Detecting False Timing Paths: Experiments on PowerPC$^{\text{TM}*}$ Microprocessors

Richard Raimi — Motorola Corp., 6200 Bridgepoint Pkwy., Austin, TX 78730

Jacob Abraham — Computer Engineering Research Center, The University of Texas at Austin, Austin, TX 78712

Abstract

We present a new algorithm for detecting both combinationally and sequentially false timing paths, one in which the constraints on a timing path are captured by justifying symbolic functions across latch boundaries.

We have implemented the algorithm and we present, here, the results of using it to detect false timing paths on a recent PowerPC microprocessor design. We believe these are the first published results showing the extent of the false path problem in industry. Our results suggest that the reporting of false paths may be compromising the effectiveness of static timing analysis.

1 Introduction

Designers of high performance chips, such as microprocessors, rely heavily on *static timing analysis*. The major flaw of static analyzers, leaving aside the difficulty of modeling the physics of devices, is the reporting of false paths, i.e., paths which cannot be sensitized.

The error of reporting a false path is a conservative error, in that all real timing violations are considered by the analyzer. The reporting of a large number of false paths, however, may divert a design teams' energies towards fixing non-existent problems. In response, researchers have proposed a variety of path sensitization methods [5], [6], [8], [10]. The exact extent of the problem in industry, however, has not been known. This makes our work of special interest, since we performed our experiments on a state of the art, PowerPC microprocessor being designed at Motorola's Somerset PowerPC center, in Austin, Texas. We believe our results show that the reporting of false paths is a significant problem.

In our work, we post-processed a timing analyzer's output to determine how many false paths it was reporting. We realize this is a different problem than that of differentiating false from true paths as timing analysis is performed; however, our methods could be adapted to that purpose. We post-processed because our initial goal was to find test patterns that exercised critical timing paths, for use in hardware laboratory testing of finished chips. We wanted to eliminate false paths, since no set of vectors could elicit the purported, critical delays for these paths. We considered a path to be false if the set of transitions designated by the timing analyzer as causing the worst case delay could not be realized. We captured the constraints imposed on a timing path by justifying functions of symbolic variables across latch boundaries, in a manner reminiscent of sequential ATPG. In this way, we were able to detect sequentially as well as combinationally false paths.

[*] PowerPC is a trademark of the International Business Machines Corporation, used under license therefrom.

2 Background

We define a *timing path* as

- a set of circuit nodes, connected through devices, and
- a designated clock event, upon which the nodes change value, with some delay.

Most often, a path's first node will be the output of a latch and its last node the input to another latch. We define a *path stem* as a contiguous subset of a path's nodes beginning with the first node.

The firing of a path is related to transitions on its nodes. For some path, P, with nodes $n_1, \cdots, n_m$, we define $Before$ and $After$ sets of binary values on these nodes such that the value of the i^{th} node, n_i, in the *After* set is the complement of its value in the *Before* set. Path P is *fired* when its nodes transition from the $Before$ to the $After$ values on the designated clock event. We will assume, here, that the $Before$ and $After$ values are those designated by a timing analyzer as creating the worst case delay for the path in question.

A path, then, is a *true* path if there is an input sequence to the entire circuit in which it is embedded such that the $Before$ to $After$ transition can occur, and it is a *false path* otherwise. We classify false paths as either *combinationally* or *sequentially* false. A path is combinationally false if, for either the $Before$ or $After$ node values, no assignment exists to circuit latches and primary inputs such that those valuations can be realized. A path is *sequentially false* if it is not combinationally false, and either

1. the *Before* or *After* node values are realizable only in unreachable states, or
2. there is no transition from a state realizing the *Before* values to a state realizing the *After* values.

