Architecture For Ordered Write Of Data Collected In Parallel For Path Based Report

Prabhat Maurya
prabhatmaurya@in.ibm.com
EDA, IBM Systems, Bengaluru, India

Anshul
anshul11@in.ibm.com
EDA, IBM Systems, Bengaluru, India

Shesha Raghunathan
shesha.raghunathan@in.ibm.com
EDA, IBM Systems, Bengaluru, India

***Abstract*—Static Timing Analysis (STA) using path based analysis of timing graph is very crucial in achieving accurate timing sign-off of chip designs. By nature, such analysis is compute and run time intensive and is a major bottleneck in speeding up the timing sign-off for large circuit designs. With many improvements made to the path tracing techniques, data collection and file writing are the expensive parts now. We propose a novel architecture in this paper that helps multi-thread this portion while maintaining slack ordering. Proposed architecture also achieves a pipeline effect and avoids any resource locking, which reduces the idle time for the threads. Experimental results demonstrate a speed up of nearly 5X end-to-end and over 6X for the steps of interest due to our implementation.**

***Index Terms*—Path based Static Timing Analysis, multi-threading, ordered write, pipeline architecture**

I. Introduction

Static Timing Analysis (STA) [1] is one of the primary tools used in VLSI chip design sign-off flow. STA enables timing verification that ensures the chips being taped out meet the specifications. There are two popular algorithms to implement STA: the first one is block based STA which uses breadth first traversal approach and the second type is path based STA, which traverses in depth first manner. While block based approach has been proved to be more efficient in design coverage for timing, tracing and identifying critical paths in the design is still a key requirement for STA tools. STA tools are used at multiple steps in the entire chip design cycle and are required to generate failing paths report which are consumed by both automated optimization tools as well as for manual fix up of critical paths. Hence, the path based report generation needs to be very efficient and accurate to keep the turn around time shorter.

In the past 20 years, there has been great interest to improve both the quality of sign-off as well as its performance. Many key conceptual developments in STA have helped improve the quality and performance of path based reports. Concepts like statistical STA [2], common path pessimism removal [3], effective current source modelling [4] and hierarchical timing [5] have improved the quality of timing analysis; they have also made the timing data collection more complex and runtime expensive.

With the advancements in technology nodes and increase in chip sizes, the amount of analysis data generated by STA tools is growing. A failing paths report for a large circuit may contain millions of paths. [7] describes method to perform path tracing in parallel and [8] presents techniques to reduce the number of paths traced to find the critical paths. [9] explains a technique to estimate critical paths in a design instead of tracing multiple paths to get the critical path. Approach like incremental timing in [10] reduces the computation needed for failing paths report. [11] and [12] describe efficient algorithms to find k-critical timing paths in a design. While there is a lot of focus to make the path tracing techniques efficient and to reduce the amount of work required to get to the critical paths [6], the reporting of these paths in a human consumable form however did not get commensurate attention. With efficient path tracing and estimation techniques, the data collection and file writing steps have become the most dominating steps in the entire failing paths report generation.

One additional constraint that applies to failing paths report is to identify the critical paths and sort the generated report in the order of slack criticality (increasing slack order). To achieve the correct slack ordering across paths in the report, all the paths must be traced before writing any path to a file (refer Sec. II-A). This constraint prevents the tool from applying the commonly used parallelism techniques to improve the performance of failing paths report.

In this paper, we present a novel architecture to speed up the data collection and report writing steps involved in failing paths report. It writes the data in ordered manner from a pipeline which is created for the data collected in parallel. The framework avoids any locking for the thread synchronization and enables efficient memory management. The framework can be tuned for better performance based on the computation requirements of the data collection and file writing portions of the application. Our approach shows substantial improvement in run-time for large reports containing millions of paths. These features add up to faster chip design cycle without compromising on the accuracy of results.

