Handling Latch Loops in Timing Analysis with Improved Complexity and Divergent Loop Detection

Xizhe Shi[1*], Zizheng Guo[1,2*], Yibo Lin[1,2,3†] Runsheng Wang[1,2,3], Ru Huang[1,2,3]
[1]School of Integrated Circuits, Peking University [2]Institute of Electronic Design Automation, Peking University
[3]Beijing Advanced Innovation Center for Integrated Circuits
Email: xizheshi@stu.pku.edu.cn, {gzz, yibolin, r.wang, ruhuang}@pku.edu.cn

***Abstract*—Latch loops introduce feedback cycles in timing graphs for static timing analysis (STA), disrupting timing propagation in topological order. Existing timers handle latch loops by checking the convergence of global iterations in timing propagation without lookahead detection of divergent loops. Such a strategy ends up with the worst-case runtime complexity $O(n^2)$, where n is the number of pins in the timing graph. This can be extremely time-consuming, when n goes to millions and beyond. In this paper, we address this challenge by proposing a new algorithm consisting of two steps. First, we identify the strongly connected components (SCCs). Second, we implement parallelized arrival time (AT) propagation between SCCs while conducting sequential iterations inside each SCC. This strategy significantly reduces the runtime complexity to $O(\sum_i k_i^2)$ from the previous global propagation, where k_i is the number of pins in each SCC. Our timer also detects timing information divergent loops in advance, avoiding over-iteration. Experimental results on industrial designs demonstrate 12.31× and 7.3× speed-up over PrimeTime and OpenSTA on average, respectively.**

I. Introduction

STA plays an essential role in the electronic design automation (EDA) flow for assessing circuit timing under varying scenarios [1]. Therefore, as integral components of EDA tools, STA engines are vital for identifying timing violations at different design stages, aiding in circuit verification [2]. As sequential elements exhibiting distinct timing behaviors compared to flip-flops (FFs), latches are widely used in modern VLSI circuits for their numerous benefits, including reduced area, lower power, and time borrowing capability [3], [4].

The procedure of STA first converts the circuits to directed acyclic graphs (DAGs), and analyze them in a level-by-level manner following the topological order. However, the incorporation of latches into a circuit may result in the emergence of feedback loops. Unlike combinational logic loops that are likely to lead to divergent AT results, latch feedback loops may create convergent AT propagation, but destroy the directed acyclic nature of the original graph. This may cause failure of timing analysis or incorrect timing results [5].

In order to address feedback loops resulting from latches, subsequently referred to as latch loops, a series of methods have been employed in industry. In early industrial EDA tools, STA engines employed cycle cutting technology in the detected latch loops, selecting some timing arcs in loops as loop-breaker arcs to remove [6]. Although this makes the resulting graph acyclic, it prevents timing propagation in the selected broken timing arcs, which may turn out to be critical. As a result, such a method is fast but may be inaccurate.

To derive accurate timing reports considering latch loops, various academic research works have proposed different algorithms. The approach proposed in [7] computes circuit yields at the cost of additional extraction of constraint graphs by verifying the absence of positive or negative loops in the corresponding constraint graphs. In [8], graph decomposition methods are initially employed, with subsequent improvements made to the form of graph traversal. However, this heuristic algorithm inherently results in some loss of accuracy. The research in [9] improves accuracy by iteratively propagating timing information to analyze latch loop convergence. However, this method increases analytical complexity and requires additional buildup of the reduced timing graph (RTG). The research work in [10], optimizes RTG-based convergence detection, reducing iterations and runtime of STA.

Most of the mentioned methods improve accuracy but incur much higher time complexity, often quadratic, due to additional construction of RTGs or other auxiliary graphs. Therefore, these methods lack practicality and are not widely adopted in industry. As of today, leading-edge STA engines including PrimeTime [11] and OpenSTA [12] still use a vanilla iteration-till-converge approach to handle latch loops. Although this approach works well in practice, its worst theoretical complexity can also be up to $O(n^2)$.

In this work, we propose a novel algorithmic framework to accelerate the STA process for circuits with latch loops. Our timer, based on this framework, can ground-breakingly pre-detect divergent loops and outperforms widely used tools like PrimeTime and OpenSTA in runtime. We summarize our contributions as follows:

1) We propose a novel graph decomposition algorithm for circuits containing latch loops according to loop locality. Thanks to this partitioning, we achieve parallel waste-free latch AT propagation without accuracy loss.

* Equal contribution, † Corresponding author.

