Efficient Critical Paths Search Algorithm using Mergeable Heap

Kexing Zhou*
CECA, CS Department
Peking University
Beijing, China
zhoukexing@pku.edu.cn

Zizheng Guo*
CECA, CS Department
Peking University
Beijing, China
gzz@pku.edu.cn

Tsung-Wei Huang
ECE Department
University of Utah
Salt Lake City, USA
tsung-wei.huang@utah.edu

Yibo Lin†
CECA, CS Department
Peking University
Beijing, China
yibolin@pku.edu.cn

***Abstract*—Path searching is a central step in static timing analysis (STA). State-of-the-art algorithms need to generate path deviations for hundreds of thousands of paths, which becomes the runtime bottleneck of STA. Accelerating path searching is a challenging task due to the complex and iterative path generating process. In this work, we propose a novel path searching algorithm that has asymptotically lower runtime complexity than the state-of-the-art. We precompute the path deviations using mergeable heap and apply a group of deviations to a path in near-constant time. We prove our algorithm has a runtime complexity of $O(n \log n + k \log k)$ which is asymptotically smaller than the state-of-the-art $O(nk)$. Experimental results show that our algorithm is up to 60× faster compared to OpenTimer and 1.8× compared to the leading path search algorithm based on suffix forest.**

I. Introduction

Static timing analysis (STA) is a pivotal step in the overall design flow [1]. Based on the circuit design and delay annotations of the timing arcs, STA evaluates the setup/hold timing performance of the circuit under best-case and worst-case scenarios. The result of STA is a set of data paths with the largest timing violations. These paths are then used by the designers to optimize the design. In real-case STA scenarios, designers often require the STA engine to generate 10K or even 100K individual paths [2], [3]. On a design with millions of gates and nets, the search for such a large number of paths in STA takes several hours to complete, slowing down the entire circuit design process.

Recent works have integrated different graph-based algorithms to reduce the path searching runtime [4], [5], [6], [7], [8], [9]. Most of them are based on a progressive approach that generates path deviations in increasing order of timing violations, i.e. path slack. In this approach, each time a path is generated, all nodes on the path and their incoming edges are enumerated exhaustively, which is the central computational challenge of the path searching process. Huang *et al* [4], [5] adopt multi-core CPU-based parallelism to explore the deviation paths in parallel and have shown 4× speedup. However, their runtime scalability saturates at around 8–16 CPUs. Guo *et al* [9] present a parallel deviation path exploration algorithm on GPUs based on a technique called *suffix forest*, and delivers 25–45× speedup compared to CPU parallelism. However, they need to enumerate even more deviation paths than [5] because their parallelization scheme prevents early pruning of unwanted paths.

Despite recent progress on path searching acceleration, state-of-the-art algorithms still need to enumerate all possible deviations for each and every critical path generated. This bounds their runtime complexity to $O(nk)$, where n is the size of the circuit graph, and k is the number of paths requested to generate. As a result of this runtime complexity, when a large-scale circuit design (large n) meets a large demand of path counts (large k), current algorithms will fail to handle the STA task in a reasonable amount of time. To reduce long runtimes of path searching, we need a novel search algorithm that can improve the runtime in both practical and theoretical aspects. However, designing an *asymptotically* faster algorithm for path searching in practice is very challenging. The previous path searching algorithm includes a complex and iterative path expansion process, which requires a very strategic algorithm design to reduce the search space. Algorithms with asymptotically lower runtime complexity may exhibit a large time constant, sometimes even slower than their opponent with higher runtime complexity [10]. Thus, it is difficult to design and implement such an algorithm of practical interest that can benefit broad STA software.

In this work, we propose a new path searching algorithm that is asymptotically better than the state-of-the-art path searching algorithms by leveraging two novel data structures, *leftist heap* and *skew heap*. We summarize our technical contributions as follows:

1) We store all deviations into a *persistent mergeable heap*, which is a data structure supporting fast merging and duplicating. This allows us to apply a group of deviations in near-constant time.
2) We incorporate a novel *deviation preprocessing* step, in which we precompute the deviation paths for each prefix tree node instead of each new path. In this way, we speed up the path searching process by using precomputed deviation paths.
3) We prove that our algorithm runs in time complexity $O(n \log n + k \log k)$, where n is the size of the circuit graph and k is the number of paths generated. This is asymptotically smaller than OpenTimer's $O(n^2 k)$

*Equal contribution, ordered by first names. †Corresponding author.

algorithm and the state-of-the-art $O(nk)$ algorithm based on suffix forest.

