Abstract—As the overheads for ensuring the correctness of electronic designs continue to increase with continued technology scaling and increased variability, better-than-worst-case (BTWC) design has gained significant attention. Many BTWC design techniques utilize dynamic timing and activity information for design analysis and optimization. These techniques rely on path-based analysis that enumerates the exercised paths in a design as targets for analysis and optimization. However, path-based dynamic analysis techniques are not scalable and cannot be used to analyze full processors and full applications. On the other hand, graph-based techniques like those that form the foundational building blocks of electronic design automation tools are scalable and can efficiently analyze large designs. In this paper, we extend graph-based analysis to provide the fundamental dynamic analysis tools necessary for BTWC design, analysis, and optimization. Specifically, we present scalable graph-based techniques to report the N-worst exercised paths in a design for three metrics – timing criticality (slack), activity (toggle count), and activity subject to delay constraints. Compared to existing path-based techniques, our scalable dynamic analysis techniques improve average performance by 977×, 163×, and 113×, respectively, and enable scalable analysis for a full processor design running full applications.

I. INTRODUCTION

As challenges in technology scaling have resulted in increasing static and dynamic variations, accompanied by increasingly restrictive design guardbands that ensure correctness even under worst-case conditions, better-than-worst-case (BTWC) design has emerged as a way to improve energy efficiency under average-case conditions by relaxing conservative design constraints and either tolerating or correcting any resulting errors [1], [2], [3], [4], [5], [6], [7].

BTWC design techniques rely on error tolerance or correction mechanisms to handle errors when worst-case conditions do occur. This allows a processor to be optimized for and operated at a BTWC operating point, potentially resulting in significant energy savings with respect to a design that has been guardedbanded for worst-case conditions. 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 characterize how a design is exercised by a set of target applications. Dynamic information describes which parts of a design are most likely to be exercised or produce errors under BTWC conditions. Such information allows a designer to optimize for BTWC conditions and improve design efficiency for a common case where errors are allowed.

Many prior BTWC design and optimization techniques have used path-based algorithms that enumerate the exercised paths in a BTWC design as targets for analysis and optimization [2], [3], [4], [5], [6], [7]. However, due to the extreme number of paths in modern electronic designs [8], path-based analysis and optimization is inefficient and in most cases infeasible, even for small designs. Due to the large computation and memory requirements of path-based analysis and optimization, existing dynamic analysis and optimization techniques have been forced to represent a full design using only a small sample set of small design modules and to represent a full application using only small code snippets covering a small execution time window. This has limited their applicability in modern semiconductor designs, which can often contain many thousands of gates, and many orders of magnitude more paths [8].

Recent work has proposed a graph-based technique to characterize the set of paths exercised by a workload running on a design, demonstrating that graph-based techniques for reporting the longest exercised path in a design are significantly faster than path-based techniques [9]. While the authors of [9] demonstrate a scalable technique for characterizing the longest exercised path in a design (i.e., the dynamic critical path), BTWC design often involves analysis and optimization of more than one dynamic critical path, as well as paths that are both timing-critical and/or highly-exercised [2], [3], [6], [1], [4]. For example, identification of a single dynamic critical path can identify the threshold at which the BTWC operating region begins, but determination of how the design behaves once in the BTWC region and how to optimize it for operation in the BTWC region requires identification, analysis, and optimization of multiple exercised paths, in terms of both timing criticality and activity. As such, a BTWC designer would be interested in having a complete set of scalable dynamic analysis tools to characterize the timing and activity of exercised paths in a BTWC design to report the following information.

1. The N-worst exercised paths in terms of timing criticality: A path that fails timing constraints in a BTWC design can only generate an error if it is exercised (toggled). When guardbands are relaxed in a BTWC design, the first paths to cause errors are the most timing-critical paths. Analysis of the N-worst exercised paths in terms of timing criticality tells a BTWC designer how much guardbands can be relaxed before a design will produce errors and which paths will generate errors at a particular BTWC operating point.¹

2. The N-worst exercised paths in terms of toggle count (activity): A path that fails timing constraints at a BTWC operating point causes an error every time it toggles. The more a path is exercised by applications running on a processor design, the more the paths the error can cause when it fails to meet timing constraints. Analysis of the N-worst exercised paths in terms of toggle count characterizes the paths that have the potential to cause the most errors in a BTWC design.