This project is supported in part by the Natural Science Foundation of Beijing, China (Grant No. Z230002), the National Science Foundation of China (Grant No. T2293701), and the 111 Project (B18001).

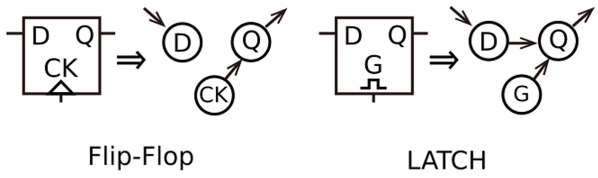

Fig. 1: Graph representation of Flip-Flops and latches in GBA.

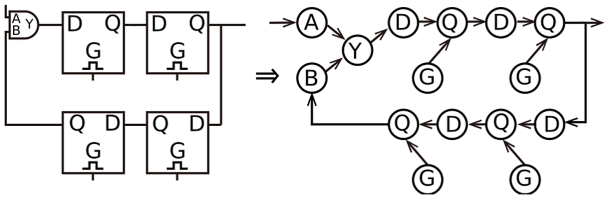

Fig. 2: Graph representation of latch loop in GBA.

2) We integrate an efficient algorithm using SCC-based iteration into loop analysis to enable early detection of divergent loops and prevent excessive iterations of AT propagation.
3) Our algorithm has provably better runtime complexity compared to prior works. We prove that our algorithm runs in $O(\sum_i k_i^2)$ where k_i denotes the number of pins in each SCC. It is much smaller than the $O(n^2)$ algorithm used by PrimeTime and OpenSTA where n is the full graph size.

We integrate our algorithm into an open-source STA engine and demonstrate an average speed-up of 12.31× over PrimeTime and 7.3× over OpenSTA on large circuit designs.

II. Preliminaries

A. Latch Loops in Graph-Based Timing Analysis

STA is typically deployed in two phases: graph-based analysis (GBA) and path-based analysis (PBA). GBA uses a graph-based approach to analyze timing across circuits, while PBA targets specific critical paths to identify and optimize timing violations. In GBA, the circuit is represented as a DAG, with nodes as pins and edges as arcs carrying timing information.

In latches, timing information such as AT at output pin can be derived from data input pin after timing checks due to their time borrowing property, which is different from FFs. So when representing latches in a graph, there exists an arc from data input pin to data output pin as shown in Figure 1.

Figure 2 illustrates how a latch loop can create a feedback loop in the extracted graph, making the conventional topological timing propagation approach based on DAGs inapplicable. Thus, circuits with latch loops require a distinct methodology.

B. AT Propagation and Convergence in Latch Loops

In STA, arcs in the circuit graph are annotated with minimum and maximum delays to account for process variations, creating the early/late split model for hold/setup violation checks. In the late case of the model, the propagation of timing information, such as ATs, via relaxation always preserves the larger possible values during the updating process [13]. However, once feedback loops exist in circuits, the propagation of AT must be iteratively updated until convergence. Divergence in this process will result in unbounded increases in ATs, leading to setup timing violations.

Unlike combinational loops, latch loops contain D-Q arcs of latches. The relaxation process of them is more specific, requiring setup time checks and potentially resulting in a decreased AT result due to the change of clock domains [7], [10]. The presence of decreased AT values is fundamental to the convergence of ATs during loop iterations, indicating that convergence is possible for latch loops.

C. Global Iteration for AT Propagation in Current Timers

In a DAG, AT propagation does not require repeated iterations; however, loops or SCCs consisting of multiple pins (MSCCs) necessitate them, until ATs converge or diverge.

Taking the late case as an example, current timers first break all D-Q arcs to ensure AT propagation on the DAG. Subsequent setup time checks determine if the Q pins' ATs are updated by the D pins, requiring re-updates of downstream ATs if true. Its worst-case complexity can be up to $O(n^2)$, leading to unacceptable runtimes with a high pin count [14]. Moreover, their iterative process lacks preemptive assessment of potential divergence scenarios and does not terminate early but iterates until a timing violation occurs.

Figure 3 reveals how AT propagates in global iteration in a latch loop chain. The number of AT updates per loop continues to grow linearly as the order of loops increases from left to right. Assuming each update requires a fixed amount of time, the total runtime will be the sum of these updates, increasing quadratically with the total number of loops.