The rest of the paper is organized in multiple sections. Section II describes the steps involved in generating a failing paths report and runtime analysis to identify the dominating steps in the process. Section III presents the overall solution architecture with sub-sections focusing on individual components of the architecture. Section IV talks about some fine tuning techniques to get optimum performance from the architecture. Finally section V explains the experimental results and we conclude and discuss next steps in section VI.

II. Background

A. Steps in path based report

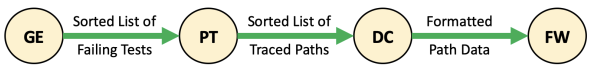


Fig. 1: Steps in path based report generation

A failing paths report generation, at an abstract level, involves multiple steps as shown in the order of execution in Fig 1. These steps are explained below in detail:

1. Gathering Endpoints (GE): Gathering all failing tests, which we call endpoints, in a design based on the slack cutoff and sort them by slack. This step is also responsible for applying various endpoint level filters such as slack threshold, edges, clock type, types of checks and many more. In order to sort the endpoints by slack, GE step also performs computation of sub-critical slack for endpoints if the check causing the worst slack is filtered out.
2. Path Tracing (PT): This step is responsible for tracing requested number of paths, collect all the traced paths and sort them based on the slack. It supports various flavors of path tracing such as k-worst paths across design, k-worst paths per endpoint for n-worst endpoints in the design, k-worst paths per physical pin in the design, etc. The time consumed in the path tracing step depends on the flavor requested. The path traversal is performed over the in-memory timing graph and the path is stored as a linked list of nodes. This makes the path tracing step memory and runtime efficient.
3. Data Collection (DC): Collecting timing data for each node and delay data for each edge in a path and formatting the data in requested format. This step also involves collecting additional path level information. Data collection step is significantly slower as it involves collecting timing data for all the nodes and edges in the path followed by multiple string formatting operations on collected data.
4. File Write (FW): Writing paths to a file in slack sorted order.

Algorithm 1 shows a pseudo-code for the failing paths report generation. GE step returns a list of failing tests T sorted based on slack (lines 1-4). This sorted list of failing tests is fed as input to PT step which is responsible for tracing the requested number of paths for every test in the list (lines 5-7). The output of PT is a list of traced paths P which are sorted based on the slack values. DC step then collects the timing data for every node of each path and formats it in a user readable format (lines 9-12). The time taken in DC step is represented by D. Finally in FW step (line 13), the formatted path data is written to file in a slack sorted order. The time taken for FW step is represented by F. The slack ordering constraint puts a limitation on running various steps in parallel.

Algorithm 1: Generate failing paths report

```
input  : timing graph $G$; number of paths to report per test $p$; slack cutoff $s$
output: Failing paths report containing $p$ paths per test
1  foreach node $n$ in the timing graph $G$ do
2      if $n.hasTest()$ and $n.slack < s$ then
3          Push $n$ to list of failing tests $T$
4  sort $T$ in increasing slack order
5  foreach test $t$ in $T$ do
6      $P_t \leftarrow tracePaths(t, p, G)$
7      Push $P_t$ to list of paths $P$
8  sort $P$ in increasing path slack order
9  foreach path $pth$ in $P_t$ do
10     foreach element $e$ in $pth$ do
11         $e\_timing \leftarrow CollectTimingData(e)$
12         $e\_line \leftarrow FormatLine(e\_timing)$
13         $WriteToFile(e\_line)$
```

B. Run-time break-down of path based report

In order to identify the steps that need speedup, we collected the run-time break-down of the failing paths report across three designs of different sizes as shown in Table I. All these reports were generated for one million endpoints tracing the most critical path per endpoint which is the recipe used in the production timing runs. The bar chart in Fig. 2 clearly shows that DC and FW steps are the major contributors taking 69% to 83% of total report generation time. The time taken by these steps depends on the number of paths being requested, the length of paths and the flavor of path tracing as mentioned in section II-A. DC step is significantly slower than PT step due to various data collection and formatting operations performed in DC step as opposed to PT step which primary deals with in-memory timing graph for traversal and stores the results as pointers to linked lists.

