Fast and Practical False-Path Elimination Method for Large SoC Designs

Chul Rim, Soo-Hyun Kim, Joo-Hyun Park, Myung-Soo Jang,
Jin-Yong Lee, Kyu-Myong Choi, and Jeong-Taek Kong
CAE center, System LSI business, Device Solution Network, Samsung Electronics Company.

Abstract—**In this paper, we propose a new fast and practical technique to eliminate known false paths during the static timing analysis (STA). False paths are verified fast using additional information stored in arrival times, which is a pass-through history of exceptional nodes. The information can be constructed with small memory overhead because individual false path list is not managed in each arrival time. We adapted this method to classical arrival time computation and critical path searching algorithm. The feature is used in CubicTime, our full-chip gate level static timing analyzer supporting multiple clock domains. We describe the details of our algorithm and the experimental results compared to those of our previous method and a de-facto industry-standard STA tool.**

I. Introduction

In SoC design era, as design size is increasing rapidly, better performance and larger capacity are needed to verify timing operation of circuits. Conventional dynamic simulation methods have become inadequate for verifying timing of large full chip design because run time is nearly infeasible and efforts for generating input stimulus is also extremely high. To overcome these capacity limits, static timing analysis (STA) has been proposed for efficient full-chip timing verification [1, 2]. STA is an exhaustive method of verifying timing performance of synchronous design. It is very efficient not only for its fast runtime but also for its large coverage [1,2]. Although the STA does not need input stimulus, it guarantees worst-case timing operation by covering both rising and falling transition at every primary input node.

Because STA does not take logic into account, it may report logically or temporally unrealizable paths. These paths are often referred to as false paths. Although STA is very efficient way to verify the timing of circuits, it suffers from handling exceptions such as false paths. False paths should be detected automatically or removed manually in order not to overestimate critical paths. The method of eliminating user-specified false path is widely used in industry because finding true critical path automatically in large circuit takes too much computation time even if heuristics are used [6].

False paths can be specified by a set of false sub-graphs or through-path exception formats [6, 9]. The former can specify more false paths in a single representation than the latter, but it is difficult to describe false paths in graph format. In this reason, the through-path exception format is widely used in industry.

Many researches have been done for elimination or detection of false paths [4–12]. In this paper, we formulate algorithm for finding true critical path given a set of known false paths described in through-path-exceptions format.

Our approach is effective for some reasons. Unlike previous methods [4–12], it does not need additional information for individual false paths in each node. Instead, each node has minimal points pass-through history information. Moreover, the proposed method can easily be adapted to classical arrival time computation and path search procedures.

Our method is implemented in CubicTime, which is the gate level in-house STA tool based on CubicWare environment [3].

The outline of the paper is as follows. In section 2, we describe backgrounds for the static timing analysis. In section 3, we formulate the algorithm for finding true critical paths in detail. In section 4, we describe the experimental results applied to several industrial designs. And finally, we give the conclusions.

II. Preliminaries

In this section, we describe some basic terms and definitions for our data structure. And then, we describe classical STA algorithm that is adapted and modified in our approach. We assume that the design is acyclic.

A. Design Formulation for Static Timing Analysis

The design has a component database for the timing analysis. Every pin or node has rising and falling *delay edge*. Each delay edge has both minimum and maximum delay events. A *delay event* means rising or falling signal transition at a certain time. *Root events* are initial transitions at clock sources or primary input nodes. A *delay arc* is a rising or falling pin-to-pin delay component [1]. A delay arc can be either of cell arc or net arc.

Every root event is propagated through delay arcs and each node may have a number of delay events from different root events[1]. Each delay event has link to the event that causes it, then the link is used in backward path search phase.

B. Arrival Time Computation

The procedure for arrival time computation is propagating all root events from primary inputs or clock sources to path endpoints [1,2,3]. Minimum and maximum arrival times (or delay events) at each node are worst-case propagation delays from its root event to the node. Arrival times are computed in breadth-first-search (BFS) manner after logical level of every node is decided. The arrival time of a node is calculated by sum of worst arrival time of its fan-in nodes and incoming cell or net arc delay. To support multiple clock domains, arrival times are stored separately grouped by their root events. Figure 1 shows pseudo algorithm for maximum arrival time computation. Minimum arrival time can be computed in similar way. Note that only the modified arrival time is propagated to next level and the procedure repeats until no more arrival time modification occurs.

```
// calculate design arrival time (max)
repeat {
  for each node of logic level $i$ {
    for each output pin $p$ in level $i$ nodes {
      $Q$ = list of cell input pins
      for each pin $q$ in $Q$ {
        // take arc unativity into account
        get $q \rightarrow p$ arc delay
        arriv($p$) = $MAX$(arriv($q$) +delay($q \rightarrow p$) , arriv($p$))
      }
    }
  }
} until no node arrival is changed
```