III. Algorithms

To address the limitations of current timers in AT propagation in circuits with latch loops, we hope to introduce SCC-based AT iteration. According to graph theory, a directed graph can be decomposed into a collection of SCCs [15]. This generates a DAG of SCCs, after which the longest path algorithms can be applied to propagate AT inside each SCC, with AT iterations carried out in topological order for each SCC. In this case, ATs within each SCC converge before propagating into downstream SCCs, reducing the runtime overhead caused by repeated updates compared to the AT propagation in Figure 3.

Following these ideas, our algorithm first decomposes the extracted graph into SCCs. Subsequently, AT iterations are then confined inside each SCC to reduce complexity, while enabling parallel AT propagation between SCCs to further improve efficiency. The overall flow of our algorithm is shown in Algorithm 1 and Figure 4. Our algorithm will be implemented based on the source code of OpenTimer [16].

A. Graph Decomposition and SCC Head Pin Determination

To decompose the circuit graph into SCCs firstly, we adopt the ideas of Tarjan's algorithm, whereby the full graph is traversed using depth-first search (DFS) [15].

Algorithm 2 details the workings of graph decomposition. In this algorithm, an $index$ is assigned to each pin, with the

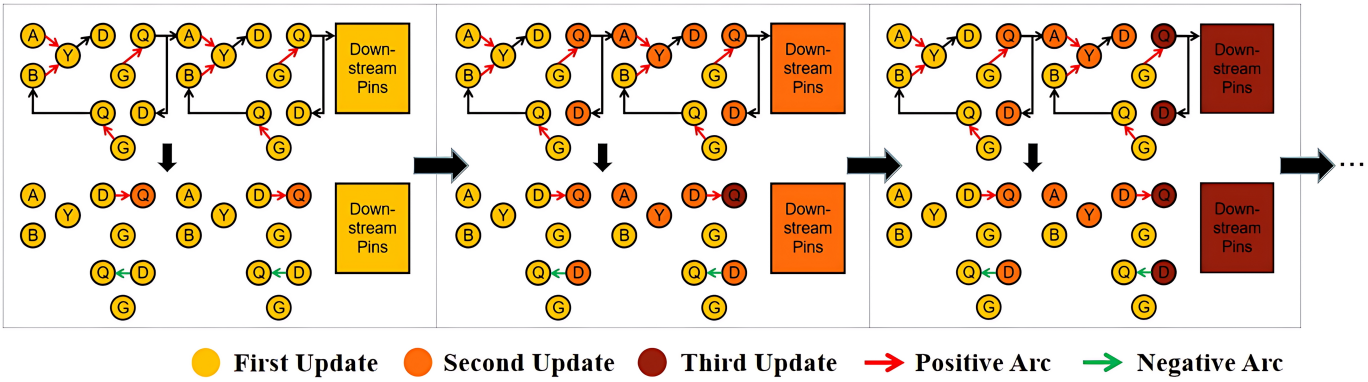

Fig. 3: Global AT propagation iterations of latch loop chains in current timers.

Algorithm 1: Overall-Flow-AT-Propagation(Graph G)

```
for each pin v in the circuit graph G do
    DFS-SCC(v)
for each pin v in the circuit graph G do
    DFS-Select-Head-Pin(v)
taskflow ← Build-AT-Prop-Taskflow()
execute(taskflow)
```

search order stored in the array dfn. The array low holds the earliest order of any pin in the $stack$ that can be traced back for pin u or its subtree. We invoke Algorithm 2 on all pins in the graph, achieving a linear complexity of $O(n)$, where n is the number of pins in the circuit graph [15]. Ultimately, the pins contained in each SCC are stored in the corresponding element of $sccs$. In turn, we also use $scc\_id[pin]$ to store the SCC number to which pin belongs.

Algorithm 2: DFS-SCC(Pin v).

```
index ← index + 1
dfn[v] ← index, low[v] ← index
Push(stack, v)
for each v's fanout arc e do
    u ← destination_pin[e]
    if u is not visited then
        mark u as visited
        DFS-SCC(u)
        low[v] ← min(low[v], low[u])
    else if u is in stack then
        low[v] ← min(low[v], dfn[u])
if low[v] = dfn[v] then
    initialize a new SCC scc
    while stack is not empty do
        u ← Pop(stack)
        Pushback(scc, u)
    assign scc_id[pin] for all pins in scc
    Pushback(sccs, scc)
```

Due to the property of OpenTimer's taskflow, each task must be assigned to an individual pin rather than an SCC. Therefore, a head pin is selected from each SCC to receive the AT iteration task in the whole SCC during AT propagation.