Here, a *state* is a valuation on circuit latches, and a state is *reachable* if there is some input sequence that drives the circuit to that state from a valid, initial state. While there has been previous work on sequential false paths [1] [6], most research has considered only combinational falseness. We feel strongly that both need to be considered.

```
 1  Compute functions of timing path nodes.
 2  Compute $Before_0$ and $After_0$ functions:
        If $Before_0$ or $After_0$ not satisfiable, exit, path is combinationally false.
 3  Set $i = 1$.
 4  Let $After_i$ = $After_{i-1}$ function **justified**.
        If $After_i$ not satisfiable, exit, path is sequentially false.
 5  If $After_i$ and $Before_{i-1}$ support overlap,
        compute $After_i = After_i \wedge Before_{i-1}$,
        else, go to 7.
 6  Check satisfiability of new $After_i$:
        If not satisfiable, exit, path is sequentially false.
 7  Let $Before_i$ = $Before_{i-1}$ function **justified**.
 8  Check satisfiability of $Before_i$:
        If not satisfiable, exit, path is sequentially false.
 9  If $i = n$, exit, cannot determine falseness examining $n$ banks.
10  Increment $i$, and go to 4.
```

Figure 1: False Path Detection Algorithm

3 False Path Detection Algorithm

The $Before$ and $After$ node values for a timing path can be viewed as Boolean functions. Let us consider the $Before$ function for a path, P, having m nodes, $n_1, \cdots, n_m$, these nodes implementing Boolean functions $f_1, \cdots, f_m$. Each f_i is a function over variables representing circuit latches and PIs (primary inputs). We form the $Before$ function by ANDing together true and complemented forms of $f_1, \cdots, f_m$, using the true form, f_i, if node n_i is a 1 in the $Before$ node value set, and the complemented form, $\neg f_i$, otherwise. The $After$ function is formed similarly, in relation to the $After$ node values. Recall that the value of any node, n_i, in the $After$ node value set, is the complement of its value in the $Before$ node value set. This is because each node along a path must transition, for a signal to be transmitted.

For the $Before$ to $After$ transition to take place, the $Before$ and $After$ functions must each, first of all, be combinationally satisfiable. If this is the case, then additionally, the latches on which the timing path nodes immediately depend must be able to realize a sequence of two assignments such that the $Before$ and then the $After$ function is realized. There may be feedback among these latches and this can restrict the set of satisfying pairs, i.e., when $Before$ holds, it may be the case that feedback prevents some particular, satisfying assignment to $After$ next clock cycle. The requirement that at least one pair of satisfying assignments to $Before$ and then $After$ be realizable can be transported from a requirement on latch outputs to a requirement on latch inputs. The needed valuations for this two clock sequence must appear on inputs to latches one clock cycle before they appear on latch outputs. This requirement on latch inputs can, in turn, be propagated back to a second bank of latches and PIs. Each assignment in the sequence of two satisfying assignments is, itself, a Boolean function. Propagating requirements across a latch boundary, or what we call "justification", results, then, in the creation of new functions that must be satisfied with respect to a new bank of latches and PIs. This process of justification may continue until

- one reaches functions over only PIs, and these are satisfiable, or,
- a justified function is not satisfiable.

In the latter case, the path would be sequentially false. This is the intuitive description of the false path detection algorithm given in Figure 1.

In the algorithm in Figure 1, the number of banks of latches to analyze, n, may be set arbitrarily. In large, industrial designs, repeated justifications will likely result in functions that are difficult to manipulate. Thus, a reasonable n would be that found by experience. In fact, the algorithm could be run interactively, until the user sees fit to terminate. The hope is that a small number of justifications would be sufficient to prove a path sequentially false that is, indeed, that.

The algorithm in Figure 1 begins by forming the $Before$ and $After$ functions as described above, and testing them for satisfiability (lines 1 and 2). We implemented the algorithm using ROBDDs (reduced, ordered, binary decision diagrams) [3]. Satisfiability testing is a constant time operation once a ROBDD has been built, and the complexity is in building the ROBDDs. After testing for combinational falseness, on line 2, the algorithm goes into a loop, on lines 4 through 10, and tests for sequential falseness.

Testing for sequential falseness involves repeated *justification* of functions (lines 4 and 7). Justification of a function means, in our context,

- removing variables representing PIs, and
- substituting, for any variable representing the output of a latch, the input function for that latch.