3. The N-worst exercised paths in terms of toggle count (activity) within a slack range: In a BTWC design, the objective is often to minimize energy while keeping the error rate below a certain acceptable threshold that can be effectively tolerated or corrected. Identifying the activity of paths within different timing slack ranges shows how the error rate of a design changes as guardbands are relaxed and can also be used to produce a dynamic timing slack distribution [3] that characterizes the error behavior of a BTWC design at different operating points. This allows a designer to optimize the design for minimum energy while bounding the error rate of the design. It also allows the design to be dynamically tuned for minimum energy when executing different applications that can tolerate different amounts of error or using different error tolerance techniques with different optimal error rates.

¹Guardband relaxation in a BTWC design can take several forms; the most common are voltage and frequency over-scaling.
TABLE I: The scalable dynamic analysis techniques proposed in this paper cover the set of capabilities needed to perform BTWC design, analysis, and optimization. BTWC techniques proposed in prior work rely on one or more of the dynamic analysis techniques and thus would benefit from availability of scalable dynamic analysis algorithms.

<table>
<thead>
<tr>
<th>Type of Dynamic Analysis</th>
<th>Usage in Prior Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-worst exercised paths in terms of timing criticality</td>
<td>[10], [11], [12], [2], [2], [1], [6], [3], [4], [9]</td>
</tr>
<tr>
<td>N-worst exercised paths in terms of activity</td>
<td>[2], [1], [3], [4]</td>
</tr>
<tr>
<td>N-worst exercised paths in terms of activity within a slack range</td>
<td>[7], [13], [14], [11], [12], [5], [15]</td>
</tr>
</tbody>
</table>

The dynamic analysis algorithms described above encompass the fundamental techniques needed for BTWC design, analysis, and optimization. Table I lists several prior works on BTWC design, analysis, and optimization that rely on each type of dynamic analysis. Scalable dynamic analysis techniques would enable these and other BTWC techniques to be applied to full processor designs and full applications.

In this paper, we present scalable graph-based algorithms that perform the dynamic analysis routines described above. Our techniques are fully-automated and can be integrated into existing electronic design automation (EDA) tools to analyze full processors and full applications. Furthermore, our techniques significantly reduce analysis time and memory requirements with respect to path-based analysis, which must enumerate all exercised paths in order to identify the N-worst exercised paths for a given type of analysis. Such an approach incurs huge overheads in terms of runtime and memory usage, even when $N = 1$. Our scalable algorithms, on the other hand, can be used iteratively, where each additional iteration incurs only a small incremental cost to identify the $N^{th}$ worst exercised path in a design. As such, the overhead of our techniques scales linearly with $N$.

Our paper makes the following contributions:

- We present novel graph-based dynamic analysis algorithms to compute the N-worst exercised paths for the following metrics: (1) timing slack, (2) toggle count / rate, (3) toggle count within a slack / delay range. These algorithms comprise a complete set of analyses needed for BTWC design, analysis, and optimization. To the best of our knowledge, these are the first graph-based dynamic analysis techniques that perform N-worst analysis for exercised paths.
- We show that our techniques are scalable and can analyze large designs. Whereas prior techniques relied on analysis of a small subset of design modules and small application snippets to represent full designs and applications, our techniques can analyze full processors and full applications. Furthermore, compared to existing path-based techniques, our techniques reduce analysis time significantly, even for large values of $N$. We demonstrate speedups of up to $436 \times$, and average speedups for our algorithms are $977 \times$, $163 \times$, and $113 \times$, respectively, with respect to path-based dynamic analysis. The performance benefits of our techniques increase as processor or application complexity increase.

II. RELATED WORK

Most previous works that perform dynamic timing and activity analysis for BTWC design and optimization use path-based tools. Blueshift [6] is a BTWC design methodology that uses path activity information to adjust path timing constraints and reduce the error rate of a frequency-overscaled BTWC design. Eval [11] proposes micro-architectural techniques that improve the power and performance of a processor by allowing variation-induced errors. Eval relies on the VATS model [12], which computes the dynamic slack distribution of a processor for a given workload. Power-aware slack redistribution [5] uses path timing criticality and toggle rate information to apply path-based optimizations that improve power and area efficiency in a voltage-overscaled design. Recovery-driven design [1], [2] optimizes a BTWC design for a specific target error rate. The approach uses path-based activity and timing analysis to identify the paths in a design that cause the most and least errors. Based on this characterization, recovery-driven design applies path-based optimizations to each path, such that error-prone paths are afforded more timing slack and paths that rarely cause errors are afforded less timing slack. Work on optimizing the processor microarchitecture of a BTWC design [4] leverages dynamic timing and activity analysis to guide architectural optimizations that manipulate the timing error rate behavior of a design and increase the effectiveness of timing speculation. Compiler-based software optimizations have also been applied to improve the energy efficiency of timing speculative processors based on path-based dynamic timing and activity information [5]. Dynamic timing and activity information have also been used in non-timing-speculative designs to identify a BTWC minimum voltage or maximum frequency at which a specific application can safely execute on a processor without causing errors [10].