Algorithm 3 determines the head pin of each SCC by performing DFS on all graph pins, while ensuring each pin is visited once. During this process, each pin is checked to see if it is the first visited pin in its SCC; if so, it is labeled as the head pin and stored in $head\_pin[scc]$.

Algorithm 3: DFS-Select-Head-Pin(Pin v).

```
if v is not visited then
    mark v as visited
    scc ← sccs[scc_id[v]]
    if scc is not visited then
        mark scc as visited
        head_pin[scc] ← v
    for each v's fanout arc e do
        u ← destination_pin[e]
        if u is not visited then
            DFS-Select-Head-Pin(u)
```

B. Local Iteration of AT Inside Each SCC

To avoid global iteration of AT in the current STA engines discussed in Section II, which results in quadratic complexity, our algorithm localizes the AT iteration inside each SCC.

It is essential to clarify that the localization of the AT iteration occurs at the level of individual SCCs, as outlined in Algorithm 4. For single-pin SCCs, the AT is computed by relaxing all fanin arcs.

But for MSCCs, the iteration of AT inside each SCC is realized by an adapted SPFA. As described in Algorithm 4, all internal pins of the SCC must be enqueued into the pending queue $queues[scc]$, and the corresponding element in the array q_cnt, which tracks the number of times each pin is enqueued, should be initialized to one.

AT propagation is then executed for each pin in the queue, beginning with relaxations on its fanin arcs originating from other SCCs. This process marks the relaxed arc to prevent further relaxations, as the AT value of its source pin must have been fixed by the time it is relaxed (further instructions will be given in the subsection C). Therefore, excess relaxations more

Algorithm 4: AT-Prop(Pin v).

```
1  scc ← sccs[sccs_order[v]]
2  if size(scc) = 1 then
3      for each v's fanin arc e do
4          Relax arc e
5  else
6      for each pin u in scc do
7          Enque(queues[scc], u)
8          q_cnt[u] ← q_cnt[u] + 1
9      while queues[scc] is not empty do
10         u ← Pop(queues[scc])
11         for each u's unrelaxed fanin arc e do
12             from ← source_pin[e]
13             if scc_id[u]= scc_id[from] then
14                 Relax arc e
15         for each u's fanout arc e do
16             to ← destination_pin[e]
17             if scc_id[u]  = scc_id[to] then
18                 Relax arc e
19                 if AT[to] is changed && to is not in
                     queues[scc] then
20                     Enque(queues[scc], to)
21                     q_cnt[to] ← q_cnt[to] + 1
22                     if q_cnt[to] > size(scc) then
23                         Error(Divergent Loop!)
```

than once will not update the AT of the destination pin again. The necessary relaxations for the AT iteration, similar to those in SPFA, actually occur on the arcs between pins inside the SCC. For each pin, we relax its fanout arcs inside the SCC; if this alters the AT of a destination pin not in $queues[scc]$, we should get that pin enqueued for further iteration. This is due to the fact that the AT of this pin has not yet converged and the updated AT may affect the ATs of its fanout pins. This process will continue until the queue is emptied.

However, the number of queue entries for each pin must be limited to avoid infinite iterations. For possible ATs that do not converge, our algorithm employs a judgment mechanism similar to SPFA. If a pin enters the queue more times than the total number of pins in the SCC, it indicates both a non-converging AT and a divergent loop inside the SCCs.

According to [17], the complexity of algorithm 4 adopted from SPFA for a single SCC is $O(ke)$, where k is the number of pins and e is the number of arcs. Since the number of pins and arcs in timing graphs are often similar or linear, the complexity can be approximated as $O(k^2)$.

C. Parallel Propagation of AT Between SCCs

After the graph decomposition, the new graph can be viewed as a DAG of SCCs, and the iteration of AT inside each SCC in Algorithm 4 will be assigned to the head pin of each SCC as a task. As a result, AT iteration is confined inside each SCC, and the AT propagation between SCCs can be parallelized after determining the dependencies between tasks based on the connectivity relationships between SCCs.

Algorithm 5 is designed to assign the tasks of AT iteration to the head pin of each SCC and to ensure that the correct dependencies between tasks have been established before the parallelization carries out. In essence, if an arc is identified as originating from a pin in SCC A and pointing to a pin in SCC B, AT iteration of SCC B must be carried out after that of SCC A. This dependency stems from the characteristics of the SCC.

