A Path Selection Algorithm for Timing Analysis

H.C. Yen *S. Ghanta* *H.C. Du*

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Abstract

Due to the rapid progress in semiconductor technology, the number of gates that can be placed in a chip increases dramatically. Existing algorithms for timing analysis have difficulties when dealing with large designs. A new algorithm for timing analysis is proposed in this paper. This algorithm enumerates all the paths with delay greater than a given threshold. The execution time of the proposed algorithm is proportional to the number of paths generated. Therefore, it is suitable for large designs.

Classification: timing verification.

1. Introduction

Due to the rapid progress in semiconductor technology, the complexity of designing digital systems has increased dramatically. The gate count for a large to a very large computer is about a hundred thousand to one million. Since, the design of a large system is difficult and the production is expensive, product verification plays an important role in design automation. The major goal of product verification is to ensure that the physical design will perform the desired function. Therefore, product verification must provide the diagnostic information to correct the faulty parts of the design.

Product verification consists of three main parts, functional design verification, physical design verification and timing verification. Timing verification consists of ensuring the path delays —— from primary input or storage elements to primary output or storage elements —— are within the tolerable limits. Timing verification is performed by timing verifier which will check all the internal paths between registers and reports the paths, which are too long or too short with respect to given thresholds.

Timing verifier is primarily a tool for aiding the engineers who produce a design to meet a specific clock cycle. The output of the timing verifier not only identifies the logic with timing problems, but also provides a measure of the severity of how large or small is the delay of the logic path under consideration. Thus, the engineer is directed to the problematic part of the design right away.

Almost all existing timing verifiers formulate the problem of timing verification into that of a graph theoretic problem of finding the longest or shortest paths. The delays associated with the input and output pins of the devices are the ones that are modeled as the edge weights of the so formulated graph. However a significant number of authors considered the propagation delays through nets as negligible. Some consider them along with the device delays. They all assume that a digital circuit can be represented by a directed acyclic graph. If it is not true, then the cycles are broken by artificially duplicating nodes. Most of the earlier work is based on one of the following four approaches for the evaluation of the so formulated graph:
(a) Plain Depth-first search [2, 3, 6]
(b) Depth-first search with pruning [4]
(c) Hierarchical Depth-first search without pruning [5] , or
(d) Breadth-first search with pruning [1].

Depth-first search without pruning, also known as *path enumeration*, systematically traces all possible paths through the graph, and then computes the total delay along each path. This technique suffers from a severe problem of path explosion. Depth-first search with pruning adds an optimization: whenever a node is revisited in the course of search, the new arrival time is propagated to the child nodes only if it is worse than the previous worst arrival time at that node. This approach is used in Crystal [4]. However, regardless of the number of useful-paths, this still performs poorly on selected classes of examples wherein each updation forces the propagation of values through child nodes. The third approach, hierarchical depth-first search without pruning (also referred to as block oriented algorithms), has been considered by several authors (see for example [5, 6]), primarily for handling large scale designs. It starts by decomposing the complete design into a number of clusters. Each cluster is analyzed independently and the results of the analysis are propagated to higher levels of timing analysis. At higher levels each cluster is replaced by a single node in the graph, with delays being derived from results of the previous level. Typically, the complexity of this approach is of the order of the number of blocks (or clusters) [2]. However, that gain is at the cost of loss of information† in terms of a number of missing or ignored paths. Yet another alternative of PERT based critical path analysis has the drawback of that it will only produce the most critical path. Since timing analysis, as considered by most researchers, does not take into account the functionality of the devices, it is generally unknown without detailed simulation whether or not the critical path so enumerated will be enabled at all during the operation of circuit. So this approach in spite of its small computational complexity may not be that useful in practice.

† We assume that a number of critical paths will be reported in each execution of the algorithm so that the designer has the opportunity to fix some or all of them before initiating the next execution. In such a case hierarchical approach may miss certain critical paths(not the most critical one obviously).

In this paper, we focus on the algorithmic aspects of Timing Analysis based on enumeration. We shall assume the graph theoretic formulation of the timing analysis problem in which edge weights reflect the proper delays taking into account both the device delays and the propagation delays in the nets. Our primary objective, is to present an algorithm that enumerates all paths exceeding a specified threshold in a given circuit. We are particularly concerned about the asymptotic time complexity. The algorithm described will spend its resources only on the paths that shall be generated.

We shall first present the informal idea behind this algorithm in the next section. In Section 3, we present the formal algorithm in a pseudo algorithmic language. We then offer some concluding remarks in the last section. For detailed example, correctness and time complexity the reader is referred to [7].