Figure 1: Algorithm for arrival time computation

C. *Logical False Path*

Logical false path is logically unrealizable path [5,11,13]. Because STA doesn't take logic into account, it searches critical path only referred to delays. Figure 2 shows simple example of logical false path. If logic is not taken into account, path A→C→E→F is selected as a longest path. But this path is unrealizable for select signals to two multiplexers are opposite. Actually, the paths A→B→E→F and A→C→D→F are realizable longest path. So, the path A→C→E→F must be specified as a false path and should be eliminated during timing analysis phase not to overestimate longest path.

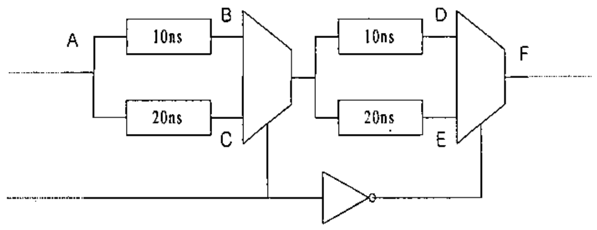


Figure 2: An example of typical logical false path

III. False Path Elimination

The key idea of the approach is storing pass-through history of the user-specified exceptional points in each delay event and making use of them later for path verification. Note that each false path is specified in a partially ordered list of nodes the path is propagating through, and false path verification occurs at every constrained path endpoint. By preceding operation, every *false path through-point* specified by user is marked as special point, and every endpoint has set of all possible false paths to the endpoint. These operations do not need any complex process and can be done with very small overhead.

A. *Building Exception-through History*

Each delay event has exceptional-point-through history if the delay event is propagated through at least one exceptional point. Computing the history information of each event can be embedded in arrival time computation process. We call the history as "false head" as in [6] and denote it by *FH*. The structure of the false head is simply an array or a list of nodes.

The function *InheritFalseHead* described in figure 3 shows overall false head inheritance mechanism. The function is called whenever the delay event D is modified for propagating the delay event S. Each delay event may store its own false head or just point false head of the event that causes it. Because the portion of exceptional nodes to entire nodes is very small, false head does not change frequently while propagating through nodes. Therefore, in most case, only pointing the false head of its cause event is enough rather than storing its own false head.

```
// inherit false head from cause event S to caused event D
InheritFalseHead (S, D) {
  if ($NODE_D$ is primary input & exceptional) {
    $FH_D$ = $NODE_D$        // initial node is exceptional
  } else {
    if ($FH_S \neq \varnothing$) {
      if ($NODE_D$ is exceptional) {
        $FH_D$ = $FH_S \bullet \{NODE_D\}$   // grow new FH
      } else {
        $FH_D$ = $FH_S$              // directly point
      }
    } else if ($NODE_D$ is exceptional) {
      $FH_D$ = $\{NODE_D\}$             // new FH issued
    } else {
      $FH_D$ = $\varnothing$        // remove previous inheritance
    }
  }
}
```

Figure 3:. Building false head of D node

B. *Finding True Critical Path*

Once arrival time and false heads are computed at every delay event, critical path to every path endpoint should be found to analyze design timing. Classical path searching algorithm is basically a depth-first backward searching process. The worst-case delay event on a path endpoint is traced backwards to its root event to build a full path [1, 2].

If a delay event propagated to an endpoint has no false head, the delay event is true, so true critical path can easily be found by tracing the cause-event backwards from an endpoint delay event. Otherwise, if a delay event on the endpoint has false head and the endpoint has a possible false path list, the delay event needs to be verified.

```
// initially, $SLACK_{CUTOFF}$ = LOOSE, TD = 0, FT = $\emptyset$

BackwardSearchTruePath (E, FT, TP, TD) {
  if ($FH_E$ = $\emptyset$ and FT is not false path) {
    $SLACK_{CURRENT} = T_{EndReq} - arriv(E) - TD$
    if ($SLACK_{CURRENT} < SLACK_{CUTOFF}$) {
      // new critical path is found
      $SLACK_{CUTOFF} = SLACK_{CURRENT}$
      Build critical path with E and TP
    }
  } else if ($FH_E = FH_{PREV}$ and E is not exceptional) {
    search backwards without false path check (1)
  } else {
    if ($FH_E$•FT makes false path) {
      // searching branches backwards ..............(1)
      PUSH(FT, E), if E is exceptional
      PUSH(TP, E)
      for each branch E′ of E {
        $TD' = TD + delay(E' \rightarrow E)$
        BackwardSearchTruePath (E′, FT, TP, TD′)
      }
      POP(FT, E), if E is exceptional
      POP(TP, E)
    } else {
      // found true path
      $SLACK_{CURRENT} = T_{EndReq} - arriv(E) - TD$
      if ($SLACK_{CURRENT} < SLACK_{CUTOFF}$) {
        $SLACK_{CUTOFF} = SLACK_{CURRENT}$
        Build critical path with E and TP
      }
    }
  }
}
```