Algorithm 5: Build-AT-Prop-Taskflow()

```
1  for each pin v in G do
2      if v is head pin of sccs[scc_id[v]] then
3          tasks[v] ← Emplace(taskflow, AT-Prop(v))
4  for each pin v in G do
5      v_head_pin ← head_pin[sccs[scc_id[v]]]
6      for each v's fanin arc e in E do
7          u ← source_pin[e]
8          u_head_pin ← head_pin[sccs[scc_id[u]]]
9          Precede(tasks[u_head_pin], tasks[v_head_pin])
10 return taskflow
```

Proof. In a given SCC, each pin needs to relax all its fanin arcs originating from other SCCs in Algorithm 4. This requires ensuring that the AT results of sources pins of these fanin arcs have converged, otherwise the localization of the AT iterations cannot be guaranteed, leading to higher complexity. Also, if there exist arc a pointing from SCC A to SCC B and arc b pointing from SCC B to SCC A then A and B should have been merged into a larger SCC. □

D. Complexity Analysis

To justify the efficiency of our algorithm, we derive the following theory results:

Theorem 1. *The time complexity of the overall flow of the algorithm above is $O(\sum_i k_i^2)$, where k_i denotes the size of SCC numbering i.*

Proof. The time complexity has three parts, *Graph decomposition*, *SCC head pin Selection*, and *AT propagation*. Graph decomposition and head pin selection both take $O(n)$ time, as they involve a DFS traversal of the entire graph. The time complexity of AT iteration for each SCC with k_i pins is $O(k_i^2)$, and overall complexity is their sum, $O(\sum_i k_i^2)$. Since $\sum_i k_i$ is equal to n, it is clear that $\sum_i k_i^2$ is greater than n. Therefore, the overall time complexity of the algorithm is $O(\sum_i k_i^2)$. □

It is clear that $\sum_i k_i^2 < (\sum_i k_i)^2 = n^2$, so this method will have a complexity advantage in the case of multiple MSCCs considering the cost of graph decomposition.

IV. Experimental Results

A. Experimental Setup

Our algorithm is implemented in C++ on top of the open-source STA engine OpenTimer [18]. Experiments are con-

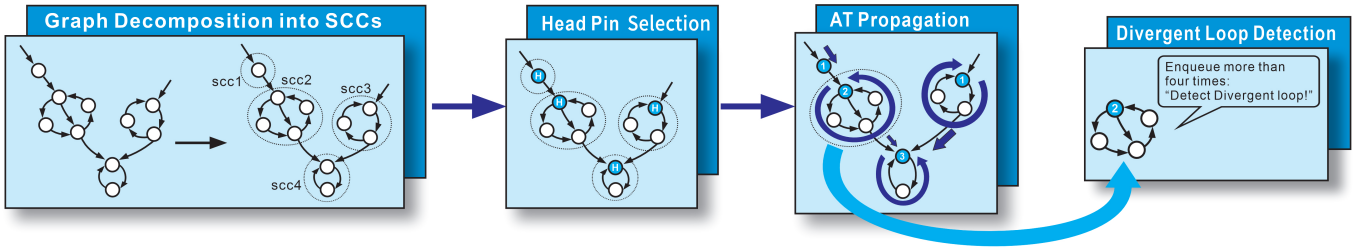

Fig. 4: Overall flow of our algorithm on AT propagation.

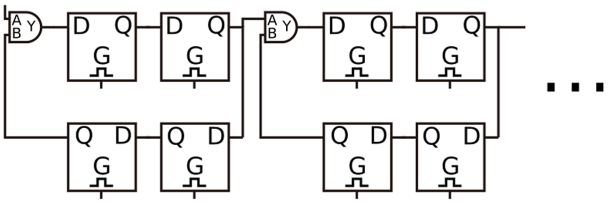

Fig. 5: Chain of n latch loops.

TABLE I: Statistics of benchmarks.

Benchmark	#Gates	#Pins	#MSCCs	#APPS	#Endpoints
SDLL	5	23	1	18	4
ELLC_1000	72000	216017	8000	18	32000
ELLC_2000	144000	432017	16000	18	64000
ELLC_3000	216000	648017	24000	18	96000
ELLC_4000	288000	864017	32000	18	128000
ac97_ctrl_latch	30482	90293	564	18	10170
aes_core_latch	46551	135544	90	97	2828
b19_iccad_latch	513256	1573945	407	2903	14388
leon2_iccad_latch	3250738	8710527	5210	772	306762
leon3mp_iccad_latch	2513450	6807681	6249	521	225678
netcard_iccad_latch	3011438	8052365	6097	643	203662
vga_lcd_latch	285808	815885	4792	27	53182
vga_lcd_iccad_latch	523534	1374733	5907	39	36558

APPS: Average number of pins per MSCC.

ducted on a 64-bit Linux machine with 96 cores Intel Xeon Gold 6248R CPU at 3.00 GHz and 502 GB memory, at different threads respectively. Previous academic methods lack practicality and industry adoption, as discussed in Section I, so we compare our timer with two advanced STA engines: PrimeTime [11] and OpenSTA [12].