We have evaluated our algorithm on a set of industrial designs and compared it with two state-of-the-art algorithms, OpenTimer [5] and the suffix forest algorithm [9]. The results show that our algorithm outperforms OpenTimer [5] up to 60× faster, and the CPU implementation of suffix forest algorithm [9] up to 1.8× faster. Our algorithm provides a significant enhancement to the previous algorithms in both theoretical and practical aspects.

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

II. Preliminaries

A. Path Searching in Static Timing Analysis

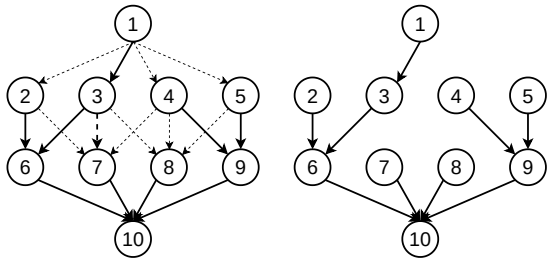
In STA, a circuit is represented as a directed acyclic graph (DAG), where nodes denote pins and edges denote interconnections between pins. In the circuit graph, there are flip-flops (FF) driven by a common clock source through a clock network. To account for process variations, the edges in the circuit graph are annotated with a minimum delay and a maximum delay, which is called the early/late split model.

In path-based STA, the input pins of FFs are called timing endpoints. A timing path starts from the clock source and then goes through a launching FF, and finally ends at the timing endpoint of a capturing FF. In setup checking, a longer path indicates slow signal propagation, which may make the signal arrives late at its timing endpoint. In hold checking, a shorter path indicates fast signal change which may interfere with the last signal at the capturing FF. The slack value of a path quantifies how much a timing path violates a setup/hold constraint, which is defined as follows [1]:

$$slack^{\texttt{setup}} = rat^{\texttt{late}} - at^{\texttt{late}},$$
$$slack^{\texttt{hold}} = at^{\texttt{early}} - rat^{\texttt{early}},$$

where $slack^{\texttt{setup/hold}}$ denotes respectively the setup/hold slack of a path, $at^{\texttt{early/late}}$ denotes the signal arrival time at the timing endpoint, and $rat^{\texttt{early/late}}$ denotes the required arrival time at the timing endpoint.

In the above definition, negative slacks indicate timing violations. To evaluate the timing correctness of the circuit under different circumstances, we are interested in the top-k paths with the smallest slacks. The required arrival time is only related to the setup/hold constraint settings and the clock period, which are constants with respect to timing endpoints. Therefore, we are interested in paths to each timing endpoint that have the earliest/latest signal arrival times. Equivalently, the task is to find the top-k shortest/longest paths on the delay-annotated circuit graph.



Path	Encoding
1,3,6,10	{}
1,3,8,10	(3,8)
1,4,8,10	(1,4),(4,8)
1,2,7,10	(1,2),(2,7)

Fig. 1: Illustration of the relation between paths and the suffix tree. In the top figure, the solid lines represent the suffix tree. In the bottom table, we give path examples which are represented as a series of non-suffix-tree edges.

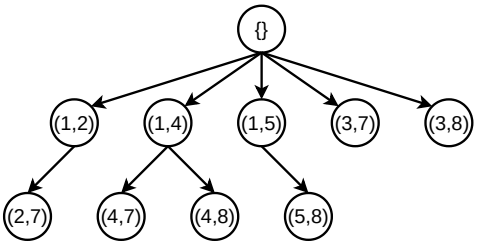


Fig. 2: The prefix tree corresponding to Figure 1. The root node is followed by all non-tree edges in suffix tree. Other nodes are succeeded by their reachable edges on the suffix tree. On this prefix tree, the shortest path "1,3,6,10" is the root node, the path "1,3,8,10" is the rightmost node, and the path "1,4,8,10" is the third node on the bottom, etc.

