Graph-based Dynamic Analysis: Efficient Characterization of Dynamic Timing and Activity Distributions

Hari Cherupalli and John Sartori
University of Minnesota
Email: {cheru007, jsartori}@umn.edu

***Abstract*— In light of increasing energy overheads required to guarantee correctness as variations increase with continued technology scaling, better-than-worst-case (BTWC) design has become a hot topic. Several BTWC design techniques utilize dynamic information like path activity when optimizing a design and rely on path-based analysis to determine the dynamic slack distribution of a workload running on a processor and subsequently optimize a design. In this paper, we show that path-based techniques are not scalable, due to the enormous number of paths in modern designs, and can also result in incorrect results. We propose a graph-based technique for performing dynamic timing and activity analysis of a workload on a processor that addresses the limitations of path-based techniques. Our tool has significantly lower runtime and memory requirements than path-based tools. Consequently, we can perform analysis for larger designs over longer time windows in a shorter amount of time. We also propose two optimizations that improve the performance of our tool.**

I. Introduction

As challenges in technology scaling have resulted in increasing static and dynamic variations, along with increasingly restrictive design guardbands that ensure correctness even in the worst case, researchers have introduced better-than-worst-case (BTWC) design techniques that relax conservative design constraints, possibly at the expense of less than perfect correctness, in order to improve energy efficiency under average conditions [1], [2], [3], [4], [5], [6].

BTWC design techniques rely on error tolerance or correction mechanisms to handle errors when worst case conditions occur, allowing a processor to be optimized for and operated at a BTWC condition, potentially resulting in significant energy savings. Several BTWC design techniques exploit not only static design information, such as timing and power characterizations, but also dynamic information, such as activity factors, that describe how a design is used. Dynamic information describes which parts of a design are most likely to be exercised or to produce errors under BTWC conditions. Such information allows a designer to optimize for BTWC conditions, where errors may occur, and make a design more efficient in the face of errors. Since these techniques are used only for design optimization and not for timing closure, they do not require worst-case inputs for the simulated benchmarks.

Several BTWC design techniques have been proposed that exploit dynamic information characterizing the activity of paths in a design to perform optimizations and improve energy efficiency in variation-affected designs [2], [3], [4], [5], [6], [7], [8]. A study of dynamic analysis-based design techniques reveals that all such techniques rely on path-based analysis and optimization methodologies. The distinguishing characteristic of these path-based methodologies is that the paths (or the exercised paths) in a design must be enumerated, individually analyzed, and optimized.

However, due to the very large number of paths in modern designs [9], path-based analysis and optimization become onerous and in most cases infeasible, even for small designs. Consequently, previously-proposed dynamic analysis and optimization techniques have been limited to working with only small design modules over small analysis time windows, due to the large computation time and memory requirements of path-based analysis and optimization [2], [3], [4], [5], [6], [10]. This has limited their applicability in modern semiconductor designs, which can often contain thousands of gates, and many orders of magnitude more paths [9]. An additional consequence of this module-based approach is that paths between modules and paths that span multiple modules are ignored during analysis and optimization. Since it does not consider the full design, module-based analysis and optimization may produce incorrect or suboptimal results.

In this paper, we propose a novel dynamic analysis technique that is designed around graph-based, rather than path-based, analysis. Our approach leverages the observations that a set of gates in a design maps to a unique set of paths in the design (see Section III). Thus, we can characterize the exercised paths in a design by identifying and analyzing the exercised gates. We propose a novel methodology that leverages the speed and memory benefits offered by commercial static timing analysis (STA) engines to quickly characterize the dynamic critical path distribution of a design for a particular workload. Our dynamic analysis tool can also characterize path activities for the design. Graph-based analysis significantly outperforms path-based analysis (by 105.6x in our experiments), and we present two optimizations that further improve the performance and reduce the memory footprint of our technique, also discussing the tradeoffs between the approaches. We demonstrate that our graph-based dynamic analysis technique can efficiently analyze large designs over large time windows, even full processor designs, without ignoring parts of the design such as cross-module paths.

Our paper makes the following contributions.

- We propose a graph-based dynamic timing and activity analysis tool [1] that reduces computation time and memory footprint compared to previously-proposed path-based analysis techniques. To the best of our knowledge, this is the first non-path-based dynamic analysis methodology.
- Whereas path-based techniques are limited to analyzing small subsets of modules over small time windows (using considerable computation time and memory resources to do so), we demonstrate that our tool can process large designs over large time windows, even full processor designs over full benchmark runs.
- We propose optimizations that improve the performance of our dynamic analysis tool. Uniquification-based dynamic analysis reduces effort by 76.6%, and analysis based on Unique Non-Includible Toggled-sets (UNITs) reduces the effort by 83.9%
- We demonstrate up to 136.6x (105.6x, on average) speedup in runtime compared to a path-based analysis tool (even for a small design module over a small time window) and show that the benefits of our approach improve considerably with increasing design size or analysis time window.

II. Related Work

Previous works that perform dynamic timing and activity analysis and optimization use path-based tools. [6] proposes a BTWC design

[1]Our automated tool is available for download at http://www.ece.umn.edu/users/jsartori/tools.html

methodology that uses path activity information to adjust path timing constraints and minimize errors for a frequency overscaled design. [8] proposes micro-architectural techniques that trade-off variation-induced errors for power and performance of a processor. They rely on the VATS model [7] which computes the dynamic slack distribution of a processor for a workload. [3] proposes power-aware slack redistribution where paths are optimized based on timing criticality and toggle rate to improve power and area efficiency under voltage scaling. [1], [2] propose a recovery-driven design methodology for optimizing a design for a specific target error rate. The methodology relies on path-based activity and timing analysis, and resizes gates to optimize a design on a path-by-path basis. [4] proposes architectural optimizations to manipulate timing error rate behavior and increase the effectiveness of timing speculation. [5] proposes compiler techniques that improve the energy efficiency of timing speculative processors.