We validate our algorithm on two sets of benchmarks. Since our primary objective is to examine the acceleration of AT propagation, we preset the cell delays of these benchmarks in each timer to eliminate discrepancies arising from different delay computation models, and meanwhile setting the net delays to zero for simplicity as well. In the first set, five special cases were constructed to validate the detection of divergent loops and to demonstrate the superiority of our algorithm in time complexity. In the second set, to examine the advantages of our timer on real designs, we constructed eight industrial circuits from TAU contests [19] using the method in [20], with some other changes to the netlist, lib and sdc files.

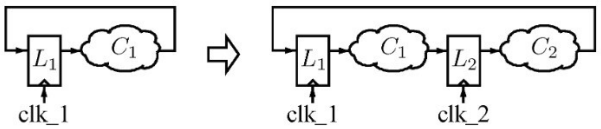

Fig. 6: Circuit construction for industrial benchmarks [10].

60% FFs in the initial circuits were converted into latches, and then latches and combinational logic were duplicated, as illustrated in Figure 6. The clocks clk_1 and clk_2 formed a two-phase clock scheme and were set to inverted clocks with clock phase shift T/2 and duty cycle 0.5. We have uploaded two examples that allow readers to reproduce and observe the prolonged PrimeTime runtimes, which stem from the presence of divergent loops and the methodology adopted by the timer in handling latch loop circuits. [1].

For accuracy check of our timer, it is evaluated by comparing the Mean Absolute Error (MAE) of each endpoint slack between our timer and PrimeTime. When testing runtime of three timers, it is assessed by execution time of command **report_timing** in PrimeTime and our timer, and **report_checks** in OpenSTA, and they can be monitored by command `time` in TCL tools. Note that all the runtime values reported are collected by averaging 10 runs of each experiment.

B. Accuracy and Runtime Results of Industrial Benchmarks

We finally compare the runtime results obtained by three timers at different threads, as shown in Table II. As for accuracy, we can well match PrimeTime's results on endpoint slacks, with all MAEs less than 0.01%.

In eight industrial benckmarks, as depicted in Figure 7 and Table II, our timer exhibits superior accuracy with an average speedup ratio of 12.31× and 7.3× at eight threads compared to PrimeTime and OpenSTA, respectively. Despite including sequential components, such as the non-parallelized graph decomposition, our timer still achieve an average 2× speedup at 8 threads compared to single-thread scenarios, allowing it to outperform the baselines, even their fully parallelized versions. Concurrently, especially for graphs with a greater number of MSCCs such as `leon2_iccad_latch`, `leon3mp_iccad_latch` and `netcard_iccad_latch`, our timer exhibits a more pronounced speedup up to around 20× and 15× at eight threads, which aligns with the effect of localized AT iterations inside SCCs.

As observed in the runtime speedup at multi-threaded tests, our timer successfully implements parallelization for AT propagation between SCCs. Figure 8 shows runtime values for the benchmarks `leon2_iccad_latch` and `leon3mp_iccad_latch` at different threads. It is clear that runtimes of three timers saturate at 8 threads and our timer exhibits good parallelism.

[1]https://github.com/xiaoshixuexi/Latch-Loop-Examples

TABLE II: Overall efficiency comparison between PrimeTime, OpenSTA and our timer.