The BTWC design techniques described above rely on path-based timing and activity analysis, and many of the techniques also perform path-based optimizations. These techniques involve enumeration of the exercised paths in a design and are not scalable, due to the extreme number of paths in electronic designs [8]. 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 or applications, module sampling methodologies ignore paths between modules and those that cross module boundaries.

Since path-based techniques are not scalable, some works employ alternative techniques that either produce inexact results or perform redundant work. One branch of work estimates error rate by performing multiple delay-annotated gate-level simulations at different (voltage, frequency) operating points [7], [13], [14]. In contrast, our graph-based technique captures the dynamic timing and activity information of an application, processor pair in a single gate-level simulation, and error rates at different operating points can be computed with little additional effort by recomputing gate delays and performing STA. One work proposes a clustered timing model to capture the dynamic delay distribution of a processor [15]. The approach requires manual analysis of a design’s architecture and produces inexact results because of architectural approximations. In contrast, our graph-based algorithms, like other EDA routines, are automated, architecture-independent, and do not introduce artificial approximations that degrade accuracy.

Recent work on graph-based dynamic analysis [9] proposes a technique for identifying the longest exercised path in a design (i.e., the dynamic critical path) without enumeration of all exercised paths. However, the methods presented in [9] cannot be used to perform the dynamic timing and activity analyses discussed in Section I – the N-worst dynamic analysis algorithms that would provide the essential information needed for many BTWC design, analysis, and optimization techniques. In this paper, we expand upon the foundation of graph-based dynamic analysis to create a complete set of scalable EDA routines needed for BTWC design.
III. Preliminaries

Before explaining our dynamic analysis algorithms, we define some constructs used in their derivation. Although the constructs can be applied to graph theory in general, we apply them in the specific context of a gate-level netlist for a digital design.

\[ G \rightarrow \text{Graph of the design containing gates (vertices) and nets (edges), i.e., the gate-level netlist of the design.} \]

\[ p(G) \rightarrow \text{Set of all paths in the graph } G. \]

\[ g(G) \rightarrow \text{Set of all gates (vertices) in the graph } G. \text{ (Note that we use the terms gate and vertex interchangeably in this paper.)} \]

\[ f(G) \rightarrow \text{Set of path endpoints (flip-flops, clock gates, etc.) in the design represented by graph } G. \text{ Note that} f(G) \text{is a subset of } g(G), \text{i.e., we consider all path endpoints as gates.} \]

\[ p_i \rightarrow \text{A particular path.} \]

\[ g_i \rightarrow \text{A particular gate.} \]

Definition 1. Path: A set of gates \{g_a, g_b, ..., g_n\} of a graph G is 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(G).

Definition 2. Toggled gate: A gate is toggled in a particular cycle when the net driven by the gate changes values in that cycle.

Definition 3. Toggled Path: A path is toggled in a particular cycle if all the gates in the path toggle 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 does not toggle 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.

Definition 7. Unique Toggled-set (UT): Given a set of toggled-sets representing per-cycle active gates from one or more gate-level simulations on the same design, a toggled-set that remains after unifying the set of toggled-sets is called a unique toggled-set (UT) [9].

Definition 8. Unique Non-Includible Toggled-set (UNIT): Given a set of toggled-sets representing per-cycle active gates from one or more gate-level simulations on the same design, a unique toggled-set that is not a subset of any other toggled-set is called a unique non-includible toggled-set (UNIT) [9].

Definition 9. UT / UNIT Membership Array: The UT / UNIT membership array for a gate is a bit vector that describes which UTs / UNITS the gate belongs to. The length of the bit vector equals the number of UTs / UNITS extracted from the gate-level simulation, and each bit position represents one UT / UNIT. A ‘1’ in a position indicates that the gate belongs to that UT / UNIT.

The set of toggled sets corresponding to each cycle of execution for an application characterizes the activity of all gates and paths in a design for that application. UTs and UNITS characterize the activity of an application more efficiently, since they reduce the number of toggled-sets necessary to characterize all unique activity generated by the application executing on the design [9].

IV. N-worst Algorithms for Dynamic Timing and Activity Analysis

In this section, we present scalable graph-based algorithms to compute the N-worst paths exercised by a workload executing on a design for the three metrics described in Section I. These automated algorithms can be incorporated into EDA tools for BTWC design, analysis, and optimization.