The above BTWC techniques all rely on path-based timing and activity analysis, and many of the techniques also perform path-based design optimization. These techniques involve enumeration of paths and are not scalable, due to the extreme number of paths in electronic designs [9]. As a result, application and evaluation of these techniques are limited to small modules and small analysis time windows. In addition to not being able to handle full designs, module sampling methodologies ignore paths between modules and those that cross module boundaries.

Since path-based techniques are not scalable, other works employ alternative techniques that either produce inexact results or do redundant work. [10], [11], [12] run multiple gate-level simulations at different operating points for error rate computation. In contrast, our technique captures the path profile of a workload (or instruction sequence) in a single gate-level simulation, and the error rates at different operating points can be computed significantly faster by recomputing gate delays and performing STA. [13] proposes a clustered timing model to capture the dynamic delay distribution of a processor. Their approach requires manual analysis of the architecture and produces inexact results because of architectural approximations. In contrast, our technique is not only architecture independent, but it also does not introduce any approximations that degrade accuracy.

III. Preliminaries

Before explaining our dynamic analysis techniques, we define some terms and derive the necessary theorems to support our methodology. The theorems in this section are applicable to graphs in general. We, however, apply them to the context of a gate-level netlist of a digital design.

A. Definitions

Given a design's gate-level netlist, we define the following :

G	$\rightarrow$	Graph of the design containing gates and nets.
$p(A)$	$\rightarrow$	Set of all paths in the graph A.
$g(A)$	$\rightarrow$	Set of all gates (vertices) in the graph A. (Note that we use the terms gate and vertex interchangeably in the paper.)
$f(A)$	$\rightarrow$	Set of path endpoints (flip-flops, clock gates, etc.) of the design represented by graph A. Note that f(A) is a subset of g(A), i.e., we consider all path endpoints as gates.
p_i	$\rightarrow$	A particular path.
g_i	$\rightarrow$	A particular gate.

Definition 1. *Path: A set of gates $\{g_a, g_b, ..., g_n\}$ of a graph A can be considered a path if (1) an ordered sequence containing all the gates in the set can be formed such that each gate in the sequence is driven by the previous gate and (2) only the first and last gates of the sequence belong to $f(A)$.*

Definition 2. *Toggled gate: A gate is toggled in a particular cycle when the net that the gate is driving has changed values in that cycle.*

Definition 3. *Toggled Path: A path is toggled in a particular cycle if all the gates in the path have toggled in that cycle.*

Definition 4. *Non-Toggled Path: A path is non-toggled in a particular cycle if at least one gate in the path has not toggled in that cycle.*

Definition 5. *Gate-set: A gate-set is any vertex-induced sub-graph of the graph G. (A vertex-induced subgraph is a subgraph defined by a set of vertices that contains all the edges between those vertices.)*

Definition 6. *Toggled-set: A gate-set containing all the toggled gates of G and no non-toggled gates of G for a given time stamp is a toggled-set.*

B. Theorems

Theorem 1: A toggled-set of a design's graph G contains all the toggled paths in G and does not contain any non-toggled paths.

Proof: We prove both parts of the theorem by contradiction. Let A be a toggled-set of graph G containing all toggled gates for a particular analysis time stamp.

Completeness: Suppose there exists path $p_1 = \{g_1, ..., g_n\}$ in $p(G)$ such that p_1 has toggled but $p_1 \notin p(A)$. This implies that at least one of the gates $g_1, ..., g_n$ does not belong to $g(A)$.

Let $g_k \notin A$, which implies g_k has not toggled, since A, by definition, contains all toggled gates and no non-toggled gate.

This leads to a contradiction that path p_1 has not toggled, from the definition of a non-toggled path.

Exclusivity: Suppose path $p_2 = \{g_m, g_{m+1}, ..., g_{m+l}\}$ is a non-toggled path such that $p_2 \in p(A)$. By definition of a non-toggled path, at least one $g_m, g_{m+1}, ..., g_{m+l}$ has not toggled. This is a contradiction, since A contains *only* the toggled gates of G. ■

Note that the exclusivity clause of Theorem 1 assumes that (1) a net in a digital design is connected to the output pin of only one gate, and (2) every toggled input of a gate contributes to the toggle of the gate's output. The first assumption does not hold if the net is driven by multiple tri-state buffers. The second assumption does not hold for tri-state buffers driving multi-driven nets and multiplexers, which are considered as cells in certain cell libraries. It also does not hold in a case where a fast-arriving controlling input renders later-arriving toggles at other inputs ineffective. Since exceptions to these assumptions do not affect completeness, a toggled-set always completely characterizes the set of toggled paths. Section VII discusses techniques to maintain exclusivity even in these exceptional cases.

Theorem 2: Let A & B be two gate-sets of a design's graph G. If $g(A) \subseteq g(B)$ then $p(A) \subseteq p(B)$.

Proof: Let path p_1 be a path $\{g_r, g_{r+1}, ..., g_{r+s}\}$ such that $p_1 \in p(A)$ and $p_1 \notin p(B)$

This implies at least one of $g_r, g_{r+1}, ..., g_{r+s}$ does not belong to $g(B)$, say g_t.