B. Problem of the State-of-the-Art Path Searching Algorithm

In this section, we present the details of the state-of-the-art path searching algorithm, OpenTimer [5] and suffix-forest [9], and discuss their weaknesses. In these algorithms, the key data structures are the *suffix tree* and the *prefix tree*. A suffix tree is essentially a shortest-path tree rooted at the timing endpoint T. Each path from a starting point S to T corresponds to a unique sequence of deviation edges, which is illustrated in Figure 1. A prefix tree is the search tree for such edge sequences, as illustrated in Figure 2. The two complementary data structures reduce the search space of path-based STA because the edge sequence is a compact and effective way to represent paths, which is suitable for searching. Efficient dynamic programming approaches are used to construct the suffix tree of a given circuit graph [4].

The path searching algorithm starts with the root of the prefix tree, i.e. the shortest path. Then, it iteratively builds the prefix tree by exploring an unvisited prefix tree node with the shortest path length. The iteration continues until k nodes are discovered, which are the top-k shortest paths. For completeness, we show the pseudocode of the above process

in Algorithm 1.

In this path searching process, the central step is to find all descendants of a prefix tree node, which we call *path expansion*. This step is repeated for k times, and it may involve a large number of prefix tree nodes, even many of them would never contribute to the top-k paths. As the maximum number of prefix tree nodes is proportional to the number of edges in the graph, which is $O(n)$, this step becomes the runtime bottleneck of path searching.

Algorithm 1: Baseline top-k path searching algorithm.

```
1 q ← an empty min-heap of prefix tree nodes;
2 Push the root node {} into q;
3 for d = 1, 2, ..., k do
4     x ←Pop the current shortest path from q;
5     Output the path represented by prefix tree node x;
6     for s in successors of x do
7         Push s into q;
```

III. Algorithms

To address the drawback of OpenTimer [5] and the suffix forest algorithm [9], we need to avoid generating a large number of deviations each time we find a new path. Specifically, instead of enumerating all of them iteratively, our algorithm precomputes the deviations and arranges them into a heap. In this way, each time we expand a new prefix tree node, all we need to do is to link this heap into our solutions. The heaps of deviations can be efficiently computed by incorporating their inheritance relationship and can be iteratively built up on the suffix tree.

The main process of our algorithm is shown in Algorithm 2. We first compute the suffix tree of the circuit graph. Then, we build heaps of deviations for all nodes on the suffix tree. Finally, we traverse the prefix tree to find the top-k shortest paths. Each step of our algorithm is explained in detail in the following sections.

Algorithm 2: Find-k-Shortest-by-Heap(G, k)

```
1 compute suffix tree of graph G
2 for each node u in G do
3     Make-Heap(u)
4 Find-k-Shortest()
```

A. Deviation of an Edge

The suffix tree converts each path into a short sequence of non-suffix-tree edges. Each non-suffix-tree edge makes the path length larger by a certain amount. This amount is independent of the path it is applied to. In path-based STA, such a contribution is called deviation [4]. It is defined as follows:

Definition 1. Let $dist[u]$ denote the shortest distance from node u to the sink T. The deviation of edge (u, v) is:

$$D(u, v) = -dist[u] + weight(u, v) + dist[v]$$

The independence of the deviation is shown in the following theorem:

Theorem 1. *For each path P from source node S to the sink node T. Let $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ be its corresponding non-suffix-tree edges. Then the path length $L(P)$ is equal to*

$$L(P) = dist[S] + \sum_i D(u_i, v_i)$$

The detailed proof of Theorem 1 can be found in [4]. Since $dist[S]$ is a path-independent constant, the deviation of each edge indicates the contribution of this edge to the path. Therefore, the value of a deviation $D(u, v)$ can be used as a ranking indicator and also as a weight for each node of the prefix tree.

B. Persistable Heaps

Persistable heap supports basic heap operations: `push`, `pop`, `top` and `merge`. For any of these operations, the persistent heap algorithm does not modify the heap in place, but generates a new version of heap instead. Each operation can take place both in the latest version and the previous version. Many widely-used heaps have corresponding persistable versions. In this work, we implement two different types of heaps, *leftist tree* and *skew heap*. The leftist tree is theoretically *mergeable* and *persistable*. For a leftist tree of size m, the time and space complexity of a merge operation is $O(\log m)$. The skew heap is not theoretically persistable, because it has an amortized time complexity based on potential analysis. For a skew heap of size m, the time and space complexity of a merge operation is $O(m)$. However, in practice, the skew heap runs very fast because it is rare to hit its runtime upper bound. Because of its simple implementation, it has a smaller runtime constant, making it another potential choice of heap we use in our algorithm.