A. N-worst exercised timing-critical paths

Algorithm 1 computes the N-worst exercised paths in a design in terms of timing criticality. In other words, among all the paths in a design that are exercised by an application, Algorithm 1 computes the N longest paths in decreasing order of delay (increasing slack). In a BTWC design, the first paths to cause errors when guardbands are relaxed (e.g., though voltage or frequency overscaling) are the exercised paths with the longest delay (least timing slack). All exercised paths with negative timing slack at a given operating point (voltage, frequency) produce errors. This algorithm can be used to identify the paths that fail first in a BTWC, as well as the point at which paths begin to fail.

Algorithm 1 begins by creating data structures that characterize the activity of gates in the design, using an activity file captured during simulation of an application (e.g., a VCD). First, a toggled-set is generated for each time stamp in the activity file, characterizing the activity observed at that time stamp. Then, uniquification and subsetting of the toggled-sets produces the UNITS for the application, and each UNIT is assigned a unique index. The indexed list of UNITS is used to generate a UNIT membership array for each gate (or pin or net) in the design. Next, two data structures are created to explore along exercised paths in the design to identify the N-worst paths. One structure is a min-heap of path segments that orders the path segments based on their timing slack. Secondly, a list is created to store fully explored paths in order of decreasing timing criticality as they are discovered. With these data structures initialized, Algorithm 1 begins to iteratively identify the N-worst exercised critical paths, as follows.

First, all design endpoints (such as flip-flops and output ports) are inserted into the heap. Each exercised endpoint represents the terminus of a potential exercised path in the design, since the smallest non-empty path segment is a single gate or port. The key value for each endpoint (path segment) is the minimum slack of any path through that endpoint (path segment). At this point, the algorithm begins iterating on the heap. In each iteration, the top path segment, i.e., the path segment with the least timing slack, is popped from the heap. If the path segment is a full path, it is appended to the list of explored paths as the next worst path in order of timing criticality. If the path segment is not a full path, each fanin gate (or pin or net) of the path at the extendible end of the path segment (the end furthest from the endpoint) is added to the path segment to produce a new path segment. The UNIT membership array of each new path segment produced is computed by performing a bitwise AND of the UNIT membership arrays of the original path segment and the newly added gate. This operation works because the UNIT membership array for the new path segment is only active in cycles when the original path segment and the newly added gate are both active. If, the sum of the values of the UNIT membership array is non-zero, the new path segment belongs to at least one UNIT. If a path segment belongs to at least one UNIT, it was exercised by the application in at least one cycle, so the path segment is pushed onto the heap for future analysis, using the minimum slack of any path through the path segment as its sorting key in the heap. The algorithm keeps iterating over the path segments in the heap in this way until the number of explored paths equals N.

Figure 1 illustrates an iteration of computation for Algorithm 1, in which path segment P1 – the segment through which the minimum-slack path passes – is popped from the top of the heap and expanded through its fanin gates (G5 and G6). The bitwise AND of G5 and

3While we describe our analysis routines in terms of gates, the analysis is also equally valid for pins or nets.
AlGORITHM 1 Pseudocode to report the N-worst exercised paths sorted by timing criticality

1. \( U \leftarrow \text{Generate\_UNITs}(\text{netlist, VCD}) \)
2. \( U \leftarrow \text{Index\_UNITs}(U) \)
3. for all Gate \( g \in \text{netlist} \) do
4. \( u_g \leftarrow \text{genUNITMembershipArray}(g, U) \) // bit vector indicating \( g \)'s membership in each UNIT
5. end for
6. \( H \leftarrow \emptyset \) // min heap of path segments, minimizes on timing slack
7. \( P \leftarrow \emptyset \) // set of explored paths
8. for all Path Endpoint \( e \in \text{netlist} \) do
9. \( e.key \leftarrow \text{min slack of any path containing } e \)
10. \( H.push(e) \)
11. end for
12. while size\( (P) < N \) do
13. \( p \leftarrow H.pop() \) // Path Segment with worst slack
14. if \( p \) is a full path then
15. \( P.append(p) \)
16. continue
17. end if
18. for all \( g \in \text{fanin}(p) \) do
19. \( u_g \leftarrow \text{getUNITMembershipArray}(p) \) // member vector for Path Segment \( p \)
20. end for
21. \( u_p \leftarrow \text{getUNITMembershipArray}(g) \) // member vector for Gate \( g \)
22. \( t_p \leftarrow \text{scalar\_product}(u_p, u_g) \)
23. if \( t_p > 0 \) then
24. // this path segment toggled at least once
25. \( s \leftarrow p.prepend(g) \) // new path segment with \( g \) added to \( p \)
26. \( u_s \leftarrow u_p \& u_g \) // bitwise &
27. \( H.push(s) \)
28. end if
29. end for
30. end while

Algorithm 2 reports the N most active paths in a design for an application, i.e., the N-worst exercised paths in terms of toggle count. Among all the exercised paths in a design, Algorithm 2 identifies the N paths that toggle the most times, in decreasing order of path toggle count. The toggle rate of a path determines its error rate when the path has negative slack. Thus, this algorithm can be used to identify the paths that have the potential to produce the most errors when guardbands are relaxed such that timing constraints for the paths are not met.