Now, $g_t \in g(A)$ and $g_t \notin g(B)$.

But $g(A) \subseteq g(B)$, which is a contradiction, since all elements in $g(A)$ must also be in $g(B)$. ■

Corollary 1: If two toggled-sets A & B of a graph G have the same set of vertices (gates), then they have the same set of paths. This follows directly from Theorem 2.

C. Examples

We illustrate the above theorems with an example for each theorem. Consider the circuit in Figure 1. The ports A through F can be replaced with any of the legal endpoints for a path, such as flip-flops, clock-gates, etc. This circuit has 9 paths, as listed and indexed below.

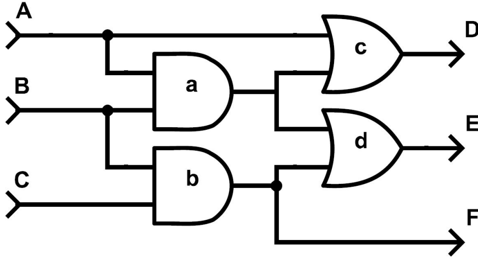

Fig. 1. An example circuit to illustrate Theorem 1 and 2.

1) A, c, D
2) A, a, c, D
3) A, a, d, E
4) B, a, c, D
5) B, a, d, E
6) B, b, d, E
7) B, b, F
8) C, b, d, E
9) C, b, F

To illustrate Theorem 1, let us assume that in a particular cycle ports A, C, D, E, F and gates b, c, d have toggled. This means that paths 1, 8, and 9 have toggled. However, any path containing gate a (paths 2,3,4 and 5) will not be considered in the sub-graph.

To illustrate Theorem 2, consider two different cycles. In one cycle, ports B, C, E, F and gates b, d have toggled while in another cycle, ports B, C, E, F and gates a, b, d have toggled. Clearly, the first set {B, C, E, F, b ,d} is a sub-set of the second set {B, C, E, F, a, b ,d}. Now the paths of the first set are {6, 7, 8, 9} while the paths of the second set are paths {5, 6, 7, 8, 9}. I.e., first set of paths is a subset of the second set.

IV. Graph-based Dynamic Analysis

In this section, we present our graph-based approach to dynamic analysis. We first present the basic technique, followed by two optimizations that improve performance by eliminating redundant work.

Theorem 1 implies that a set of gates that toggle during a time stamp and the nets that they drive (a toggled-set) identify the set of all toggled paths for that time stamp, i.e., the toggled-set contains all the toggled paths and no non-toggled paths. As such, we can perform dynamic timing analysis (DTA) for a design by identifying the gates that toggle at a particular time stamp, ignoring all paths that do not pass through one of the toggled gates, and performing timing analysis (STA) on the vertex-induced subgraph defined by the toggled gates using a conventional CAD tool. The following steps describe our methodology.

1) Perform gate-level simulation for a workload on the design and generate a VCD file.
2) For each time stamp in the VCD file:
   a) Read the toggled nets, mark the toggled gates (i.e., the gates driving the toggled nets) and generate a toggled-set.
   b) Run STA on the vertex-induced sub-graph defined by the toggled gates (the toggled-set).

Marking and unmarking of gates for the purposes of timing analysis is achieved in commercial CAD tools such as PrimeTime [14] using the commands `reset_path` and `set_false_path`, respectively. We first unmark all gates from timing analysis using `set_false_path` on every gate in the design and then mark the toggled gates using `reset_path`. The pseudocode for our graph-based dynamic analysis algorithm is presented in Algorithm 1. While our dynamic analysis algorithms are presented for finding the dynamic critical path of a workload, they can apply dynamic (i.e., activity-based) analysis corresponding to any kind of static analysis that can be done using a commercial STA tool such as PrimeTime [14] (e.g., statistical STA, on-chip variation analysis, crosstalk, etc.). Some of these analyses are discussed in Section VII.

The method presented above can perform dynamic analysis (such as finding the dynamic critical path distribution) over any time window of interest, from a single cycle up to full application or multiple application runs. As we will demonstrate in Section VI, our graph-based dynamic analysis techniques achieve significant performance benefits over previously-proposed path-based techniques. Nevertheless, we observe that our graph-based approach affords even further opportunities for performance improvement, based on the following two observations.

1) The set of paths corresponding to a set of toggled gates is unique (see Corollary 1). I.e., two toggled-sets containing the same set of toggled gates also contain the same unique set of toggled paths.
2) A toggled-set that includes all the gates (i.e., is a superset) of another toggled-set also includes all its paths (see Theorem 2).

Based on these observations, we propose the following optimizations.

1) Uniquification of the toggled-sets (Section IV-A).
2) Unique Non-Includible Toggled-sets (UNITs) identification (Section IV-B).

Algorithm 1 Pseudocode for Basic Graph-based DTA

Procedure *FindDynamicCriticalPath( )*
1. Read netlist and initialize PrimeTime Tcl socket interface;
2. Open VCD File;
3. **foreach** Time stamp of activity t in the VCD **do**
4. Mark all gates as not toggled; // using `set_false_path`
5. Read Toggled nets
6. **foreach** Toggled net n **do**
7. Infer Toggled gate g that drives net n
8. Mark gate g as toggled // using `reset_path`
9. **end for**
10. $S_t \leftarrow FindCriticalSlack()$ // using `report_timing`
11. **if** $S_t < S_{min}$ **then**
12. $S_{min} \leftarrow S_t$
13. **end if**
14. **end for**

A. Uniquification of Toggled-sets

Since the set of toggled paths corresponding to a toggled-set is unique, dynamic analysis only needs to be performed once per unique toggled-set. Thus, we can avoid redundant work by storing and analyzing only the unique toggled-sets, instead of the toggled-sets for every time stamp. If the same toggled-set is observed at multiple time stamps, analysis (e.g., STA) of the toggled-set need not be repeated. Algorithm 2 describes uniquification-based dynamic analysis.