In this work, we only use the merge operation of persistable heaps. The pseudocode for leftist tree operations can be found in [11].

C. Prefix Tree Rearrangement

In our algorithm, we arrange all successors of a prefix tree node into a precomputed heap. Each time we process a new prefix tree node, we no longer enumerate all its successors, but instead only link the heap roots to our solutions. Analogously, this can be seen as a rearrangement to the prefix tree topology. For each node x, we rearrange all the successors into a heap and connect the heap root to x. Since the successors are all derived from an identical prefix, we build a small root heap with deviation as the keyword. Figure 3 gives an example.

As long as we ensure that monotonicity in the prefix tree preserves after the transformation, i.e., the path length of a son must be longer than the parent, the path searching process shown in Algorithm 1 is still correct for our new topology.

Theorem 2. *After rearrangement, the number of successors of each node is no greater than 3. And the tree after rearrangement satisfies monotonicity.*

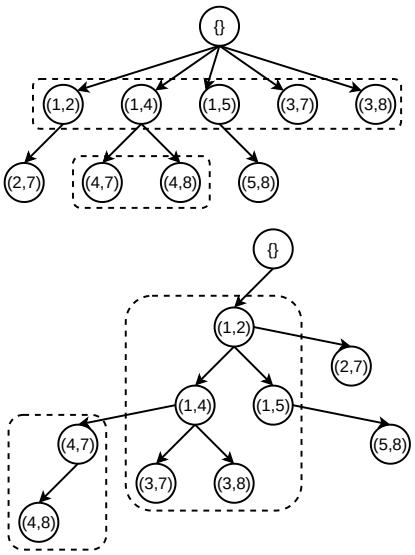


Fig. 3: Rearrange successors into a heap. The structure of the prefix tree is shown in the top figure. The root node of the prefix tree represents the empty sequence. The successors of the root node are arranged in ascending order from left to right according to deviation. We arrange them in the form of a heap to get the dashed box in the center of the bottom figure. Similarly, the successors of the node $(1, 4)$ are arranged in the dashed box to the left. After rearrangement, the number of successors of each node is no greater than 3, which is suitable for enumeration.

Proof. There are at most 3 children of a node x, the left son, the right son, and the link to the heap of x. According to the properties of the heap, the path length of the son must be equal to or greater than that of the father, so the monotonicity still satisfies. □

We denote the heap top of each node by *Successor*(x), and *Heap-Left*(x), *Heap-Right*(x) are the heap children of node x. The rearranged search algorithm is shown in Algorithm 3. In Lines 6–9, instead of adding all the successors one by one to the queue as in Algorithm 1, we only need to add the three nodes mentioned in the Theorem 2.

Algorithm 3: *Find-k-Shortest()*

```
1 q ← createPriorityQueue()
2 Enque(q, {}) push the prefix tree root into the queue
3 for d = 1, 2, ..., k do
4     x ← Deque(q)
5     E(x) ← compute the suffix encoding of x
6     output Convert-Suffix-Encoding-to-Path(E(x))
7     Enque(q, Successor(x))
8     Enque(q, Heap-Left(x))
9     Enque(q, Heap-Right(x))
```

D. Deviation Preprocessing

Although we can reduce the number of deviations by arranging them into a heap, for each node, it takes $O(n)$ time to compute its deviations. To speed up this procedure, we take the inheritance relationship of deviation on the suffix tree and use persistable heaps to preprocess deviation in near-constant time. In the following theorem, we give the successor rule, which implies the inheritance relationship of deviations.

Theorem 3. *Let x be a prefix tree node, and (u, v) be the corresponding edge. A node y with corresponding edge (u', v') is a successor of x if and only if:*

1) *u' is the ancestor of v in the suffix tree.*
2) *(u', v') is non-tree edge in the suffix tree.*

Theorem 3 states that the successors depends only on the end points of the edge, that is, each node in the suffix tree corresponds to a list of successors. And if two suffix tree node are adjacent, their successor list are almost identical to each other. Specifically, if x is the parent of y in the suffix tree, we can obtain y's successor list by adding the edges with endpoint y to the list of x. We use persistable heaps to accomplish this procedure. The details are shown in Algorithm 4. Since persistable operations are memory intensive, for each node x, we first build its connected edges into a binary heap, and then persistently merge it with its parent in the suffix tree. To demonstrate, Figure 4 gives an example. To obtain node 1's heap, we first build the edge into a heap and merge it persistently with 3's heap.