The algorithm for finding the most active paths is similar to that for finding the most timing-critical paths. Instead of ordering explored path segments based on their timing slack using a min-heap, Algorithm 2 uses a max-heap that orders path segments based on the activity of a path segment. Another difference is that determining path toggle rates requires using UTs instead of UNITs, since the set of UNITs for an application do not retain all unique toggles for the application’s toggled-sets [9]. For example, if one toggled-set is a subset of another, it is not stored in the set of UNITs, and thus, the set of UNITs cannot maintain the number of times the gates in a UNIT have toggled. A UT membership array, on the other hand, can retain the number of toggles for each toggled-set by using a companion array that stores the number of times each toggled-set is encountered during unification of toggled-sets.

AlGORITHM 2 Pseudocode to report the N-worst exercised paths in terms of toggle count/rate

1. \( U \leftarrow \text{Generate\_UTs}(\text{netlist, VCD}) \)
2. \( T \leftarrow \text{getToggleRatesOfUTs}(U) \)
3. \( U \leftarrow \text{Index\_UTs}(U) \)
4. for all Gate \( g \) in \text{netlist} do
5. \( u_g \leftarrow \text{getUTMembershipArray}(g, U) \) // bit vector indicating \( g \)'s membership in each UT
6. end for
7. \( H \leftarrow \text{max heap of path segments. } \text{Maximizes on path segment activity} \)
8. \( P \leftarrow \emptyset \) // set of explored paths
9. for all Path Endpoint \( e \) do
10. \( H.push(e) \)
11. end for
12. while size\( (P) < N \) do
13. \( p \leftarrow H.pop() \) // Path Segment
14. if \( p \) is a full path then
15. \( P.append(p) \)
16. continue
17. end if
18. for all \( g \in \text{fanin}(p) \) do
19. \( u_g \leftarrow \text{getUTMembershipArray}(p) \) // member vector for Path Segment \( p \)
20. end for
21. \( u_p \leftarrow \text{getUTMembershipArray}(g) \) // member vector for Gate \( g \)
22. \( t_p \leftarrow \text{scalar\_product}(u_p, u_g) \)
23. if \( t_p > 0 \) then
24. \( s \leftarrow p.prepend(g) \) // new path segment \( s \)
25. \( u_s \leftarrow u_p \& u_g \) // bitwise &
26. \( H.push(s) \)
27. end if
28. end for
29. end while

C. N-most active paths in a slack range

While identifying the most active paths in a design may be interesting for several application-specific and BTWC analyses and optimizations, in an overscaling scenario, only the active paths with negative slack cause errors. Thus, it is useful not only to identify the most active paths in a design but also the most active paths with timing slack in a particular range. For example, the most active paths with the least timing slack will be the paths that cause the most errors as the design is scaled past its critical operating point. As another example, this analysis routine can be used to create the dynamic timing slack distribution [5] for a design − a histogram showing the sum of toggle rates for the paths within each range of timing slack.

UNITs characterize design activity using fewer toggled-sets and thus have lower overhead in terms of analysis time and memory usage. However, UNITs do not retain toggle count information, while UTs do.
The dynamic timing slack distribution describes how the error rate of a design changes as guardbands are relaxed or overscaling changes the operating point of a design. This information can also be used to determine the optimal operating point for a BTWC design that minimizes energy while meeting a particular error constraint. It can also be used to determine how to apply optimizations to reshape the dynamic slack distribution to optimize a design for a particular target error rate. Algorithm 3 reports the N-worst active paths within a timing slack range, i.e., the N most toggled paths in decreasing order of toggle count within a specified timing slack range.