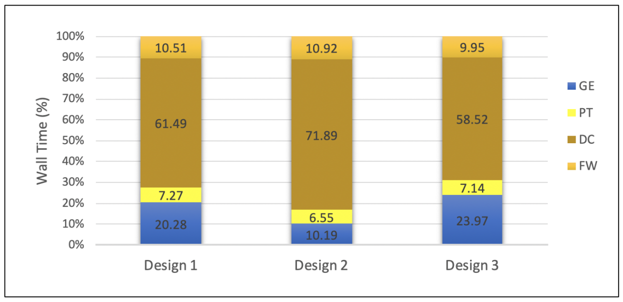


Fig. 2: Run-time break-down

III. Proposed Architecture

Architecture in Fig. 3 shows the proposed solution to improve performance of DC and FW steps described in section II-A. The flow starts with GE step which finds all the failing tests in the design, sorts them by slack and passes

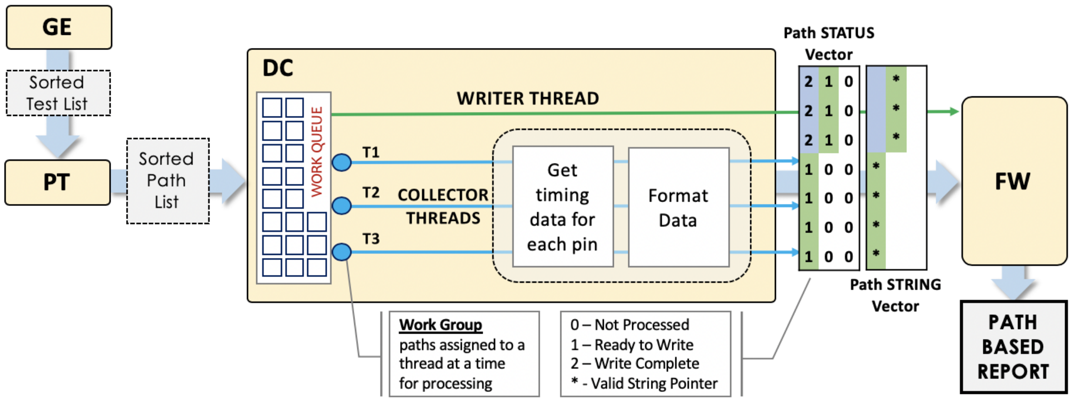


Fig. 3: Proposed Architecture

on the list to PT step for tracing paths. PT step generates a list of paths sorted by slack which is then fed to the DC step. In DC step, all the traced paths are put in a global work queue which is responsible for feeding paths to the threads for processing. n child threads are created and one of the thread is designated as a writer thread; remaining $n-1$ child threads are designated as data collector threads which perform the data collection and formatting for the paths in the list. The data collector threads synchronize with the writer threads using two vectors described in Sec. III-C. Finally the writer thread generates the path based report by writing the formatted path details from the path string vector.

A. Data Collector Threads

Data collector threads, as shown in Fig. 3, are responsible for processing the paths assigned to them. Processing a path includes collecting timing data for each node in the path, delay information for each edge along the path, information related to the timing check for the path and format it in a human readable format. The formatted timing data is then stored in a specific location for the writer thread to pick and write to file. Each collector thread is assigned a fixed size chunk of paths (aka workgroup) to process from the work queue. We will discuss more about the workgroup size in Sec. IV-A. After processing all the paths in a workgroup, the child thread is assigned a new workgroup till all the paths in the work queue are processed.