While toggled-sets need not ever be repeated when a workload is executed on a processor, intuition argues that repetition of toggled-sets is likely to be common, even frequent, given that real workloads exhibit significant repetition of instruction and data use. Indeed, processors are designed with structures like caches precisely to take advantage of instruction and data reuse. Consider, for example, executing the loop in Listing 1. The *jump* instruction is executed to the same location 499 times, and the code in the loop body is executed in each of the loop's 500 iterations. The *jump* instruction, for example, excites the same paths in several stages of the processor (e.g., same decoding, same execution, etc.) each time it executes.

```
mov #500, r5  ; loop 500 times
mov #0, r4    ; initialize loop counter
loop:
 ...          ; loop body
inc r4        ; increment loop counter
cmp r5, r4    ; compare with loop limit
jl loop       ; jump if counter < limit
```

Listing 1. Assembly code for a simple loop.

Leveraging uniquification of toggled-sets to eliminate redundant work requires all unique toggled-sets to be stored before running

DTA on each set. This increases the memory footprint of our tool. However, the additional memory requirement is negligible, even for long time windows, compared to the memory requirements of path-based techniques.

Algorithm 2 Pseudocode for Uniquification-based DTA

Procedure *FindDynamicCriticalPath( )*
1. // Toggled-set Uniquification
2. Read netlist and initialize PrimeTime Tcl socket interface;
3. Open VCD File;
4. Initialize List $\mathcal{C}$ // $\mathcal{C}$ is the set of all unique toggled-sets
5. **foreach** Time stamp of activity t in the VCD **do**
6. Read Toggled nets
7. **foreach** Toggled net n **do**
8. Infer Toggled gate g that drives net n
9. $C \leftarrow insert(g)$ // C is the set of toggled gates for the current cycle
10. **end for**
11. **if** $C \notin \mathcal{C}$ **then**
12. $\mathcal{C} \leftarrow insert(C)$
13. **end if**
14. **end for**
15. // Dynamic Timing Analysis
16. **foreach** $C \in \mathcal{C}$ **do**
17. Mark all gates as not toggled; // using `set_false_path`
18. **foreach** $g \in C$ **do**
19. Mark gate g as toggled ; // using `reset_path`
20. **end for**
21. $S_t \leftarrow FindCriticalSlack()$ // using `report_timing`
22. **if** $S_t < S_{min}$ **then**
23. $S_{min} \leftarrow S_t$
24. **end if**
25. **end for**

B. Unique Non-Includible Toggled-sets (UNITs) Identification

In this section, we present another optimization that can improve the performance of DTA. When performing DTA for unique toggled-sets, it is not necessary to analyze any toggled-set that is a subset of another toggled-set. This is because, as stated in Theorem 2, if a gate-set A is a subset of another gate-set B, then the paths of A are also a subset of the paths of B. Thus, analyzing B will inherently involve complete analysis of A.

For an example of how UNITs may improve the efficiency of DTA, consider again the code in Listing 1. The paths exercised during an increment of r4 from 127 to 128 (0b1111111 + 1) are a superset of the paths covered during an increment from 31 to 32 (0b11111 + 1), since the former increment executes the same instruction but toggles more bits than the latter. Identification of UNITs can reduce the execution time and memory requirements of our DTA tool (see Section VI). Algorithm 3 describes UNITs-based dynamic analysis.

During UNITs identification, we only store the *Non-Includible Toggled-sets*, that is, the toggled-sets that are not subsets of any other toggled-sets. We use a data structure called the SetTrie [15] to perform fast subset and superset operations. We briefly explain the data structure below. Complete details can be found in [15].

1) SetTrie: A SetTrie is a data structure that is similar to the Trie data structure used for text searching. The Trie is designed for efficient substring searches while the SetTrie is designed for efficient subset and superset searches. Unlike Trie, SetTrie requires the elements of the universal set to be indexed. Element indices are inserted into the SetTrie such that a traversal path from the root to a leaf corresponds to a set of elements that is stored in the SetTrie. For example, the SetTrie in Figure 2 stores the following sets.

1) {1, 3, 7}
2) {1, 3, 8}
3) {1, 2, 5}
4) {3, 7, 9}
5) {3, 8, 9}

The original SetTrie [15] allows for any internal node to also act as the last element in the set, by using a flag for each node. We do not need this feature, since UNITs require that we only insert a new set if a superset does not already exist in the SetTrie.

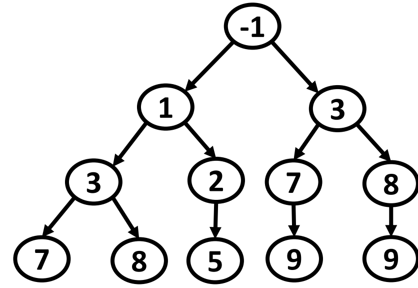

Fig. 2. A SetTrie data structure used for fast superset lookups.