We can state this more formally, using the $Before$ function as an example. Let $Before$ be a function over two vectors of n and m variables, $\bar{l} = (l_1, \cdots, l_n)$, and $\bar{i} = (i_1, \cdots, i_m)$, representing latches and PIs, respectively. The Boolean operation for justifying $Before(\bar{l}, \bar{i})$ is:

$$\exists \bar{l}, \bar{i} \left[(l_1 \leftrightarrow t_1 \wedge \cdots l_i \leftrightarrow t_i \cdots \wedge l_n \leftrightarrow t_n \wedge Before(\bar{l}, \bar{i}) \right]$$

where $\leftrightarrow$ has the meaning of $XNOR$, and where existential quantification of variable x_i from any function, f, i.e.,

$$\exists x_i \left[f(x_0, \cdots, x_i, \cdots, x_n) \right]$$

has the meaning

$$f\,|_{x_i=0} \vee f\,|_{x_i=1}$$

On line 4 of the algorithm, the current $After_i$ function is justified. Let us consider when $i = 1$. The $After_0$ and $Before_0$ functions formed on line 2 depend on variables representing latches and PIs upon which the timing nodes immediately depend. Justification of $After_0$ produces $After_1$, a function over a *second* bank of latches and PIs. $After_1$, intuitively, denotes assignments to these latches and PIs which, if realized at, say, time t, the inputs to the latches that drive the timing nodes will be such that one time step later, at time $t+1$, $After_0$ will be satisfied. Now, if feedback is present in the circuit, some latches upon which $After_1$ depends may be among those upon which $Before_0$ depends. It may also be the case that $Before_0$ and $After_1$ depend upon some subset of common PIs. These overlapping dependencies must not conflict, since when the first bank of latches and PIs is producing a satisfying assignment to $Before_0$, the second bank of latches and PIs must be producing values satisfying $After_1$. This argument can be made for any iteration, i, of the algorithm, i.e., there must be a transition driven by inputs to latch bank i, such that $Before_{i-1}$ is succeeded by $After_i$. If the function produced on line 5 is satisfiable this is the case, and latch bank i can sustain the needed transition *if more distant latch banks can supply the needed valuations*. In preparation for the next iteration of the algorithm, the current $Before_i$ function is justified, on line 7. This must, in turn, produce a satisfiable function, or the path is sequentially false.

For brevity, we did not insert a check on initial states into the algorithm; but, such a check could be carried out in practice. A path can be deemed a true path if one is representing all latches in each bank, and, when these are set to values consistent with initial circuit states, it is found the needed transition can take place. It is expected, however, that for industrial designs the algorithm will be run on sub-circuits cut from a large design, and in this case some latch outputs may be modeled as PIs. The algorithm will still give a sound result with respect to proving a path false in this case, but cannot be relied upon to prove it a true path. In addition, a check on arriving at functions of all PIs was omitted from the algorithm, because, due to feedback among latches, this is unlikely to be encountered on industrial designs.

We implemented the algorithm using the *Versys* verification system at Motorola's PowerPC design center, Somerset. *Versys* is based on the *Voss* [9] system, primarily developed by Carl Seger. *Voss* has an entry language called FL, a functional programming language in which Boolean data types are implemented as ROBDDs. Due to the excellent list-handling abilities of FL, the algorithm of Figure 1 was implemented in less than 100 lines of code. A separate C++ program parsed circuit descriptions and provided the FL program its input, these being the lists of Boolean variables and Boolean formulae. The latter were given as strings and the FL interpreter implemented them as ROBDDs. The FL language has a number of built in functions that manipulate ROBDDs, similar to what one normally finds in separate ROBDD packages. This combination of a functional programming language and ROBDD manipulation abilities made it an excellent vehicle for implementing this algorithm.