B. Writer Thread

One of the child threads is designated as Writer thread which is responsible for writing all the paths processed by collector threads to a file. Writer thread is responsible for maintaining the slack order of paths while writing them to file. The paths are picked in slack order and written to file in a serial manner. Even though the file writing is serial to ensure path ordering, it is performed in parallel with data collector threads working on subsequent paths. Writer thread keeps looking at a designated location for the availability of processed paths from data collector threads. Collector and writer threads synchronization is discussed in more detail in the next sub-section.

C. Collector-Writer Threads Synchronization

Synchronization between collector and writer threads is required to maintain the slack based ordering among paths in the final path based report. The system uses two vectors for the synchronization of collector and writer threads and avoids any kind of locking to achieve this synchronization. Both the vectors are of size P (total number of paths in report).

- Path Status Vector: represents current status of a path (index of an entry in vector corresponds to path number in global list)
  - 0 represents that the path is not yet processed by any data collector thread
  - 1 represents that the processing of path is complete by data collector thread and it can be picked up by writer thread
  - 2 indicates that the path is written to file successfully by the writer thread
- Path String Vector: contains the pointers to the processed paths which are available for writer thread

Data collector threads work on the set of paths assigned to them and update the vectors as shown in Fig. 3. Once a path is processed, corresponding entry in the Path String Vector is updated by the collector thread. Then collector thread updates the Path Status Vector replacing 0 by 1. Collector thread gets back to processing next path in the workgroup. Writer thread is responsible for maintaining the slack order of paths while writing to file. It starts from first entry in the vector and keeps moving in serial manner by writing a path as soon as it is available. Writer thread keeps polling for status 1 for the next path in the queue. As soon as bit 1 is available, it reads the path from Path string vector, writes it to file, releases memory and then updates the status of the path as 2.

This vector based synchronization avoids any locking of resources. Collector threads can update the path status vector without any conflicts with other collector threads as each thread works on separate sets of paths. Writer thread updates the entries for a path in these vectors only after processing a path and hence has no conflict with collector threads. This architecture also keeps the memory usage contained. Path status vector uses two bits per path to store the status of the paths. Path string vector stores the formatted data for each path and may add to significant memory usage for large reports. Writer thread releases the memory used to store the path data after writing the path to file. This ensures that memory is used only for unprocessed paths in the path string vector.

IV. Discussion

A. Load Balancing

The number of paths assigned to each collector thread is called as a workgroup. The workgroup size can be varied for load balancing between the data collector threads. It depends on the total number of paths to be processed and the time consumed to process individual paths, which in turn depends on the path length. Too small a workgroup can lead to congestion at the global queue while too large a work group can result in under utilization of collector threads. The optimal value for the workgroup size can be fine tuned by running experiments with different workgroup sizes and analyzing the impact on the wait times of collector threads. For the experimental results in this paper, we are using a workgroup size of 1000, which shows the optimum balancing of collector threads for the set of designs used.

B. Pipeline Architecture

The architecture presented in this paper ensures reduced wait time for collector as well as writer threads and hence result in optimal performance with available resources. Fig. 4 shows an example of pipeline effect achieved by this architecture with one writer and three collector threads. In this example, each collector thread is assigned a workgroup of two paths to process at a time.

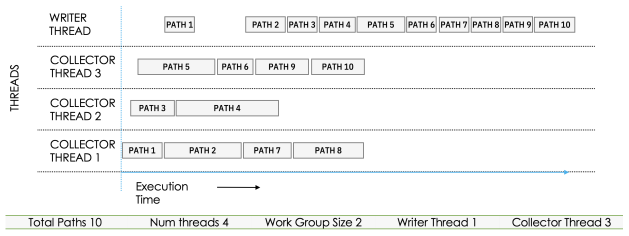


Fig. 4: Pipeline architecture: example

Writer thread needs to wait for a very small time at the beginning when paths are yet to be processed. To avoid this, the writer thread can be assigned the first set of paths to collect data and then write to file. Once the data collector threads start processing paths, they will soon overrun the single writer thread's file writing speed. Towards the end when all the paths are processed by the collector threads, writer thread may take some additional time to write all the paths resulting in some idle time for the collector threads. As the number of paths being processed increase, these wait times become negligible, ensuring optimal usage of all the child threads.