Figure 4:. Recursive backward critical path search algorithm

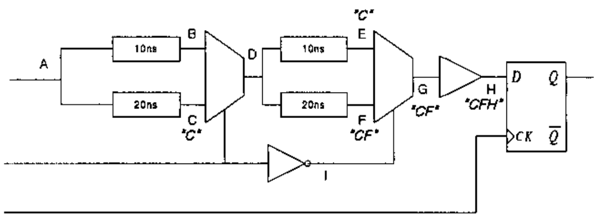
Another exception point through history "false tail", named as in [6], is generated during the backward path searching process. False tail, denoted by *FT*, is exceptional points pass-through history of current backward path search session. The false tail is managed dynamically by *push* and *pop* operations for the stack during the recursive search process as shown in figure 4. Note that only one false tail is managed for critical path searching.

Given *FH* and *FT*, the concatenation *FH•FT* makes complete exceptional points pass-through history of current searching path. If one of known false paths to the path endpoint is ordered subset of the sub-path *FH•FT*, current searching path is false.

To find a true critical path to an endpoint, every possible critical branch path must be checked. Since a delay event stored in a node has worst-case arrival time, whatever the event is true or not, there cannot be more critical branch path in the fan-in cone to the node. Thus, the fan-in branch whose slack is less than the minimum true slack found before can be pruned for the backward searching. Figure 4 shows the algorithm for finding true critical path. In the pseudo code, *E* denotes the delay event currently tracing, *TP* denotes tail path, and *TD* denotes the delay of the tail path. The algorithm runs recursively, tracing branches backwards.

While tracing backwards the nodes from N_{n+1} to N_n, if FH_n is identical to FH_{n+1} and the node N_n is not exceptional, false path matching is not needed because there is no difference in *FH•FT* between the nodes. Thus, the procedure may continue to the further fan-in branches without redundant false path matching.

Although the described algorithm runs very fast in most case, the worst-case complexity of the procedure is F^D — number of possible paths to the endpoint, where *F* is average fan-in count of node and *D* is logical depth to the endpoint. But the performance degradation hardly happens because many branches are pruned during the process. Moreover, *F* is typically small value around 2, and false path matching occurs only at a few nodes.



Bold : user specified exception-through point
Italic : false head of arrival event

Figure 5: False path verification example

Figure 5 shows an example flow of proposed algorithm. To simplify the problem, only maximum delay path is explained. It is assumed that there is single root event and all gate delays are same. In the figure, bold letters represent user-specified exceptional nodes and italic string represents the false head stored in the event. By preceding operation, there is user specified false path {C→F→H} on the D pin of the flip flop. The initial longest path is found to be {A→C→D→F→G→H} by referencing propagation delay only. But it is false path because user specified false path {C→F→H} is an ordered subset of the path, so true critical path should be found. The detailed flow of the proposed algorithm is described as follows.

1. On the final event on the endpoint, *FH* is {*C, F, H*} and *FT* is {}. The current searching path is false because the user-specified false path {*C, F, H*} is an ordered subset of the *FH•FT*.
2. The procedure continues to *G* and *F* respectively. On these two nodes, *FH* is {*C, F*} and *FT* is {*H*}. Because {*C, F, H*} is an ordered subset of the *FH•FT*, the current searching path is false.
3. On the nodes *D* and *C*, the *FH* is {*C*} and *FT* is {*F, H*}. The current searching path is false path because the {*C, F, H*} is an ordered subset of the *FH•FT*.
4. When tracing a branch node *B*, *FH* is {} and *FT* is {*F, H*}. This makes true path because {*C, F, H*} is not an ordered subset of the *FH•FT*. Hence, the event on B and the tail path make a full true path {*A, B, D, F, G, H*}.
5. The other branches *E* and *I* don't have worse slack. So further branch searching is not needed from them. Finally, the true critical path is proven to be {*A, B, D, F, G, H*}

IV. Experimental Result

The method described was applied to 7 industrial full-chip gate-level designs ranging from 12,000 to 300,000 instances. They consist of PDA, MODEM, and processor core chips. The experiments were performed on a dedicated SUN blade-1000 machine running in 32bit mode with enough physical memory. Table 1 shows the specifications of the sample designs. Every design script used is real signoff STA script and has multiple clock definitions and real false path descriptions.