4 Experimental Results

At Motorola's Somerset PowerPC design center, an internally developed static timing analyzer, STEP, is used. Circuit devices are characterized by timing rules and information on wire delays is added, when available. Designers may package up any sort of circuitry to be treated as single components, and thus the "classical" timing path, consisting of combinational devices between latch output and latch input, is not the only type of path encountered. The legal input/output combinations of path components are described in timing rules, but the analyzer does not know the internal makeup of the components, nor does it understand their logic functionality. Entire on-board RAM arrays, such as caches and register files can be, and are, handled as single components.

4.1 Results of Simulation Monitoring

We have implemented a program called *SiMon* at Somerset to enable monitoring, during functional simulation, for the firing of critical timing paths. SiMon enables this by a gate to RTL level translation, since timing analysis is done at the gate level, after logic synthesis is performed. Logic synthesis will, in general, reconstruct Boolean networks between latches, adding new circuit nodes and removing existing nodes, with respect to circuit nodes visible at the RTL level. SiMon creates a mapping from the gate back to the RTL level, enabling the tracking of timing paths during simulation.

We used SiMon, at the Somerset PowerPC design center, to monitor 40 timing paths on a recent PowerPC microprocessor design, over some 680 million simulation cycles. The original purpose was to find patterns that fired these paths for later use in hardware laboratory testing. We found, however, that the false path problem greatly interfered with this goal. The first 20 paths among the 40 chosen were those the design team would consider their most critical paths. These might, however, share large path stems. The other 20 were chosen to reduce path stem overlap. They differ from the first 20 and from each other by at least 5 nodes.

The results of the monitoring are given in Figure 2. Paths with RG prefixes are the top 20 paths, those with UN prefixes, the remainder. Each RG path is more critical (longer delay) than any UN path, and the lower the numerical suffix of any path, the higher its criticality within its group.

The propagation column marks the longest chain of transitions that occurred along a path stem, where the transition polarities were those specified by the timing analyzer. We also monitored transitions with arbitrary polarities, though we do not list those here. Propagation is given as a ratio: longest propagation vs. total number of path nodes. Most paths exhibited propagation over less than half their length.

A Y in either the $Before$ or $After$ column denotes that those node values appeared at least once in simulation. For only four paths did the $Before$ and $After$ values appear in succession, meaning, the path fired. These four were relatively short paths with RAM array components, as indicated by the Y in the *Array* column. Since designers know that the delay through an array can be very large, signals are latched very close to array boundaries. These short paths are more likely to be true paths, and there is a high probability of exercising them with a randomly chosen set of vectors. Most of the critical timing paths went through synthesized control logic and were fairly long paths. Not a single "classical" path of this type fired in simulation, and, none exhibited arbitrary transitions down its entire length, either.

4.2 False Path Analysis Results

We ran the false path detection algorithm of Figure 1 on path stems within circuit partitions made for logic synthesis. We treated inputs to a partition as primary inputs. In all but two cases, we were able either to prove a path false by looking at the path stem within a single partition, or we ran out of memory trying to do so.

The results of running the false path detection algorithm are shown in Figure 3. The "Nodes Tested" column gives the number of nodes which were tested that were outputs of combinational devices. The first node in a path comes directly out of a latch and was not counted. We used data from *SiMon* to estimate the length of path stem needed for false path detection, examining to just beyond where the longest propagation had occurred in simulation. If, in the "Nodes Tested" column, another path is listed, this means the two paths share the same, tested stem. Paths were sometimes not tested because of parser limitations. We used a parser for gate level netlists built from standard cell components. Most path involving arrays do not involve such circuitry. When the algorithm was quit