C. Inflection Point

As mentioned in section II-A, D and F are the times taken for data collection and file write steps for a paths report generated in serial manner. The ratio of D and F provides a good metric to understand the performance improvement that can be achieved using the proposed architecture and can be used to find the optimum number of threads. As the DC step is more expensive as compared to FW step, the ratio is much greater than 1 for report generated in serial fashion. As we apply more threads for the DC step, this ratio starts decreasing and approaches 1.

$$D/F = 1 \tag{1}$$

The point where time taken by file write step becomes equal to the time taken by data collection step, we define it as inflection point as shown in Eq. (1). With just one writer thread, increasing number of thread count after inflection point, does not help in improving the performance.

We can further compute the number of threads for optimum performance for a given design using Eq. (1). Let's define the number of paths as P and average path length (computed across all paths) as l. We can compute average data gather time per node d and file write time per node f using Eqs. (2).

$$\begin{aligned} d &= D/(P * l) \\ f &= F/(P * l) \end{aligned} \tag{2}$$

If n is the number of threads required to get optimum performance using the architecture described in this paper, $(n-1)$ threads will be used as data collector threads and one thread as writer thread. For optimum performance, data collection and file write times in multi-threaded mode should be equal.

$$(d * P * l)/(n-1) = (f * P * l) \tag{3}$$

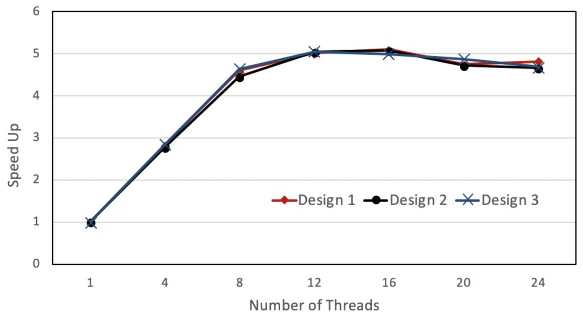
$$n = (d/f) + 1 \tag{4}$$

Eq. (4) indicates that the inflection point and in turn the number of threads for optimum performance is independent of average path length in a design.

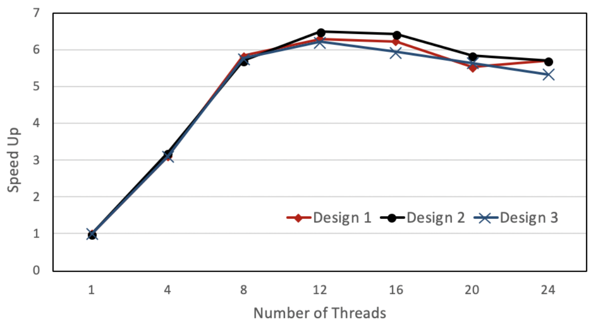
V. Experimental Results

All the experiments mentioned in this paper were performed on a Intel Xeon E5-2667 v2 3.30 GHz machine with 32 CPUs, Linux 2.6.32-696.28.1 64 bits operating system and HDD disks. The proposed solution is implemented in C++ as part of actual EDA timing analysis tool. The three circuit instances used for the experiments are sequential circuits with different functions and sizes. The number of nodes in timing graph is used to represent the size of timing graph as shown in Table I. Here, number of nodes is the count of pins and connections represent the count of edges joining any two nodes in the design.