Benchmark	PrimeTime_RT		OpenSTA_RT		Ours_RT		PrimeTime_RTR		OpenSTA_RTR		Ours_TRTR	
	1 Thread	8 Threads	1 Thread	8 Threads	1 Thread	8 Threads	1 Thread	8 Threads	1 Thread	8 Threads	1 Thread	8 Threads
SDLL	81.65	164.26	36.40	53.08	0.005	0.008	16330	20533	7280	6635	0.63	1.00
ELLC_1000	756.61	242.67	155.68	65.31	4.31	2.36	175.56	102.83	36.12	27.67	1.83	1.00
ELLC_2000	2942.79	564.70	713.05	306.55	8.77	4.80	335.55	117.64	81.31	63.86	1.82	1.00
ELLC_3000	6562.99	1446.52	1616.72	762.08	12.89	6.45	509.15	224.27	125.42	118.15	2.00	1.00
ELLC_4000	11907.63	2453.88	3042.37	1296.47	16.71	8.21	712.61	298.89	182.07	157.91	2.04	1.00
ac97_ctrl_latch	10.88	4.32	7.47	3.54	3.24	1.96	3.36	2.20	2.22	2.09	1.65	1.00
aes_core_latch	9.08	3.84	5.87	3.02	3.39	1.97	2.68	1.94	1.73	1.53	1.72	1.00
b19_iccad_latch	371.72	117.07	184.05	74.72	36.64	20.45	10.15	5.74	5.02	3.65	1.79	1.00
leon2_iccad_latch	4790.84	1484.87	2642.56	966.25	160.01	69.57	29.94	21.54	16.51	13.89	2.30	1.00
leon3mp_iccad_latch	4125.82	1219.47	2275.04	867.89	142.14	63.74	29.03	19.13	16.01	13.61	2.23	1.00
netcard_iccad_latch	3854.54	1195.05	2202.96	707.08	135.64	63.09	28.42	18.94	16.24	11.21	2.15	1.00
vga_lcd_latch	235.51	77.01	159.33	59.78	16.32	8.66	14.43	8.89	9.76	6.90	1.88	1.00
vga_lcd_iccad_latch	416.11	123.75	233.71	87.90	29.21	15.92	14.25	7.77	8.00	5.52	1.83	1.00

RT: Runtime in seconds. **RTR**: Runtime ratio of this timer to our timer at the same thread.
TRTR: Runtime ratio of this thread number to 8 threads in the same timer.

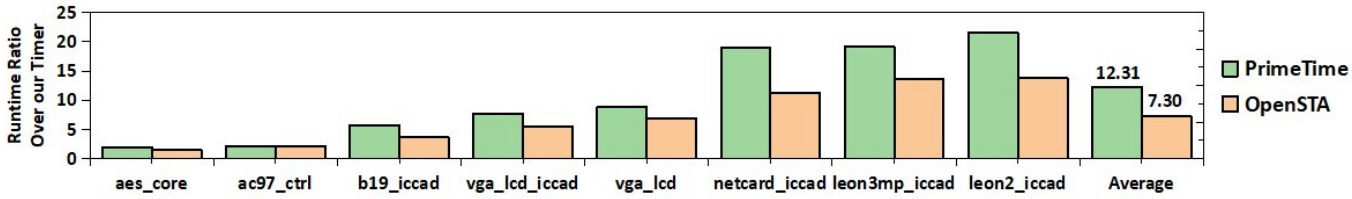

Fig. 7: Runtime ratio of PrimeTime and OpenSTA over our timer at 8 threads.

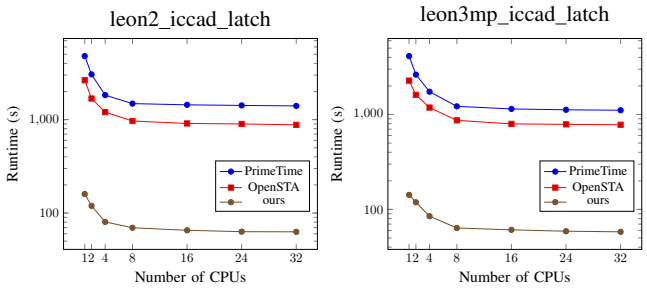

Fig. 8: Runtime values at different numbers of threads for three timers on two benchmarks.

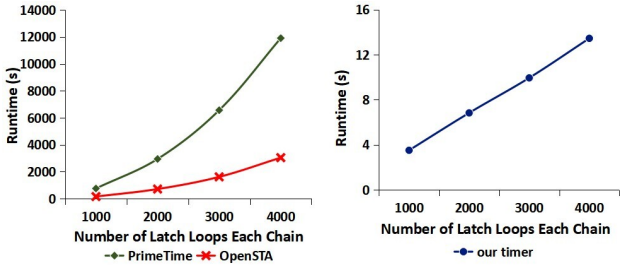

Fig. 9: Runtime result of four ELLC cases at single thread.