To insert a set, we first check if there already exists a superset of the set being inserted. If this is the case, we do not perform insertion. If the check reveals no superset, we traverse down the tree while the path of traversal matches exactly with the set being inserted and create new nodes after the first point of deviation to accomodate the set being inserted. The exact algorithms for *insert* and *existsSuperset* can be found in [15].

Note that after a new set has been inserted, we would also like to delete all the subsets of the new set. However, due to the exponential complexity of the *getAllSubsets* function [15], we use a different strategy. If a set is inserted successfully into the SetTrie, we store a copy of the set in a separate list. Once all the sets have been inserted (VCD file parsing has been completed) we check if there exists a *proper* superset for each toggled-set stored in the separate list. If so, we delete the toggled-set from the list of UNITs. For this purpose, we enhance the *existsSuperset* function [15] to *existsProperSuperset*. Note that we do not iterate over the entire list of toggled-sets again. The number of toggled-sets remaining after initial insertion is less than or equal to the number of unique toggled-sets (see Section IV-A) and hence is significantly less than the number of parsed time stamps (see Table III).

After removal of all sets that have a proper superset, we are left with a set of Unique Non-Includible Toggled-sets (UNITs). These UNITs cover all the toggled paths of a workload. There may still exist redundancy between the UNITs, i.e., these sets may have a significant number of paths in common. However, elimination of this redundancy would require a path-based analysis which can be resource expensive, both in terms of time and memory.

C. Toggle Rate of Paths

Path-based analysis and optimization techniques in previous works also determine and utilize the toggle rates of paths for power and performance optimizations [1], [2], [3], [4], [5], [6]. Since these techniques rely on path-based analysis, they enumerate paths and count the number of times each path has been toggled.

Although we do not enumerate paths due to the high overheads involved, our technique still allows for an efficient approach to finding path toggle rates. Namely, the toggle rate of a path can be found by summing the toggle rates of all the unique toggled-sets the path belongs to. A path belongs to a toggled-set if the set of gates in the path is a subset of the set of gates in the toggled-set. Thus, the *getAllSupersets* function of a SetTrie [15] containing all the unique toggled-sets can be used to determine which unique toggled-sets a path belongs to. The toggle rate of a unique toggled-set can easily be determined by maintaining a counter for each unique toggled-set during VCD parsing. Each time a toggled-set is encountered at a time stamp, the counter for the set is incremented, indicating that all the paths in the toggled-set have toggled at that time stamp.

A tradeoff exists between uniquification of toggled-sets and UNITs. UNITs produces (sometimes significantly) fewer toggled-sets

to analyze than uniquification and can perform analysis faster (see Section VI). However, UNITs discards information about the subsets that have been merged into a superset, and thus it is not possible to determine path toggle rates from UNITs-based analysis. I.e., a UNIT may encompass the information for more than one unique toggled-set, so determining the toggle rate of each unique toggled-set is not possible for a UNIT.

One way to get the benefits of both methods (activity analysis possible with uniquification and increased efficiency of UNITs) is to maintain unique toggled-sets information for activity analysis and perform subsetting on the unique toggled-sets and use UNITs-based timing analysis on the dynamic critical paths. Note that we do not need to enumerate and analyze all the toggled paths as in path-based analysis. E.g., we can focus only on the critical/near-critical paths reported by our DTA methodology.

Algorithm 3 Pseudocode for UNITs-based DTA

Procedure *FindDynamicCriticalPath( )*
1. // UNIT Identification
2. Read netlist and initialize PrimeTime Tcl socket interface;
3. Open VCD File;
4. Initialize SetTrie $\mathcal{C}_{st}$
5. Initialize List $\mathcal{C}$
6. **foreach** Time stamp of activity t in the VCD **do**
7. Read Toggled nets
8. **foreach** Toggled net n **do**
9. Infer Toggled gate g that drives net n
10. $C \leftarrow g$ // C is the set of toggled nets for the current cycle
11. **end for**
12. **if** $\neg existsSuperSet(\mathcal{C}_{st}, C)$ **then**
13. $\mathcal{C}_{st} \leftarrow insert(C)$
14. $\mathcal{C} \leftarrow insert(C)$
15. **end if**
16. **end for**
17. **foreach** $C \in \mathcal{C}$ **do**
18. **if** $existsProperSuperSet(\mathcal{C}_{st}, C)$ **then**
19. $\mathcal{C} \leftarrow delete(C)$
20. **end if**
21. **end for**
22. // Dynamic Timing Analysis
23. **foreach** $C \in \mathcal{C}$ **do**
24. Mark all gates as not toggled; // using `set_false_path`
25. **foreach** $g \in C$ **do**
26. Mark gate g as toggled ; // using `reset_path`
27. **end for**
28. $S_t \leftarrow FindCriticalSlack()$ // using `report_timing`
29. **if** $S_t < S_{min}$ **then**
30. $S_{min} \leftarrow S_t$
31. **end if**
32. **end for**

V. Methodology

We verify our techniques with experiments on a silicon-proven processor – openMSP430 [16]. Designs are synthesized, placed, and routed with TSMC 65GP library (65nm), using Synopsys Design Compiler [17] and Cadence EDI System [18] assuming worst-case operating conditions. Gate-level simulations are performed by running full benchmark applications from Table I on the placed and routed processor using Synopsys VCS [19]. Activity information is read from the VCD file generated from gate-level simulation. Timing analysis is performed with Synopsys PrimeTime [14]. Experiments were performed on a server housing two Intel Xeon E5-2640 Processors with 8-cores each, 2 GHz operating frequency, and 64 GB RAM. We implemented our algorithms in C++. For comparison against path-based DTA, we implemented the path-based tool from [2]. Benchmarks dhrystone_4mcu, dhrystone_v2.1 and coremark_v1.0 are available in [16]. All other benchmarks are taken from [20].