Number of paths P written in each path report was kept constant at one million and the average path length was



(a) Paths report end-to-end



(b) Data gather and file write steps

Fig. 5: Performance improvement with respect to different threads

TABLE I: Design details

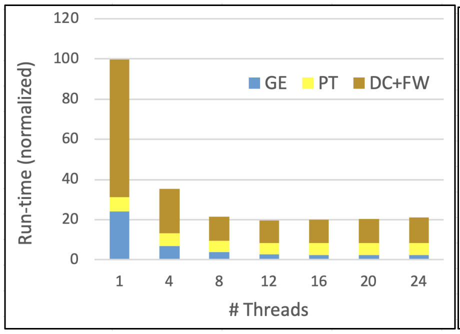
	Number of nodes	Connections	Avg path length
Design 1	710M	820M	42
Design 2	1234M	1270M	40
Design 3	1265M	1374M	38

calculated across these one million paths for every design. Experiments involved generating the path based reports in serial manner followed by multi-threaded report generation with 4, 8, 12, 16, 20 and 24 threads.

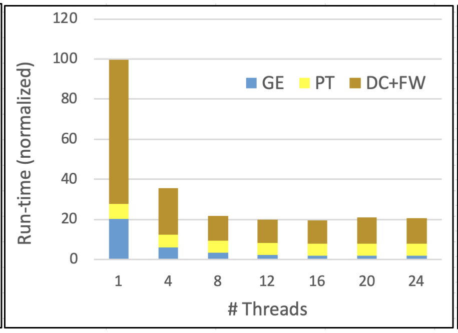
In Fig. 5 we plot the performance improvement achieved by our solution over the serial paths report generation. Performance improvement for end-to-end report generation time for different designs is shown in Fig. 5a and performance improvement for the multi-threaded portion (DC and FW steps) of path report is shown in Fig. 5b. X-axis represents number of threads used to generate the report and Y-axis represents the scaling of the performance with respect to serial mode. Paths report performance scales with increasing number of threads and shows an improvement of nearly 5X end-to-end. The improvement in overall report generation time is limited by the improvement in the portions of interest for our architecture which includes DC and FW steps. The improvement seen for the multi-threaded portion of the report is over 6X as compared to serial report. Significant performance improvement is seen by increasing the number of threads upto 12 threads as shown in Fig. 5a and Fig. 5b but starts to flatten beyond that.

Fig. 6 shows the break-down of normalized (w.r.t. serial) run-time for all 3 designs. As described in Sec. II-A, the figures show break-down in terms of 4 major steps. Since the proposed architecture lends itself to pipelining of file write, and thus the data gathering and file write steps can happen in parallel, we can only get combined run-time of $DC + FW$. We see that the $DC + FW$is the long pole in all 3 designs and it reduces quite rapidly before flattening out from about 12 threads onwards. Note that the GE step also successively reduces as we have multi-threaded that portion in the implementation. Since the % contribution of GE in the serial run is small relative to $DC + FW$ to start with, the benefit accrued from this multi-threading is comparatively less interesting. We talked about the significance of the ratio of data collection time and file write time in Sec. IV-C. With just one writer thread, increasing number of threads after inflection point, does not help in improving the performance.

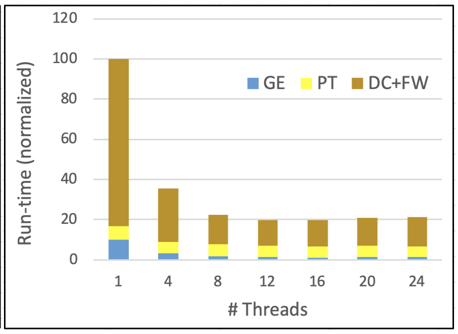
The plot in Fig. 7 shows the ratio of data collection D and file write F times for different designs used for the experiments for different number of threads. The plot clearly shows a decline in the ratio as number of threads is increased. A horizontal line intersects the plots at the inflection points



(a) Design 1



(b) Design 2



(c) Design 3

Fig. 6: Run-time break-down

for different designs and it lies between 8 to 12 threads for all the designs. Using Eq. (4), inflection points are calculated as 7, 8 and 7 respectively, for $Design$ 1, $Design$ 2 and $Design$ 3. There is slight variation between empirical value and theoretical value calculated using inflection point for optimum number of threads possibly due to certain noises involved here like disk write time variation, etc.