2. Informal Idea

In practice the digital circuits which have to be analyzed may contain feedback loops that result in the formation of cycles in the corresponding graph. As mentioned already, in such cases a preprocessor to the algorithm breaks such loops by artificially splitting the nodes (or vertices) at the time of formation of a cycle.

The proposed algorithm accepts an acyclic directed graph as input. One of the nodes is identified as a starting node. The edges have weights that correspond to delays between the corresponding pins in the circuit under analysis. A threshold T is also specified so that all the paths whose lengths exceed T can be enumerated by the algorithm.

For the purpose of easy handling and simplifying the boundary conditions we shall modify the graph by adding a source node and a sink node. A dummy edge (an edge of zero delay) is added from the source node to the starting vertex (or vertices). The algorithm proceeds by partitioning the nodes of the graph into a set of equivalence classes, so that each class of nodes will be assigned the same "level number". The first level contains just the source node. The level numbering is carried out —— in a manner analogous to the breadth first numbering of a graph —— in such a way that two nodes will have an edge between them if and only if their level numbers differ by at most 1. This process is repeated until all the nodes of the (original) graph are visited and are assigned level numbers. The rationale behind this layer numbering is to see that all the paths from the starting vertex going through a vertex must pass through the nodes of the same level number or those of the next higher level number and none else. By doing this we can ensure that the maximum delay from any node to sink through any path to the sink can be computed by simply observing the maximum delays of nodes in the next level and the weights of edges among the vertices of the current and next level.

At that point all the leaf-nodes (nodes with outdegree zero) are extended through dummy edges and dummy vertices so as to bring the level numbers of all leaf nodes to be the same (maximum level number). Then a sink node is created and dummy edges are created connecting the leaf-nodes to the sink.

Having converted the graph into a canonical form by creating the source, sink and dummy nodes together with the dummy edges the maximum delay is computed by starting at the sink vertex and moving towards source one level at a time and thereby computing the worst possible delays from each vertex to the sink on the way(similar to [2]). The worst delay from a particular node to sink is the maximum over all the neighboring nodes the summation of the worst delay from the neighbor to the sink and the edge weight from the vertex under consideration to the neighbor. This computation can be carried out relatively easily since the edges span across two consecutive levels at most.

The set of adjacent vertices for each vertex $SUCC(v)$ is sorted in non-increasing order of the worst delays to the sink through each of those neighboring nodes. This ordering is to prune away the search space that will not result in any *useful-paths*† i.e., those that exceed the threshold. Because of the order on the vertices, the moment a path through one of the neighbors fails to produce a path of length exceeding T, one can immediately discard trying out all the other paths with the current partial path as a prefix (which are potentially exponential in the number of remaining stages to sink) and back up to an earlier vertex in the path.

The next phase of the algorithm is to enumerate the useful-paths. The worst path can be right away computed (using *nextnode*) by following the first vertex in the neighborhood set of each vertex starting from the source. The alternative useful-paths can be computed by the alternation of depth-first enumeration and pruning steps. Intuitively, this corresponds to traversing the state-space in a depth-first-search order, in which the algorithm finds out a useful-path and outputs it and then from there prunes away enough of the state space whereupon the next useful-path can be found. When a path is output, the pruning is carried out with the last component of the output path and working backwards towards the beginning of the path until, we find a node that can provide us an alternative useful-path. Note that we can clearly do that in linear time in the length of the path since at no vertex in the path we shall have to try different alternatives. If the next unused neighbor provides a useful-path the backtracking stops right away. Or else we need not even try any of the remaining unused vertices in the neighboring set since they have already been sorted and we know that none of the others can produce a useful-path either. So only a constant amount of time is required at each node.

3. Formal Algorithm

{Construct the Component + Cycle Detection and Elimination}
let V, E be the vertex and edge sets of the graph G.

{Creation of Source vertex, s}
$V \leftarrow V \bigcup s$; $E \leftarrow E \bigcup \langle s$,starting−vertex$\rangle$

{Layering and Breadth First Numbering; $L(v)$ = layer number of vertex v; L_{max} = max level number}
$Q \leftarrow s$ where s is the source; $L_{max} \leftarrow L(s) \leftarrow 0$; mark s as visited.
while Q is not empty **do**
begin
$v \leftarrow$ create a node in the graph
dequeue the first element q of Q and copy its information into v
if v is not a terminal-pin **then**
for each unvisited output-pin op **of** v **do**
begin
$L(op) \leftarrow L(v)+1$; **if** $L_{max} < L(op)$ **then** $L_{max} \leftarrow L(op)$

† We shall hence forth call the paths whose delays exceed T as *useful-paths*.

enqueue the output-pin into Q, together with the link timing information and mark it.
end
end