Algorithm 4: Make-Heap(u)

```
1  if heap[u] has already been built then
2      return heap[u]
3  buffer ← {}
4  for each edge (u, v) starting at u do
5      if (u, v) not in suffix tree then
6          buffer ← buffer ∪ {(u, v)}
7  hp1 ← build buffer into a heap
8  hp2 ← Make-Heap(Suffix-Tree-Father(u))
9  heap[u] ← Heap-Merge(hp1, hp2)
10 return heap[u]
```

E. Complexity Analysis

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

Theorem 4. *The time complexity of the algorithm above is $O(n \log n + k \log k)$. Where n denotes the size of the circuit graph and k denotes the number of paths generated.*

Proof. The time complexity has three main parts, *heap building*, *merging*, and *top-k path searching*. Using the linear heap building algorithm, it takes only $O(n)$ time to build the heap. The heap merge operation is performed at most $O(n)$ times, each time the heap size does not exceed n. Using persistable heaps, the complexity of a merge operation is $O(\log n)$. So

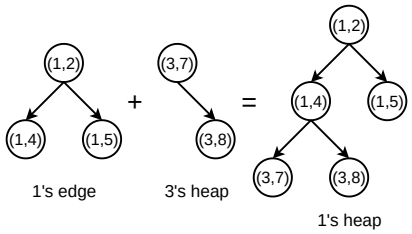


Fig. 4: Build heap persistently. In the Make-Heap algorithm, we have to calculate the heap corresponding to 1 node. First, we find the father of node 1 in the suffix tree, node 3. Then recursively build the heap of node 3. Next, we build a heap from the edges starting at node 1. Finally, we merge them together persistently to get the heap of node 1. Since it takes only linear time to build a heap, building heaps first and then merging can reduce the complexity.

the total time complexity is $O(n \log n)$. From Algorithm 3, we can clearly find that the time complexity of finding the k-shortest path is $O(k \log k)$. □

And for space complexity, we have:

Theorem 5. *The space complexity is $O(n \log n + k)$, where n,k are defined as above.*

Proof. The space complexity has two main parts, *the persistable heap*, and *the search queue*. For the persistable heap, the space complexity is equal to its time complexity, $O(n \log n)$. For the search queue, there are at most $3k$ push operations. So the size of the queue is $O(k)$. □

For common circuit designs, the number of nodes and the number of edges will not differ too much. The symbol n represents both the number of points and the number of edges. As stated in the proof, our algorithm has a fairly good asymptotic complexity.

F. Performance Optimizations

Since persistable data structures take a longer time to build and require a lot of memory, we propose the following method to optimize the performance of the algorithm:

Delayed heap building. We build the heap of node u when we need to access its successor. With this optimization, we won't create useless heaps. Because in some cases, such as designs with a large number of nodes, not all nodes exist in the top-k shortest paths, this optimization can improve the speed.

Memory pooling. When the size of the heap grows large, dynamic memory allocation becomes very time-consuming. We use memory pools to optimize node allocation. As the heap size is $O(n \log n)$, we acquired $\alpha \cdot n \log n$ nodes each time when the memory runs out.

IV. Experimental Results

We implement our algorithm in C++ on top of the open-source STA engine OpenTimer [5]. We conduct the experiments on a 64-bit Linux machine with 40 cores Intel Xeon CPU at 2.10 GHz and 512 GB memory. We compare our algorithm with OpenTimer [5] and the suffix forest algorithm in [9]. The suffix forest algorithm was proposed to run on GPUs and we have reimplemented it on CPUs to compare it with ours. We evaluate all algorithms on large industrial designs from TAU contests [2], [3]. We address the performance benefit under different numbers of paths, including 100, 100K, and 1M, and different types of persistable heaps, including skew heap and leftist tree. Each experimental configuration is performed 10 times to obtain the average runtime. The memory pooling parameter α in Section III-F is set to $1/1000$.