VI. Results and Analysis

To illustrate the benefits of our graph-based analysis over path-based analysis, we compare the computation time (in seconds) for each technique to perform dynamic timing analysis of the processor. This involves running a benchmark on the processor, characterizing all the toggled paths in the design, and finding the critical timing path among the toggled paths.

TABLE I. Benchmark Descriptions

mult	Integer Multiplication
tea8	8-bit Tiny Encryption Algorithm
binSearch	Binary Search
rle	Run-Length Encoding Algorithm
intAVG	Integer Average
inSort	Insertion Sort
tHold	Threshold Cross Detection
div	Integer Division and Outputing
intFilt	FIR Lowpass Integer Filter
dhrystone_v2.1	Dhrystone Benchmark
dhrystone_4mcu	Dhrystone Benchmark for MCUs
coremark_v1.0	Coremark Benchmark

TABLE II. Computation times (seconds) for path-based DTA and the proposed techniques at different length of time windows (cycles)

time window (cycles)	5	25	50	125	250	375	500
Path-based DTA	135	1455	1516	1527	5054	5105	5326
Basic DTA	8	21	38	81	159	248	341
Uniquified DTA	7	13	15	17	41	42	44
UNITs DTA	7	13	15	16	37	37	39

When we attempted to run the path-based tool [2] for the full processor and a benchmark with relatively low activity (div), we observed that after two hours of computation our server (with 64 GB of RAM) ran out of memory and was only able to analyze paths for a time window of 25 cycles in the VCD file. For a benchmark with higher activity (coremark_v1.0) and thus more toggled paths, the path-based tool was not even able complete analysis for one cycle before running out of memory.

Due to the high memory and computation time requirements of path-based analysis, we could only perform a comparison for a processor module (not the full processor). Note that previous works that use path-based analysis are likewise limited to analyzing only small modules [1], [2], [3], [4], [5], [6], [8]. Table II compares the runtime of path-based DTA for the execution unit of openMSP430 and the div benchmark against our three approaches described in Section IV. Figure 3 shows the performance data for our approaches normalized to that of path-based analysis. Table II and Figure 3 show data for different time windows of execution (in cycles), demonstrating that even for a single module the performance benefit of graph-based analysis is significant (up to 136.6X, 105.6X on avg.) and increases for larger time windows. Note that even for single-module analysis over these short time windows the computation time of path-based analysis quickly becomes unreasonable. While it can be seen that UNITs is faster than uniquification-based DTA, the next set of results makes a more convincing case for UNITs.

The next set of results compares the performance benefits offered

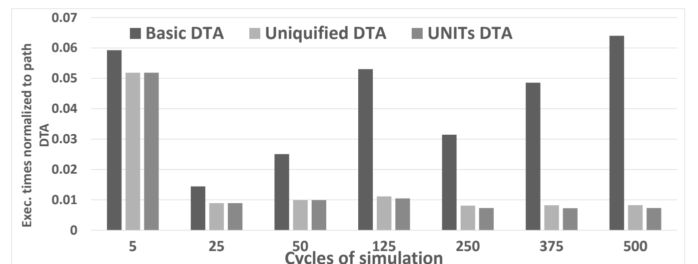

Fig. 3. Execution times for graph-based DTA normalized to path-based DTA.

TABLE III. NUMBER OF TOGGLED-SETS IDENTIFIED FOR ANALYSIS BY EACH DTA APPROACH.

Benchmarks	Basic DTA toggled-sets	Unique toggled-sets	UNITs
mult	147	74	73
tea8	4191	1704	1579
binSearch	4723	925	764
rle	5848	2318	1372
intAVG	12308	4704	1849
inSort	28813	5762	3333
tHold	28870	10016	5508
div	68801	6594	3387
intFilt	222495	8559	6547
dhrystone_4mcu	332977	12773	2908
dhrystone_v2.1	478429	4703	2818
coremark_v1.0	980930	180695	108379

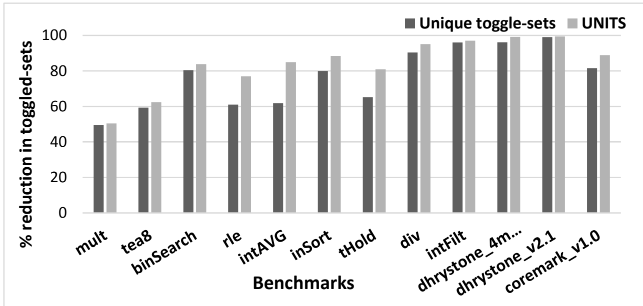

Fig. 4. Percentage reduction in the number of toggled-sets due to Uniquification and UNITs.

by our two DTA optimizations for full processor analysis and full application execution of the benchmarks in Table I. I.e., the analysis time window spans the full execution of the benchmark on the full processor. Note that this full level of analysis, which would be expected of a commercial CAD tool, is enabled by our graph-based analysis approach and is not possible for path-based analysis. Execution time profiling (e.g., gprof) revealed that the time taken to unmark all gates, mark all toggled gates, and report the critical timing path is approximately the same for any given toggled-set, and on average, these steps consume over 90% of total analysis time. Thus, the primary performance benefit of our optimizations comes from reduction of the number of toggled-sets that must be analyzed. Table III shows the number of toggled-sets identified for analysis by basic graph-based DTA (Algorithm 1), uniquification-based DTA (Algorithm 2), and UNITs-based DTA (Algorithm 3). Figure 4 shows the percentage reduction in the number of toggled sets for the two optimized DTA techniques, relative to basic graph-based DTA.

