 2021.
[23] Z. Guo, T.-W. Huang, and Y. Lin, “A provably good and practically efficient algorithm for common path pessimism removal in large designs,” in *Proc. DAC*, San Francisco, CA, December 2021.
[24] M. S. Golanbari, A. Gebregiorgis, F. Oboril, S. Kiamehr, and M. B. Tahoori, “A cross-layer approach for resiliency and energy efficiency in near threshold computing,” in *Proc. ICCAD*, 2016, pp. 1–8.
[25] D. Ma, X. Zhang, K. Huang, Y. Jiang, W. Chang, and X. Jiao, “Devot: Dynamic delay modeling of functional units under voltage and temperature variations,” *IEEE TCAD*, pp. 1–1, 2021.
[26] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, “Axbench: A multiplatform benchmark suite for approximate computing,” *IEEE Design Test*, vol. 34, no. 2, pp. 60–68, 2017.
[27] “Synopsys PrimeTime,” http://www.synopsys.com.
[28] M. Brandalero, A. C. S. Beck, L. Carro, and M. Shafique, “Approximate on-the-fly coarse-grained reconfigurable acceleration for general-purpose applications,” in *Proc. DAC*, 2018, pp. 1–6.
[29] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, “Open cell library in 15nm freepdk technology,” in *Proc. ISQED*. New York, NY, USA: Association for Computing Machinery, 2015, p. 171–178.