{Dummy node creation and the formation of sink, t}
for $l \leftarrow 1$ **to** L_{max} **do**
begin
for each v such that $L(v) = l-1$ **do**
begin
if v does not have a neighbor at level l
then
begin
$u \leftarrow$ create a dummy node; $L(u) \leftarrow l$;
$V \leftarrow V \bigcup u$
let u be neighbor of v with $d(v,u) = 0$;
$E \leftarrow E \bigcup <v,u>$
end
end
end;
$t \leftarrow$ create a sink node; $V \leftarrow V \bigcup t$
for each v such that $L(v) = L_{max}$ **do**
begin
let t be the neighbor of v such that $d(v,t)=0$;
$E \leftarrow E \bigcup <v,t>$
end

{Computation of Maximum Delay; D_{max} is the maximum delay of any path from s to t}
$max\_delay\_to\_sink(t) \leftarrow 0$
for $l \leftarrow L_{max}$ **downto** 0 **do**
begin
for each v such that $L(v) = l$ **do**
begin
$max\_delay\_to\_sink(v) \leftarrow$
$\max_{u \in SUCC(v)} \{max\_delay\_to\_sink(u)+d(v,u)\}$
where $SUCC(v) = \{w \mid <v,w> \in E$ and $L(w) \geq L(v)\}$
end
end;
$D_{max} \leftarrow max\_delay\_to\_sink(s)$

if $D_{max} < T$ **then exit;** { No paths to list out }

{Sorting of neighbors of each vertex}
for each v **do**
begin
Rearrange/Sort the adjacency list of v in the non-increasing order of the objective function:
$(max\_delay\_to\_sink(u)+d(v,u)) \quad \forall\, u \in SUCC(v)$
end

{Path Enumeration}
{Trace the worst path}
let $P = (s=v_0,v_1,v_2, \cdots ,v_q=t)$ where $v_{i+1}=nextnode(v_i)$
{Compute the delays from source to the nodes in the current path}
$delay\_from\_source(v_0) \leftarrow 0$
for $i \leftarrow 0$ **to** $q-1$ **do**
begin
$delay\_from\_source(v_{i+1}) \leftarrow$
$delay\_from\_source(v_i)+d(v_i,v_{i+1})$
end
$j \leftarrow q-1$
while $j > 0$ **do**
begin
output (*strip (P)*)
{Compute the new path}
let $P = (s=v_0,v_1,v_2, \cdots ,v_q=t)$ where v_i is:
for $0 \leq i \leq j$: v_i as in the existing P
for $i = j+1$: $nextnode(v_j,v_{j+1})$
for $i > j+1$: $nextnode(v_{i-1})$
{Compute the delays from source to the new nodes in the current path}
for $i \leftarrow j+1$ **to** $q-1$ **do**
begin
$delay\_from\_source(v_{i+1}) \leftarrow$
$delay\_from\_source(v_i)+d(v_i,v_{i+1})$
end
output ("Delay of the last output path is : ", $delay\_from\_source(v_q)$)
{Prune the state space}
find the largest j from among the ones belonging to P such that:
$nextnode(v_j,v_{j+1}) \neq$ **nil** and
$delay\_from\_source(v_j)+d(v_j,v)+$
$max\_delay\_to\_sink(v) > T$ where
$v=nextnode(v_j,v_{j+1})$
end

$nextnode(v)$ is that neighbor node of v which results in the worst delay to sink.
$nextnode(u,v)$ is that neighbor node of u which occurs after the vertex v in the sorted adjacency list of u. It is **nil** if no such node exists. In other words it is that neighbor vertex that results in the worst path length to sink, that is not greater than the worst path length obtained by going through u.

4. Experiments and Conclusion

We have compared the proposed algorithm with an existing package called DAMSEL from Honeywell. DAMSEL is a tool for static timing analysis. It consists of a path enumeration algorithm and a block oriented algorithm. The experimental results presented in this section are compared with the algorithms of DAMSEL. The path enumeration algorithm of DAMSEL is plain depth first enumeration of all paths and then filtration of the paths that don't qualify as useful ones. Unlike the proposed algorithm, this need not go through a number of preprocessing steps such as layering, dummy node creation or max-delay computation. Therefore, the path enumeration algorithm of DAMSEL is a little bit faster than the proposed algorithm when all possible paths are enumerated (i.e., given a small threshold). It is not fair to compare the proposed algorithm with the block oriented approach except for the case when only one path is generated. Results are presented in Table 2 and 3.