Path	Prop	Before	After	Fire	Array	Path	Prop	Before	After	Fire	Array
RG_1	21/30	Y	Y	N	N	UN_1	7/21	Y	N	N	N
RG_2	21/30	Y	Y	N	N	UN_2	7/21	Y	N	N	N
RG_3	21/30	Y	Y	N	N	UN_3	7/21	Y	N	N	N
RG_4	21/30	Y	Y	N	N	UN_4	0/33	N	N	N	N
RG_5	7/21	Y	N	N	N	UN_5	21/30	Y	Y	N	N
RG_6	2/18	Y	Y	N	N	UN_6	7/21	Y	N	N	N
RG_7	0/33	N	N	N	N	UN_7	1/5	Y	Y	N	Y
RG_8	7/21	Y	N	N	N	UN_8	7/21	Y	N	N	N
RG_9	2/26	N	N	N	N	UN_9	7/21	Y	N	N	N
RG_10	21/30	Y	Y	N	N	UN_10	4/4	Y	Y	Y	Y
RG_11	7/21	Y	N	N	N	UN_11	6/6	Y	Y	Y	Y
RG_12	21/30	Y	Y	N	N	UN_12	4/4	Y	Y	Y	Y
RG_13	21/30	Y	Y	N	N	UN_13	2/24	N	Y	N	N
RG_14	7/21	Y	N	N	N	UN_14	1/8	N	Y	N	Y
RG_15	21/30	Y	Y	N	N	UN_15	20/31	Y	N	N	N
RG_16	0/33	N	N	N	N	UN_16	15/15	Y	Y	Y	Y
RG_17	1/5	Y	N	N	Y	UN_17	13/23	N	Y	N	N
RG_18	7/21	Y	N	N	N	UN_18	7/21	Y	N	N	N
RG_19	7/21	Y	N	N	N	UN_19	1/8	N	Y	N	Y
RG_20	21/30	Y	Y	N	N	UN_20	2/28	N	Y	N	N

Figure 2: Simulation Monitor Results

due to memory overflow, the iteration on which this occurred is noted.

In summary, of the 40 timing paths considered,

- 15 proved to be false, 14 of these combinationally, 1 sequentially.
- 16 caused ROBDD blowup with inconclusive results,
- 5 could not be analyzed because of temporary parser limitations,
- 4 had already fired in simulation, and so were known to be true paths.

For the path found to be sequentially false, the algorithm exited on the second iteration on line 6. Thus, when the $Before$ function held, feedback prevented the $After$ function from holding one time step later.

Of the 14 paths found to be combinationally false, 12 shared the same stem, and the $After$ values for this stem were combinationally unrealizable. Likewise, 10 of 16 paths for which ROBDD memory blowup occurred shared a common stem, the RG_1 stem. We knew these paths were not combinationally false, since the $Before$ and $After$ node values had appeared in simulation. However, none of these 10 paths had ever fired in over 680 million simulation cycles. Unfortunately, the logic functions of the shared stem caused ROBDD blowup.

The case of paths RG_9, UN_13 and UN_20, is of interest. *SiMon* reported that these paths had never transitioned beyond the 2^{nd} of their nodes, strongly suggesting sequential falseness. However, the latches driving these nodes were driven by some 89 latches and 116 primary circuit inputs, and ROBDD blowup was experienced on the second iteration of the algorithm. More will be said about these paths in the next section.

These experiments were run on an IBM RS6000 workstation equipped with 256 Megabytes of local memory. We did not keep track of memory usage because it was slight for the paths which were proven false, and out of range for the others. All paths proven false were proven so in less than 3 minutes of (wall clock) run time.

5 Future Work

In our future work, we intend to explore non-ROBDD techniques and to make better use of abstraction when applying ROBDDs. We have already started using satisfiability solvers [4] in place of ROBDDs, and have achieved some promising first results. We have used the bounded model checker, BMC, from Carnegie-Mellon University. BMC implements a technique of state space exploration combined with satisfiability solving on propositional formulae. We were able to prove, using it, that the three paths, RG_9, UN_13 and UN_20, discussed above, were sequentially false. We designated the $Before$ predicate for RG_9, UN_13 and UN_20 as the initial state predicate of the circuit, and proved that a transition to a state satisfying the $After$ predicate was impossible. However, we believe such proofs could be carried out better by adapting our false path detection algorithm to using satisfiability solvers. Our algorithm goes backwards in time as opposed to going forwards in time, as bounded model checkers do. For instance, the technique we used with BMC only checked the first two banks of latches. We feel going backwards in time facilitates analysis of multiple latch banks. We have begun work on a prototype of our algorithm using satisfiability solvers, and hope to publish new results, shortly.