Table III and Figure 4 demonstrate significant reduction in toggled-sets for both uniquification and UNITs. Uniquification reduces toggled-sets by up to 99.0%, 76.6% on average, and UNITs reduce toggled-sets by up to 99.4%, 83.9% on average. While all benchmarks benefit from UNITs over uniquification, benchmarks such as div, rle, intAVG, tHold, dhrystone_v2.1, dhyrstone_4mcu and coremark_v1.0 benefit significantly, showing 50.35% average reduction for UNITs relative to uniquification. Note that for the larger benchmarks (dhrystone_v2.1, dhrystone_4mcu and coremark_v1.0) the benefit of UNITs over uniquification is hard to distinguish in Figure 4 (percent reduction metric), since both approaches result in very significant reduction of toggled-sets compared to basic graph-based DTA. Table III, however, provides the absolute results which show significant differences in the number of toggled sets.

VII. APPLICABILITY TO ADVANCED TIMING ANALYSIS TECHNIQUES

Since our dynamic analysis techniques are based on traditional timing analysis methodologies, they can easily be extended to advanced timing analysis techniques such as variation-aware analysis and multiple input switching (MIS). Below, we list some advanced timing techniques that can easily be incorporated in our graph-based dynamic analysis approach. Note that other dynamic analysis approaches [1], [2], [3], [4], [5], [6] have not considered the advanced timing analysis techniques discussed below; however, we mention them here for completeness and to describe how they can be easily integrated into our approach. We also discuss how to handle scenarios where the assumptions in Section III are not valid (e.g., compound cells and false paths due to controlling inputs).

Removing graph-based pessimism: Since our technique leverages the benefits of graph-based STA for dynamic analysis, it inherently incorporates the pessimism of graph-based analysis. This is a well-known issue in traditional STA which has been addressed by using path-based analysis for the critical paths reported by graph-based STA to remove pessimism. The same approach can be applied on the UNITs of a benchmark to accurately report the slack of the dynamic critical paths. Also, such analysis can be restricted to the UNITs with near-critical slack (only 8.09% of UNITS, on average, where near-critical means:
$slack_{UNIT} \leq slack_{dynamic\ critical\ path} + 10\% \times clock\ period$), so the cost of path-based analysis is significantly reduced by only analyzing the near-critical exercised paths, rather than all exercised paths (in the case of previous dynamic analysis techniques). Since they rely on path-based analysis, previous works [1], [2], [3], [4], [5], [6] do not suffer from graph-based pessimism; however, they would unnecessarily perform timing analysis on a large number of non-critical paths over a large number of redundant toggled-sets.

Compound cells: Some cells provided by a cell library are compound cells, such as a 2:1 MUX. For compound cells, it may be the case that a path through the cell can be considered as a false path, even though all gates on the path toggled. For example, if both the inputs of a MUX toggle, the path through one input can be marked as false, based on the value of the select pin. Similarly, the input to a tri-state buffer is marked as false if its enable pin is OFF. This functionality is easily incorporated in our analysis by tracking toggled pins rather than toggled gates. The rest of the analysis remains the same. Toggled pins completely and exclusively characterize all the toggled paths, and our optimizations are still valid on the new toggled-sets that consider toggled pins instead of gates.

Rise and fall toggled-sets: Our results in Section VI were generated by considering both rise and fall transitions simply as toggles, rather than differentiating the two. Note that previous works on dynamic analysis also did not differentiate between rising and falling transitions [1], [2], [3], [4], [5], [6]. However, in some circumstances, differentiating rising and falling toggles could provide more accurate timing analysis. For completeness, we re-evaluated our results in Table III by differentiating rising and falling sets for pins, as well as false path marking for compound cells, and observed that the results only change by 3%, at the most, compared to basic DTA for any benchmark.

Multiple input switching: If more than one input of a gate switches at the same time, the delay of the gate can be different than in the single input switching scenario traditionally assumed for STA. Our graph-based analysis can easily perform more accurate timing analysis that accounts for multiple input switching, since we can track the value of each pin in the design from the VCD file and determine when multiple inputs of the same gate toggle with similar arrival times/windows.

False paths due to controlling inputs: If an input to a gate toggled to a controlling value, any other inputs that toggled to a non-controlling value can be marked as false. If multiple inputs of a toggled gate toggled to a controlling value in the same cycle, the

slower transitioning path(s) can be considered false path(s). This is because the fast path toggles the gate's output first, precluding the effect of any slower path's toggle. The arrival times of the input pins of the toggled gate can be used to identify which controlling input arrives first, and the path(s) through the other pin(s) can be marked as false. Since the number of gates with multiple input switching is small, the overhead of checking the above conditions is negligible. Note that analysis of controlling inputs would likely have significantly higher overhead for path-based techniques, since the same gate would be analyzed multiple times (once per toggled path it is in). Since variations may affect which input arrives first to a gate, we did not mark false paths due to fast-arriving controlling inputs for our analysis.