Algorithm 3 Pseudocode to report the N-worst exercised paths in terms of activity, within a slack range

Procedure Find_Nworst_Active_Paths(N, netlist, VCD, CDS, Smin, Smax) // num paths, min_slack, max_slack
1. \( U \leftarrow \text{Generate_UT(netlist, VCD)} \)
2. \( T \leftarrow \text{getToggleRatesOf(T(VCD))} \)
3. \( U \leftarrow \text{Index_UTs(T)} \)
4. for all Gate \( g \) in netlist do
5. \( u_g \leftarrow \text{getUTMembershipArray}(g, U) \) // bit vector indicating \( g \)'s membership in each UT
6. end for
7. if \( \text{max}_{H(e)} \) max heap of path segments. // Maximizes on path segment activity
8. \( P \leftarrow \emptyset \) // set of explored paths
9. for all Path Endpoint \( e \) do
10. \( H.push(e) \)
11. \( t_e \leftarrow \text{scalar product}(u_e, T) \)
12. end for
13. while \( \text{size}(P) < N \) do
14. \( p \leftarrow H.pop() \) // Path Segment
15. if \( p \) is a full path then
16. \( P.push(p) \)
17. continue
18. end if
19. for all \( g \in \text{fanin}(p) \) do
20. \( s \leftarrow p.prepend(g) \) // generate new path segment \( s \)
21. \( S_{min} \leftarrow \text{getLowestTimingSlackThrough}(s) \) // slack of longest path through segment \( s \)
22. \( S_{max} \leftarrow \text{getHighestTimingSlackThrough}(s) \) // slack of shortest path through segment \( s \)
23. if \( [S_{min}, S_{max}] \cap [S_{min}, S_{max}] \neq \emptyset \) then
24. \( u_g \leftarrow \text{getUTMembershipArray}(g) \) // Gate \( g \)
25. \( u_{s} \leftarrow \text{getUTMembershipArray}(s) \) // Path Segment \( s \)
26. \( t_e \leftarrow \text{scalar product}(u_{s}, T) \) // toggle count of this path segment
27. \( H.push(s) \)
28. end if
29. end for
30. end while

D. Correctness Proof

In this section, we prove that our algorithms generate the N-worst paths for their respective metrics. Due to space constraints, we only present the proof for Algorithm 1; however, all the algorithms follow a similar structure and have similar proofs that can be derived from the proof for Algorithm 1.

Theorem 1. Algorithm 1 reports the exercised (toggled) timing critical paths in increasing order of timing slack (decreasing order of timing criticality).

Proof. We prove the theorem in two parts: (1) the reported paths are exercised, and (2) paths are reported in decreasing order of timing criticality.

Reported paths are exercised: For each path segment popped from the heap, all the gates in the segment belong to at least one common UNIT, since only segments with at least one non-zero UNIT membership vector element are pushed onto the heap. UNIT membership implies that all the gates in a segment toggled together in at least one cycle. Therefore, for any path popped from the heap, all the gates in the path must have toggled together in at least one cycle. Thus, the path is exercised (toggled), by definition (see Section III).

Paths reported in decreasing order of timing criticality: Suppose that \( P \) is a path with slack \( S \) popped from the top of the heap. The longest path through \( P \) (\( P \) is also a path segment) is \( P \) itself. Now, let \( p_1 \) be any path segment remaining in the heap, \( H \), and let the longest path through \( p_1 \) be \( P_1 \) with slack \( S_1 \). Since the heap orders path segments based on the minimum slack of the longest path through a path segment, it follows that \( S < S_1 \), and hence \( P \) is the longest path among all the paths that can be reported using path segments in \( H \).

Since all path endpoints are pushed onto \( H \), and all paths must contain an endpoint, the path segments in \( H \) can be used to generate all possible paths in the design. Thus, no exercised paths can be missed.

V. METHODOLOGY

We verify our techniques with experiments on a silicon-proven processor – openMSP430 [17]. Designs are synthesized, placed, and routed with TSMC 65G library (65nm), using Synopsys Design Compiler [18] and Cadence EDI System [19] assuming worst-case operating conditions. Gate-level simulations are performed by running full benchmark applications from Table II on the placed and routed processor using Synopsys VCS [20]. Activity information is read from the VCD file generated from gate-level simulation. Timing analysis is performed with Synopsys PrimeTime [21]. 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 and dhrystone_v2.1 are available in [17]. All other benchmarks are taken from [22].