Table I shows the detailed performance comparison across the benchmarks. The size of the circuit designs is represented in the number of endpoints and pins in the table. Among these benchmarks, `leon2`, `leon3` and `netcard` are three largest ones, and the others are smaller to medium scale circuits. For large-scale circuits, our algorithm has superior performance– the number of successor nodes on the prefix tree is very large, and optimizations on the enumeration of successors become significant. For small and medium-sized circuits, such as `wb_dma`, and `vga_lcd`, due to the larger runtime overhead of the deviation preprocessing and persistable heap operations, our algorithm runs mostly slower. However, such an overhead is only a few hundred milliseconds, which is only a small portion of the runtime.

Among different algorithms in Table I, OpenTimer's original algorithm is the least efficient, because it is the one with $O(n^2k)$ worst asymptotic complexity. For example, at 100K paths, OpenTimer takes 218463 ms while leftist and skew heaps take only 6275 ms and 5476 ms, respectively. The difference between the suffix-forest algorithm and our algorithm becomes apparent when the path amount becomes large. For instance, at 100K paths, our algorithms are 1.25× (leftist heap) and 1.11× (skew heap) faster; at 1M paths, our algorithms are 1.67× (leftist heap) and 1.8× (skew heap) faster. These discussions demonstrate that our algorithm reduces the time complexity significantly and does not introduce excessive constants. This is achieved by the optimizations in Section III-F.

Figure 5 compares the performance in two large circuit designs, `leon2` and `leon3mp`. The `leon2` have about 4.3 million pins and the `leon3mp` have about 3.6 million pins. As the number of paths increases, the gap among OpenTimer, suffix-forest, and our persistable heap becomes apparent gradually. At a path count of 1M, our algorithm is 60× faster than OpenTimer and 1.8× faster than suffix-forest. These speed-ups demonstrate the efficiency of our algorithm, which is able to process an extremely large number of paths at only quasilinear runtime with respect to the input size.

V. Conclusion

In this work, we propose a new path searching algorithm that has asymptotically better time complexity in path-based STA. Unlike other algorithms, which enumerate a large number of successor nodes in the prefix tree, we organize the deviations of successors in the form of a heap, and only need

TABLE I: Performance comparison between skew heap, leftist heap, and no heap. The runtime values are in milliseconds.

Absolute runtime														
num_paths	endpoints	pin_count	100				100K				1M			
method			OpenTimer	leftist	skew	forest	OpenTimer	leftist	skew	forest	OpenTimer	leftist	skew	forest
wb_dma	2439	25318	28	4	4	**4**	597	157	**109**	116	582	143	**117**	122
aes_core	3748	193584	113	77	80	**71**	11361	884	**869**	935	52613	**9658**	9857	10430
b19	17684	1898669	609	212	219	**169**	86780	**3247**	3893	3757	799597	**22313**	25116	26515
des_perf	25217	744603	281	86	83	**68**	15057	**1427**	1519	1523	52858	12551	**12520**	13647
vga_lcd	66965	717311	537	68	66	**54**	12326	**1378**	1386	1735	44614	11026	**10738**	14276
netcard	163821	4862680	2218	601	573	**451**	91084	**6140**	6708	6914	291826	**19269**	20404	26739
leon3mp	178591	3688590	2370	703	689	**590**	151778	6025	**5746**	6704	473318	22855	**22818**	36992
leon2	232655	4332962	4469	1607	1481	**1441**	218463	6275	**5476**	6834	1009337	20792	**19472**	34572
Runtime ratio with suffix forest=1														
num_paths	endpoints	pin_count	100				100K				1M			
method			OpenTimer	leftist	skew	forest	OpenTimer	leftist	skew	forest	OpenTimer	leftist	skew	forest
wb_dma	2439	25318	7.237	1.079	1.184	**1.000**	5.121	1.349	**0.938**	1.000	4.792	1.178	**0.965**	1.000
aes_core	3748	193584	1.585	1.078	1.122	**1.000**	12.146	0.945	**0.929**	1.000	5.044	**0.926**	0.945	1.000
b19	17684	1898669	3.600	1.249	1.294	**1.000**	23.098	**0.864**	1.036	1.000	30.157	**0.842**	0.947	1.000
des_perf	25217	744603	4.119	1.252	1.212	**1.000**	9.888	**0.937**	0.998	1.000	3.873	0.920	**0.917**	1.000
vga_lcd	66965	717311	10.041	1.264	1.234	**1.000**	7.103	**0.794**	0.799	1.000	3.125	0.772	**0.752**	1.000
netcard	163821	4862680	4.922	1.333	1.270	**1.000**	13.173	**0.888**	0.970	1.000	10.914	**0.721**	0.763	1.000
leon3mp	178591	3688590	4.020	1.193	1.168	**1.000**	22.638	0.899	**0.857**	1.000	12.795	0.618	**0.617**	1.000
leon2	232655	4332962	3.102	1.116	1.028	**1.000**	31.966	0.918	**0.801**	1.000	29.195	0.601	**0.563**	1.000