Design	# clocks	# gates	# instances	# false paths
A	14	342K	55K	20761
B	2	1680K	80K	74
C	113	34K	12K	97
D	18	2041K	298K	2736
E	9	1579K	179K	488
F	1	1103K	128K	84362
G	233	1147K	185K	63105

Table 1: Sample design specification

To show the effectiveness of our approach, the proposed method was compared to our previous method and the industry de-facto standard STA tool in the aspect of performance and memory usage. For the experiments, timing analysis is done for both minimum and maximum paths in order to check setup, hold, recovery, and removal violations at every path endpoint.

Our previous false path verification method matches false path using path structure after building a full path without any look-ahead data such as false head and false tail.

The performance comparison result is shown in table 2. Elapsed time is used as the metric to compare with commercial tool. The runtime includes time for arrival time computation and all path verification as well as time for preprocessing operations such as loop breaking and constant propagation. The result shows that the proposed approach is up to about 35 times faster than the previous method, and also up to about 3.3 times faster than de-facto industry standard commercial STA tool. The average runtime is also about 16 times faster than the previous method and 1.9 times faster than commercial STA tool.

For memory usage comparison shown in table 3, the proposed method uses slightly more memory of about 5% than the previous method for the speed gain. This means that saving false head in every node can be done with only small memory overhead. The memory usage comparison to commercial STA tool does not tell regular tendency but seems to be comparable.

Design	Runtime (elapsed, sec)		
	with FH, FT	w/o FH,FT	Commercial
A	48	1208	59
B	130	2225	88
C	79	208	69
D	322	3480	623
E	112	1922	191
F	114	4001	379
G	255	3988	628

Table 2: Elapsed time for all constraints check

Design	Memory Usage (MByte)		
	with FH,FT	w/o FH,FT	Commercial
A	128	123	173
B	243	217	169
C	63	61	100
D	638	598	549
E	426	417	361
F	330	321	688
G	499	488	479

Table 3: Memory usage of each method

V. Conclusion

In this work, we have proposed a technique for eliminating known false paths during the static timing analysis and searching true critical path. The key idea of the approach is looking ahead of the pass-through history of each arrival time and verifying the path without constructing it. Our experiment for some real industry designs have demonstrated that proposed method is up to 3.3 times faster than commercial de-facto standard tool while memory usage is comparable. The method can be extended to handle another path exceptions such as multi-cycle paths.

References

[1] J. Cherry. "Pearl – A CMOS timing analyzer", *Proc. of the DAC*, 1988, pp. 148-153

[2] T. G. Szymanski. "LEADOUT: A Static Timing Analyzer for MOS Circuits", *Proc of ICCAD*, 1986, pp. 130-133

[3] Myung-Soo Jang, et. al. "CubicWare: a hierarchical design system for deep submicron ASIC", *Proc. of 12th ASIC/SOC Conference 1999*, pp.168-172

[4] Hoon Chang; Abraham, J.A. "CHAN, An efficient critical path analysis algorithm", *Design Automation, 1993, with the European Event in ASIC Design. Proceedings. [4th] European Conference, Feb 1993*, pp. 444 - 448

[5] Haizhou Chen; Bing Lu; Ding-Zhu Du. "Static timing analysis with false paths", *Proc .of International Conference on Computer Design , 2000*, pp. 541-544

[6] David Blaauw, et. al. "Removing user specified false paths from timing graphs", *Proc.of DAC 2000*,

[7] P. C. McGeer , R. K. Brayton, "Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network", *Proc. of DAC 1989*, pp. 561-567

[8] Tsukiyama, S, Tanaka, M.; Fukui, M. "Techniques to remove false paths in statistical static timing analysis", *ASIC, 2001. Proceedings. 4th International Conference on , 2001*, pp. 39-44

[9] Goldberg, E, Saldanha, A. "Timing analysis with implicitly specified false paths", *VLSI Design, 2000. Thirteenth International Conference on* 2000, pp. 518-522

[10] Belkhale, K.P, Suess, A.J. "Timing analysis with known false sub graphs", *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on, Nov 1995*, pp. 736 -739

[11] Bolender, E, Lipp, H.M. "Timing verification - a new understanding of false paths", *Design Automation, 1992. Proceedings. [3rd] European Conference on , Mar 1992*, pp. 383 -387

[12] Kukimoto, Y, Brayton, R.K. "Timing-safe false path removal for combinational modules", *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on , 1999*, pp. 544 -549

[13] D. H. C. Du, S. H. C. Yen, and S. Ghanta, "On the General False Path Problem in Timing Analysis", *Proc. of the 26th DAC* 1989, pp. 555-5607