VI. RESULTS

To demonstrate the efficiency and scalability of our graph-based techniques for dynamic timing and activity analysis, we compare them against the state-of-the-art path-based dynamic analysis technique [2]. However, head-to-head comparison against path-based analysis presents a challenge, due to the significant time and memory requirements of path enumeration used in path-based analysis [9]. In fact, attempting to perform path-based dynamic analysis on a full processor, even a small embedded processor, proved impossible. Due to the huge number of exercised paths in a processor, even for a small processor running an application with low activity, our server (with
TABLE II: Benchmark Descriptions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>Integer Multiplication</td>
</tr>
<tr>
<td>tea8</td>
<td>8-bit Tiny Encryption Algorithm</td>
</tr>
<tr>
<td>binSearch</td>
<td>Binary Search</td>
</tr>
<tr>
<td>rle</td>
<td>Run-Length Encoding Algorithm</td>
</tr>
<tr>
<td>intAVG</td>
<td>Integer Average</td>
</tr>
<tr>
<td>inSort</td>
<td>Insertion Sort</td>
</tr>
<tr>
<td>tHold</td>
<td>Threshold Cross Detection</td>
</tr>
<tr>
<td>div</td>
<td>Integer Division and Outputting</td>
</tr>
<tr>
<td>intFilt</td>
<td>FIR Lowpass Integer Filter</td>
</tr>
<tr>
<td>dhrystone_v2.1</td>
<td>Dhrystone Benchmark</td>
</tr>
<tr>
<td>dhrystone_4mcu</td>
<td>Dhrystone Benchmark for MCUs</td>
</tr>
</tbody>
</table>

TABLE III: As the number of exercised paths increases with the length of the execution time window, path-based analysis time grows and becomes unreasonable, even for a relatively short time window. Analysis time for our scalable graph-based algorithms remains low, since they efficiently explore exercised paths.

<table>
<thead>
<tr>
<th>Time Window (cycles)</th>
<th>Path-based</th>
<th>Alg. 1</th>
<th>Alg. 2</th>
<th>Alg. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>26.3</td>
<td>2.0</td>
<td>1.9</td>
<td>2.2</td>
</tr>
<tr>
<td>500</td>
<td>35.6</td>
<td>1.9</td>
<td>1.7</td>
<td>2.2</td>
</tr>
<tr>
<td>1000</td>
<td>54.7</td>
<td>2.0</td>
<td>1.9</td>
<td>2.3</td>
</tr>
<tr>
<td>2000</td>
<td>348.0</td>
<td>2.2</td>
<td>6.7</td>
<td>9.1</td>
</tr>
<tr>
<td>5000</td>
<td>645.0</td>
<td>2.2</td>
<td>6.5</td>
<td>8.7</td>
</tr>
<tr>
<td>10000</td>
<td>2258.6</td>
<td>2.5</td>
<td>12.9</td>
<td>18.7</td>
</tr>
<tr>
<td>30000</td>
<td>102355.9</td>
<td>2.3</td>
<td>17.1</td>
<td>25.4</td>
</tr>
</tbody>
</table>

many cores, large caches, and 64 GB of RAM) ran out of memory and could not complete the path-based analysis.

Since path-based dynamic analysis of full processors and full applications is not possible, in order to perform a head-to-head comparison, we were forced to resort to the same approach that has been used in all prior works involving path-based dynamic analysis; namely, use a small sample processor module to represent the processor and a small sequence of instructions to represent the application [1], [2], [3], [4], [5], [6], [11]. Thus, we compare the runtime of our scalable dynamic analysis algorithms against that of path-based analysis by analyzing only the execution unit module of openMSP430 over a limited time window of execution. Also, for the path-based technique, we only report the time required for enumeration of exercised paths, omitting the time required to select the N-worst paths for a given metric. For our algorithms, we report the entire runtime for identification of the N-worst paths.

Table III compares the analysis time of our scalable N-worst algorithms against that of path-based analysis for fixed N (N = 25k) and variable execution time window length (in cycles). As the length of the time window grows, path-based analysis time increases significantly, as does the gap between path-based and graph-based analysis time. The reason for this trend is that as an application executes for a longer period of time, more paths are exercised, and the time to perform path enumeration grows dramatically. However, graph-based analysis efficiently expands exercised path segments to identify the N-worst paths without any path enumeration. Due to the massive number of paths exercised in a processor design, even over a relatively short time window, path-based analysis time quickly becomes prohibitive. For example, for an execution time window of 50k cycles, path-based analysis takes almost three hours, whereas scalable dynamic timing analysis to report the N-worst exercised timing paths takes under three seconds. While analysis time for our scalable algorithms does increase slightly for larger time windows, since there are more exercised path segments to explore and manage in the heap, our graph-based approach explores them efficiently, and analysis time remains low.