C. Divergent Loop Detection

Now, we examine the results on the first set of benchmarks in Table II. Figure 2 illustrates the circuit graph for the first special case, named single-divergent-latch-loop (SDLL), which consists of four latches and an AND gate. This circuit is designed to feature a divergent loop with a large clock period and a small total loop delay. As shown in Table II, over-iteration in PrimeTime and OpenSTA results in runtimes ranging from 50 to 300 seconds, while our timer detects the divergent loop in microseconds. The presence of the divergent loop leads to timing violations and unnecessary AT iteration propagation time, which our method effectively mitigates. This approach not only reduces the runtime but also improves accuracy by quickly identifying and handling such timing anomalies.

D. Analysis on Cases of Eight-Latch-Loop-Chain

Figure 5 shows the circuit designs of the remaining four special cases named eight-latch-loop-chain (ELLC). These benchmarks all consist of eight latch loop chains, each chain of which only differs in the number of latch loops.

As can be observed in Table II and Figure 9, the runtime of our timer increases in a linear relation to the number of loop chains, achieving a speedup of hundreds of times in processing these circuits in comparison to PrimeTime and OpenSTA in single-threaded experiments. The runtimes of the latter two timers, on the other hand, demonstrate a clear square relationship, which matches our statement in Section II.

V. Conclusion

In this work, we propose a novel STA algorithm for circuits containing latch loops. By strategically partitioning the graph into SCCs and leveraging parallelized AT propagation, we make full use of loop locality in latch-enabled circuits. The innovative incorporation of the SPFA for local iteration inside SCCs and the pre-identification of divergent loops have reduced the worst-case complexity of the algorithm to $O(\sum_i k_i^2)$ where k_i is the SCC sizes. Experimental evaluations on industrial designs confirm the superiority of our timer, with an average speed-up of 12.31× over PrimeTime and 7.3× over OpenSTA. Our future work includes integrating our algorithms into timing analysis under multiple clock domains, as well as applying it in automatic timing borrowing flows.

REFERENCES

[1] K. Kang, B. C. Paul, and K. Roy, "Statistical timing analysis using levelized covariance propagation," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 764–769.

[2] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media, 2009.

[3] S. Paik, L.-e. Yu, and Y. Shin, "Statistical time borrowing for pulsed-latch circuit designs," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2010, pp. 675–680.

[4] H. Cheng, X. Li, Y. Gu, and P. A. Beerel, "Saving power by converting flip-flop to 3-phase latch-based designs," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 574–579.

[5] N. Xiromeritis, S. Simoglou, C. Sotiriou, and N. Sketopoulos, "Graph-based sta for asynchronous controllers," in *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2019, pp. 9–16.

[6] S. Simoglou, C. Sotiriou, and N. Blias, "Timing errors in sta-based gate-level simulation," in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE, 2020, pp. 1–2.

[7] R. Chen and H. Zhou, "Statistical timing verification for transparently latched circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1847–1855, 2006.

[8] X. Yuan and J. Wang, "Statistical timing verification for transparently latched circuits through structural graph traversal," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2010, pp. 663–668.

[9] L. Zhang, J. Tsai, W. Chen, Y. Hu, and C. C.-P. Chen, "Convergence-provable statistical timing analysis with level-sensitive latches and feedback loops," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, 2006, pp. 941–946.

[10] B. Li, N. Chen, and U. Schlichtmann, "Statistical timing analysis for latch-controlled circuits with reduced iterations and graph transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 11, pp. 1670–1683, 2012.

[11] Synopsys, *PrimeTime, version: S-2021.06-SP1*, Synopsys, Inc., Mountain View, California, USA, 2021, synopsys, Inc. Software. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html

[12] "OpenSTA, version: 2.5.0," https://github.com/abk-openroad/OpenSTA.

[13] R. Chen, L. Zhang, V. Zolotov, C. Visweswariah, and J. Xiong, "Static timing: back to our roots," in *2008 Asia and South Pacific Design Automation Conference*. IEEE, 2008, pp. 310–315.

[14] M. Panda and A. Mishra, "A survey of shortest-path algorithms," *International Journal of Applied Engineering Research*, vol. 13, no. 9, pp. 6817–6820, 2018.

[15] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[16] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. ICCAD*. IEEE, 2015, pp. 895–902.

[17] "Segmented spfa: An improvement to the shortest path faster algorithm," https://konaeakira.github.io/posts/segmented-spfa-an-improvement-to-the-shortest-path-faster-algorithm.html.

[18] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–786, 2021.

[19] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 882–889.

[20] T. G. Szymanski, "Computing optimal clock schedules," in *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*. IEEE, 1992, pp. 399–404.