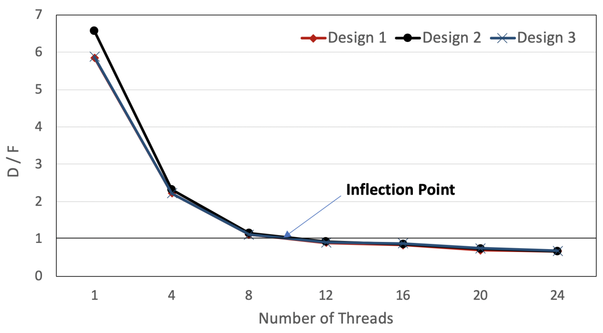


Fig. 7: Inflection Point

VI. Conclusion

In this paper, a novel architecture to improve the performance of data collection and file write steps of a path based report was introduced and the proposed solution is implemented in the actual timing analysis tool. This paper demonstrates a significant improvement of nearly 5X end-to-end and over 6X for DC and FW steps. The proposed architecture can implement path based report more efficiently without compromising on the quality of results and can save a lot of time in the timing sign-off flows. The architecture ensures there is no locking of resources and also keeps the memory usage contained.

References

[1] L.-T. Wang, C. E. Stroud, and N. A. Touba, “System on Chip Test Architectures”, Morgan Kaufmann, 2008.

[2] C. Visweswariah, K. Ravindran, K. Kalafala; S. G. Walker, S. Narayan, D. K. Beece, J. Piaget, N. Venkateswaran, and J. G. Hemmett, “First-Order Incremental Block-Based Statistical Timing Analysis”, IEEE trans. Computer-Aided Design of Integrated Circuits and Systems, vol.25, no.10, pp.2170-2180, Oct. 2006.

[3] JBaihong Jin, Guojie Luo, Wentai Zhang, “A fast and accurate approach for common path pessimism removal in static timing analysis”, IEEE International Symposium on Circuits and Systems (ISCAS), 2016.

[4] A. Korshak, Jyh-Chwen Lee, “An effective current source cell model for VDSM delay calculation”, IEEE 2nd International Symposium on Quality Electronic Design, 2001.

[5] Iris Hui-Ru Jiang, Pei-Yu Lee, “Timing Macro Modeling for Efficient Hierarchical Timing Analysis”, IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2018.

[6] Sourabh Kumar Verma, Naresh Kumar, Ajay Tomar, Rakesh Agarwal, Umesh Gupta, Manish Bansal, Kaustav Guha, Prashant Sethial, “Comprehensive path based analysis process”, US Patent, US9875333B1, 2016.

[7] Ajay K., Ravi, “Determination of most critical timing paths in digital circuits”, US Patent, US8028260B1, 2008.

[8] Vasant Rao, Debjit Sinha, Nitin Srimal, Prabhat K. Maurya, “Statistical path tracing in timing graphs”, 53rd ACM/EDAC/IEEE Design Automation Conference (DAC), 2016.

[9] Cristian Soviani, Rachid N. Helaihe, lKhalid Rahmat, “Efficient exhaustive path-based static timing analysis using a fast estimation technique” UD Patent, US8079004B2, 2009.

[10] Chaitanya Peddawad, Aman Goel, Dheeraj B, Nitin Chandrachoodan, “iitRACE: A memory efficient engine for fast incremental timing analysis and clock pessimism removal”, IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2015.

[11] Kuan-Ming Lai, Tsung-Wei Huang and Tsung-Yi Ho, “A General Cache Framework for Efficient Generation of Timing Critical Paths”, IEEE/ACM Design Automation Conference (DAC), 2019.

[12] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin and Martin Wong, “An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints”, Design Automation Conference (DAC), 2020.