Our other direction would be to find abstractions that make ROBDD building easier. We wish to utilize the new ABDD data structure outlined in [7]. These result in reduced ROBDD graphs according to an abstraction function the user defines. Our hope is that by using smaller ABDDs we will be able to justify across more latch banks. Our use of ABDDs would be a conservative abstraction, i.e., any determination that a path was false would be sound. In addition, we plan to try very straightforward techniques of existentially quantifying away variables representing latches or PIs, or arbitrarily cutting circuits at certain points.

References

[1] T. Chakraborty, V. Agraawal, *Effective Path Selection for Delay Fault Testing of Sequential Circuits.* International Test Conference, 1997.

Path	Nodes Tested	Result	Path	Nodes Tested	Result
RG_1	21	memory out, iter. 1	UN_1	tested RG_5	comb false
RG_2	tested RG_1	memory out, iter. 1	UN_2	tested RG_5	comb false
RG_3	tested RG_1	memory out, iter. 1	UN_3	tested RG_5	comb false
RG_4	tested RG_1	memory out, iter. 1	UN_4	untested	-
RG_5	8	comb false	UN_5	tested RG_1	memory out, iter. 1
RG_6	8	seq false	UN_6	tested RG_5	comb false
RG_7	untested	-	UN_7	untested	-
RG_8	tested RG_5	comb false	UN_8	tested RG_5	comb false
RG_9	2	memory out, iter. 2	UN_9	tested RG_5	comb false
RG_10	tested RG_1	memory out, iter. 1	UN_10	-	true path
RG_11	tested RG_5	comb false	UN_11	-	true path
RG_12	tested RG_1	memory out, iter. 1	UN_12	-	true path
RG_13	tested RG_1	memory out, iter. 1	UN_13	tested RG_9	memory out, iter. 2
RG_14	tested RG_5	comb false	UN_14	4	memory out, iter. 3
RG_15	tested RG_1	memory out, iter. 1	UN_15	20	memory out, iter. 1
RG_16	untested	-	UN_16	-	true path
RG_17	untested	-	UN_17	14	comb false
RG_18	tested RG_5	comb false	UN_18	tested RG_5	comb false,
RG_19	tested RG_5	comb false	UN_19	4	memory out, iter. 3
RG_20	tested RG_1	memory out, iter. 1	UN_20	tested RG_9	memory out, iter. 2

Figure 3: False Path Algorithm Results

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu *Symbolic Model Checking using SAT procedures instead of BDDs*, Proceeding Design Automation Conference, 1999.

[3] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, Vol. C-35, No. 8, August, 1986.

[4] M. Davis, H. Putnam, *A Computing Procedure for Quantification Theory*. Journal of the Association for Computing Machinery, vol. 7, 1960.

[5] H. Chang, J. Abraham *An Efficient Critical Path Tracing Algorithm for Designing High Performance VLSI Systems.* Journal of Electronic Testing, Theory and Applications, 11, pp. 119-129, Kluwer Academic Publishers, 1997.

[6] H. Chang, *Strategies for Design and Test of High Performance Systems.* Ph.D. Dissertation, The University of Texas at Austin, August 1993.

[7] S. Jah, Y. Lu, M. Minea, E. Clarke, *Equivalence Checking Using Abstract BDDs*, ICCD, 1997.

[8] Y. Kukimoto, R. Brayton *Exact Required Time Analysis via False Path Detection*. Design Automation Conference, 1997.

[9] C.-J. Seger *Voss–A Formal Hardware Verification System User's Guide*, Tech. Rep. 93-45, Dept. of Comp. Sci., Univ. of British Columbia, 1993.

[10] P. McGeer, A. Saldanha, R. Brayton, A. Sangiovanni-Vincentelli *Delay Models and Exact Timing Analysis.* in T. Sasao, editor, Logic Synthesis and Optimization, pp. 167-189, Kluwer Publishers, 1993.