Statistical Static Timing Analysis, Multi-Mode, Multi-Corner, and On-Chip Variation Analyses: Since SSTA can be graph-based and also be applied incrementally [21], [22] our method, which is based on pruning a design then applying STA, can easily incorporate SSTA. On-chip variation analyses such as Parametric On-chip Variation analysis [23] are inherited from SSTA. Having a graph-based and path-based version these analyses are easily incorporated into our methodology. MMMC techniques involve re-running timing analysis for various modes at various corners, which can easily be performed with our approach.

Crosstalk Analysis: Crosstalk analysis can easily be included in our methodology, since transitions (rise/fall) and values on nets and pins for crosstalk analysis can be excluded/included using commands such as `set_si_delay_analysis` and `set_case_analysis` provided by PrimeTime [14].

VIII. Conclusion

In this paper, we show that path-based dynamic analysis tools used by existing BTWC techniques to analyze timing and activity information do not scale to larger designs or analysis time windows. We propose a novel graph-based dynamic analysis methodology that is not only scalable but also significantly faster than previous tools. Also, our methodology is easily integrated with industry-standard CAD tools. We further improve our methodology with two optimizations – uniquification of toggled-sets and Unique Non-Includible Toggled-sets (UNITs) and discuss their trade-offs. Our results demonstrate 105.6x speedup compared to path-based DTA and 93.8%, 96.9% average reduction in analyzed toggled-sets for uniquification and UNITs, respectively.

Acknowledgment

This research has been supported by the National Science Foundation and the Semiconductor Research Consortium. The authors would like to thank Pranav Kumar Cherupalli and Sri Harsha Vadlamani for discussions on timing analysis techniques.

References

[1] Andrew B Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Recovery-driven design: a power minimization methodology for error-tolerant processor modules. In *Proceedings of the 47th Design Automation Conference*, pages 825–830. ACM, 2010.

[2] Andrew B Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Recovery-driven design: Exploiting error resilience in design of energy-efficient processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(3):404–417, 2012.

[3] Andrew B Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 825–831. IEEE, 2010.

[4] John Sartori and Rakesh Kumar. Exploiting timing error resilience in processor architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):89, 2013.

[5] John Sartori and Rakesh Kumar. Compiling for energy efficiency on timing speculative processors. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1297–1304. IEEE, 2012.

[6] Brian Greskamp, Lu Wan, Ulya R Karpuzcu, Jeffrey J Cook, Josep Torrellas, Deming Chen, and Craig Zilles. Blueshift: Designing processors for timing speculation from the ground up. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 213–224. IEEE, 2009.

[7] Smruti R Sarangi, Brian Greskamp, Radu Teodorescu, Jun Nakano, Abhishek Tiwari, and Josep Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. *Semiconductor Manufacturing, IEEE Transactions on*, 21(1):3–13, 2008.

[8] Smruti Sarangi, Brian Greskamp, Abhishek Tiwari, and Josep Torrellas. Eval: Utilizing processors with variation-induced timing errors. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 423–434. IEEE, 2008.

[9] A Hakan Baba and Subhasish Mitra. Testing for transistor aging. In *VLSI Test Symposium, 2009. VTS'09. 27th IEEE*, pages 215–220. IEEE, 2009.

[10] Jing Xin and Russ Joseph. Identifying and predicting timing-critical instructions to boost timing speculation. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 128–139. ACM, 2011.

[11] Abbas Rahimi, Luca Benini, and R Gupta. Application-adaptive guardbanding to mitigate static and dynamic variability. *Computers, IEEE Transactions on*, 63(9), Sept 2014.

[12] Koushik Chakraborty, Brennan Cozzens, Sanghamitra Roy, and Dean M Ancajas. Efficiently tolerating timing violations in pipelined microprocessors. In *Proceedings of the 50th Annual Design Automation Conference*, page 102. ACM, 2013.

[13] Omid Assare and Rajesh Gupta. Timing analysis of erroneous systems. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, page 7. ACM, 2014.

[14] Synopsys. *PrimeTime User Guide*.

[15] Iztok Savnik. Index data structure for fast subset and superset queries. In Alfredo Cuzzocrea, Christian Kittl, DimitrisE. Simos, Edgar Weippl, and Lida Xu, editors, *Availability, Reliability, and Security in Information Systems and HCI*, volume 8127 of *Lecture Notes in Computer Science*, pages 134–148. Springer Berlin Heidelberg, 2013.

[16] O Girard. Openmsp430 project. *available at opencores.org*, 2013.

[17] Synopsys. *Design Compiler User Guide*.

[18] Cadence. *Encounter Digital Implementation User Guide*.

[19] Synopsys. *VCS/VCSi User Guide*.

[20] Bo Zhai, Sanjay Pant, Leyla Nazhandali, Scott Hanson, Javin Olson, Anna Reeves, Michael Minuth, Ryan Helfand, Todd Austin, Dennis Sylvester, et al. Energy-efficient subthreshold processor design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(8):1127–1137, 2009.

[21] Chandramouli Visweswariah, Kaushik Ravindran, Kerim Kalafala, Steven G Walker, and Sambasivan Narayan. First-order incremental block-based statistical timing analysis. In *Proceedings of the 41st annual Design Automation Conference*, pages 331–336. ACM, 2004.

[22] Jin Wook Kim, Wook Kim, Hyoun Soo Park, and Young Hwan Kim. Incremental statistical static timing analysis with gate timing yield emphasis. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 1016–1019. IEEE, 2008.

[23] Ayhan Mutlu, Jiayong Le, Ruben Molina, and Mustafa Celik. A parametric approach for handling local variation effects in timing analysis. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 126–129. IEEE, 2009.