We have run both algorithms for more than 40 instances. However, due to the space limitation we can present only two instances. Each instance is a graph with $n=|V|$ nodes and $m=|E|$ edges. Each node has a maximum degree (max_degree) and a minimum degree (min_degree). Both algorithms are executed on a Sun 50/3. Consider the example discussed in the previous section. Since this example is such a simple one and it has altogether 10 paths from source node to sink, the difference in execution times will not be much. For example the execution times (CPU seconds of Sun 50/3) are 0.624 and 0.626 respectively, for the depth first search and the proposed algorithm.

In Tables 2 and 3, we present the results for graphs of 1000 and 3000 nodes respectively. Similar results

have also been obtained for other instances. It is evident that the execution time required by the proposed algorithm is proportional to the number of paths generated. When all paths are generated, the proposed algorithm may require a little bit more time than that of a depth-first path enumeration algorithm. If a bigger threshold is given, the execution time required is much less as the results in Table 2 and 3 indicate. Therefore, it is suitable for big designs, especially, when the designers have some idea of the "critical" threshold. When there are paths with delays greater than this "critical" threshold, the timing specifications of the design may not be met or the performance of the design has to be reduced.

In this paper we have mainly concentrated on the basic algorithmic aspects of timing analysis. The major limitation of the proposed algorithm is that of finding the "critical" threshold. Yet another limitation is that of potentially large time complexity if there are many paths of the same delay, as is the case with some tight designs. Currently we are investigating ways in which these limitations can also be overcome. We plan to test the algorithm by executing several "real" designs.

5. Acknowledgement

We thank professor Patrick Powell for bringing our attention to this problem. This work was supported in part by NSF Grants MIP-8605297 and DCR-8420935.

6. References

[1] Hitchcock, R. B., Smith, G. L., and Cheng, D. D. "Timing Analysis of Computer Hardware," *IBM Journal of Research and Development, Vol. 26*, 1, January 1982, 100-105.

[2] Hitchcock, R. B. "Timing Verification and the Timing Analysis Program," *Proceedings of the 19th Design Automation Conference*, 594-604, 1982.

[3] Larson, B. "DAMSEL Users Manual," *Honeywell SSED Design Technology*, Honeywell Corporation, Minneapolis, Minnesota, April 1987.

[4] Ousterhout, J. K., "A Switch-Level Timing Verifier for Digital MOS VLSI," *IEEE Transactions on Computer Aided Design, Vol. CAD-4*, 3, July 1985, 336-349.

[5] Reddi, R. and Chen, C. "Hierarchical Timing Verification System," *Computer Aided Design, Vol. 18*, 9, November 1986, 467-477.

[6] Sasaki, T., Yamada, A., Aoyama, T., Hasegawa, K., Kato, S. and Sato, S. "Hierarchical Design Verification for Large Digital Systems," *Proceedings of 18th Design Automation Conference*, 105-112, 1981.

[7] Yen, H.C., Ghanta, S., and Du. H.C. "Timing Analysis Algorithms for Large Designs," Technical Report - CSD-TR-87-57, Computer Science Department, University of Minnesota, September 1987.

n=1000, m=3000 Max_Degree=9, Min_Degree=1 Max_delay=10, Min_delay=1		
Block-Oriented algorithm	Time= 1 s.	
	p= 1	
Proposed algorithm	Threshold = 239.0 ns.	Time= 1 s.
		p= 1
	Threshold = 191.2 ns.	Time= 13 s.
		p= 2246
	Threshold = 143.4 ns.	Time= 315 s.
		p= 82044
	Threshold = 95.6 ns.	Time= 1135 s.
		p= 399997
	Threshold =47.8 ns.	Time= 1388 s.
		p= 546321
	Threshold = 1.0 ns.	Time= 1408 s.
		p= 550065
Path-Enumeration algorithm	Time= 1402 s.	
	p= 550065	

Time = CPU time required in seconds
p = Number of paths enumerated

Table 2

n=3000, m=9000 Max_Degree=9, Min_Degree=1 Max_delay=10, Min_delay=1		
Block-Oriented algorithm	Time= 2 s.	
	p= 1	
Proposed algorithm	Threshold = 220.0 ns.	Time= 4 s.
		p= 1
	Threshold = 176.0 ns.	Time= 26 s.
		p= 6280
	Threshold = 132.0 ns.	Time= 809 s.
		p= 310046
	Threshold = 88.0 ns.	Time= 4003 s.
		p= 1849172
	Threshold =44.0 ns.	Time= 5086 s.
		p= 2462610
	Threshold = 1.0 ns.	Time= 5094 s.
		p= 2471449
Path-Enumeration algorithm	Time= 5089 s.	
	p= 2471449	

Time = CPU time required in seconds
p = Number of paths enumerated

Table 3