Figure 2 shows the speedup achieved by our scalable dynamic analysis algorithms with respect to path-based analysis, for different execution time windows. Even for the smallest of time windows, the speedup is significant. As the time window grows larger, the speedup becomes extreme. For example, for a time window of 50k cycles, Algorithms 1, 2, and 3 achieve speedups of 4364×, 599×, and 404×, respectively. Note that these results are only for the processor’s execution unit module executing over a limited time window. The speedups would increase for a full processor and application, or as the complexity of the processor or application increase. Furthermore, the analysis times for our scalable techniques stand to improve significantly further in a parallel computing environment. As the length of the time window increases, the most time-consuming operation in our analysis techniques is computing the UNIT / UT membership vectors through bitwise AND operations (see Section IV). Since these AND operations are embarrassingly parallel, and the amount of parallelism is large (number of AND operations equals the number of UNITS / UTs, which is O(number of execution cycles)), execution time should easily improve by a factor close to the number of parallel compute units in a parallel processor, which is several hundred to thousand in modern data-parallel architectures [23].

Figure 3 compares analysis time of graph- and path-based techniques for a fixed execution time window of 10k cycles and varying number of reported paths (N). Path-based analysis time to report the N-worst paths is large even for a small value of N, since path-based analysis requires enumeration of all exercised paths, independent of the value of N. For graph-based analysis, on the other hand, analysis time is low and grows gradually with increasing N, since our scalable graph-based analysis computes the N-worst paths iteratively and only incurs a small incremental overhead to identify each additional path. Figure 4 shows the same data as Figure 3, excluding path-based analysis so the trends for different types of N-worst analysis can be distinguished.

Our final set of results demonstrates the capability of our scalable graph-based dynamic analysis algorithms to analyze a full processor and full applications. This full level of analysis, which is expected for commercial EDA tools is possible only for graph-based analysis, not for path-based analysis, due to the massive number of paths in modern designs [8]. Table IV shows dynamic analysis time for the full applications in Table II executed on the full openMSP430 processor. Our scalable algorithms complete dynamic analysis, even for large applications. Furthermore, note that analysis times can likely be reduced significantly with data-parallel processing, as described above. Also note that analysis time does not always increase with application execution time. One reason for this is that the number
TABLE IV: Our scalable dynamic analysis algorithms enable analysis of full applications on a full processor.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Sim. Time (cycles)</th>
<th>Analysis Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alg. 1</td>
<td>Alg. 2</td>
</tr>
<tr>
<td>multi</td>
<td>147</td>
<td>26</td>
</tr>
<tr>
<td>stdfl</td>
<td>4191</td>
<td>90</td>
</tr>
<tr>
<td>riscv</td>
<td>4723</td>
<td>100</td>
</tr>
<tr>
<td>rle</td>
<td>3848</td>
<td>148</td>
</tr>
<tr>
<td>intAVG</td>
<td>12398</td>
<td>85</td>
</tr>
<tr>
<td>intSort</td>
<td>28813</td>
<td>83</td>
</tr>
<tr>
<td>rHrd</td>
<td>28590</td>
<td>114</td>
</tr>
<tr>
<td>dlv</td>
<td>68801</td>
<td>77</td>
</tr>
<tr>
<td>intFilt</td>
<td>222395</td>
<td>238</td>
</tr>
<tr>
<td>dhrystone 4mcu</td>
<td>332977</td>
<td>116</td>
</tr>
<tr>
<td>dhrystone 2v2.1</td>
<td>478429</td>
<td>100</td>
</tr>
</tbody>
</table>
for various modes at various corners, which can easily be performed with our approach.

Crosstalk Analysis: Timing analysis tools such as PrimeTime [21] incorporate crosstalk analysis into STA. Since our techniques are inspired by STA, our algorithms can handle crosstalk analysis. For each UNIT or UT, we can perform crosstalk analysis to capture the exact delays of the nets and then generate a delay vector with the same dimensions as the UNIT / UT membership vector. This vector can be used while tracing path segments, and the delay of a path segment can be computed as its longest delay among all the UNITs or UTs it belongs to.

VIII. CONCLUSION

As variability increases with continued CMOS scaling, BTWC design has become an increasingly-popular technique to improve energy efficiency. Dynamic analysis techniques used in prior work on BTWC design are based on path-based analysis that involves enumeration of the exercised paths in a design; however, such techniques are not scalable due to the massive number of paths in modern CMOS designs – even small designs. In this paper, we presented a suite of scalable graph-based dynamic analysis techniques that encompass the core functionalities needed for BTWC design, analysis, and optimization. Compared to existing path-based techniques, our scalable dynamic analysis techniques for timing, activity, and timing-constrained activity analysis improve performance by $977\times$, $163\times$, and $113\times$ on average, and enable scalable analysis for full processor designs and full applications.

REFERENCES