OpenTimer: the OpenTimer's original implementation[5], where no persistable heap and no suffix forest are used.
skew/leftist: our algorithm implementation using skew/leftist heap. **forest**: the CPU implementation of suffix forest algorithm [9].

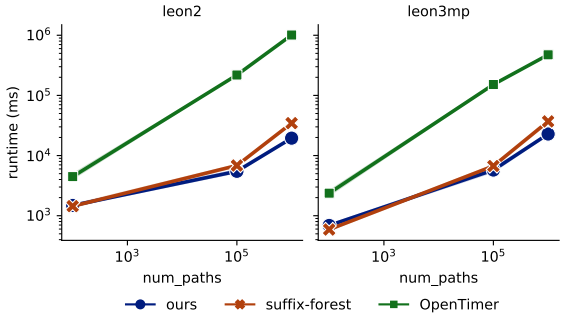


Fig. 5: Runtime scalability with increasing path counts based on the TAU contest benchmarks [2]. As the number of paths increases, the gap between OpenTimer, suffix-forest and our persistable heap algorithm becomes significant.

to enumerate a constant number of nodes at a time. At the same time, we take the inheritance relationship of deviation on the suffix tree and use persistable heaps to preprocess deviation. We prove our algorithm has a runtime complexity of $O(n \log n + k \log k)$ which is asymptotically better than the state-of-the-art OpenTimer [5] and the suffix-forest algorithm [9]. Our future work shall focus on the parallelization of our algorithm using techniques such as skip list [12] and task parallelism [13] to incorporate efficient heterogeneous parallelism. We also plan to combine the proposed path searching algorithm with CPPR [14], [15] and demonstrate its practicality in critical paths-driven EDA tasks like [16].

ACKNOWLEDGE

This work was supported in part by the National Science Foundation of China (Projects No. 62004006 and No. 62034007), the National Science Foundation of USA (CCF-2126672), and NumFOCUS Small Development Grant.

REFERENCES

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

[2] J. Hu, D. Sinha, and I. Keller, "TAU 2014 contest on removing common path pessimism during timing analysis," in *Proc. ISPD*, 2014, pp. 153–160.

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

[4] T.-W. Huang and M. D. Wong, "UI-timer 1.0: An ultrafast path-based timing analysis algorithm for CPPR," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, 2016.

[5] 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.

[6] P.-Y. Lee, I. H.-R. Jiang, C.-R. Li, W.-L. Chiu, and Y.-M. Yang, "iTimerC 2.0: Fast incremental timing and cppr analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 890–894.

[7] B. Jin, G. Luo, and W. Zhang, "A fast and accurate approach for common path pessimism removal in static timing analysis," in *Proc. ISCAS*. IEEE, 2016, pp. 2623–2626.

[8] P.-Y. Lee, I. H.-R. Jiang, and T.-C. Chen, "Fastpass: fast timing path search for generalized timing exception handling," in *Proc. ASPDAC*. IEEE, 2018, pp. 172–177.

[9] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *Proc. DAC*. ACM, 2021.

[10] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," in *SODA 2021*. SIAM, 2021, pp. 522–539.

[11] S. Cho and S. Sahni, "Weight-biased leftist trees and modified skip lists," *Journal of Experimental Algorithmics (JEA)*, vol. 3, pp. 2–es, 1998.

[12] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731504002333

[13] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 6, pp. 1303–1320, 2022.

[14] 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*. ACM/IEEE, 2021.

[15] ——, "Heterocppr: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism," in *Proc. ICCAD*. ACM/IEEE, 2021.

[16] 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 Transactions on Components, Packaging and Manufacturing Technology*, vol. 9, no. 9, pp. 1669–1679, 2019.