Parallel Multithread Analysis of Extremely Large Simulation Traces

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 1, 2022, accepted May 12, 2022, date of publication May 23, 2022, date of current version June 1, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3177613

Parallel Multithread Analysis of Extremely Large Simulation Traces

D. APPELLO[1], PAOLO BERNARDI[2], (Senior Member, IEEE), ANDREA CALABRESE[2], (Graduate Student Member, IEEE), G. POLLACCIA[1], STEFANO QUER[2], (Member, IEEE), V. TANCORRE[1], AND R. UGIOLI[1]

[1]STMicroelectronics, 20864 Agrate Brianza, Italy
[2]Department of Control and Computer Engineering, Politecnico di Torino, Dauin, 10129 Turin, Italy

Corresponding author: Stefano Quer (stefano.quer@polito.it)

ABSTRACT With the explosion in the size of off-the-shelf integrated circuits and the advent of novel techniques related to failure modes, commercial Automatic Test Pattern Generator and fault simulation engines are often insufficient to measure the coverage of particular metrics. Consequently, a general working framework consists of storing simulation traces during the analysis phase and collecting test statistics from post-processing. Unfortunately, typical simulation traces can be hundreds of gigabytes long, and their analysis can require several days, even on large and powerful computational servers. In this paper, we propose a set of strategies to mitigate the evaluation time and the memory needed to analyze huge dump files stored in the standard Value Change Dump format. We concentrate on burn-in-related metrics that current commercial fault simulators and Automatic Test Pattern Generators cannot evaluate. We show how to divide the analysis process into several concurrent pipeline stages. We revise the logic process of each stage and all principal intermediate data structures, to adopt smart parallelization with very low contention and extremely low overhead. We exploit several low-level optimizations from modern programming techniques to reduce computation time and balance the different pipeline phases. We analyze simulation traces up to almost 250 GBytes computing different testing metrics. Overall, we can keep under control the memory usage, and we show time improvements of over two orders of magnitude compared to previously adopted state-of-the-art tools.

INDEX TERMS Automotive SoCs burn-in, simulation analysis, parallel architectures, parallel applications.

I. INTRODUCTION

With the explosion of the size of the chips in terms of the number of gates, and the advent of novel failure mode strategies, a major concern for testing and reliability is the elaboration times of simulation and fault simulation phases. Nowadays, "big monster" Systems-on-Chip (SoCs) integrate tens of millions of gates, implementing computational units, processors, real-time hardware modules, memory cores, etc. However, even if each module is verified and validated during each design phase, the functional and structural analysis of the entire SoC may require huge computational efforts when the system is finally assembled [1].

In this framework, all reliability measures rely on Automatic Test Pattern Generators (ATPGs) and fault simulation engines [2]–[4]. These tools currently elaborate on several fault models [5] and provide the coverage of different stress metrics. Nevertheless, when considering new failure modes and fault metrics, the capabilities of ATPGs and fault simulators may be insufficient to reach the desired result accuracy. For these reasons, when a commercial tool cannot help too much to perform a specific measure, many strategies [6], [7] record the simulation trace and save into a file all values and timings for the set of selected signals (i.e., possibly all signals of the circuit). A commonly used file format for this task is the so-called Value Change Dump (VCD) [8].

Given the organization and structure of a VCD file dump, its analysis is theoretically straightforward. Usually, it is sufficient to select the signals to monitor, sequentially read their time evolution, and statistically collect all signal transitions or value combinations. Unfortunately, software tools performing this task entail two main practical problems: A huge

The associate editor coordinating the review of this manuscript and approving it for publication was Hongwei Du.

elaboration time and an extremely large memory requirement. For example, in a recent simulation session related to the effectiveness of Burn-In (BI) techniques [9], [10], we had to analyze an automotive device including approximately 30 million gates. During the examination phase, we stored a simulation trace of a size of almost 250 GBytes. We then used this file dump to evaluate the coverage of several BI-related stress metrics, including the "straightforward" toggle activity plus some more complex, time and topology-related metrics [8]. The evaluation of the toggle activity for about 1K clock cycles required an elaboration time of approximately three days on a single core mainframe architecture. Practically speaking, this delay constituted a huge problem at the corporate level, with nasty consequences on the productivity and performance of the entire working team.

In this paper, to reduce the time required to elaborate VCD files storing BI-related metrics for which commercial ATPG and fault simulation approaches are either inappropriate or extremely slow. We adopt a divide-and-conquer strategy composed of three main ingredients: Pipelining, parallelization, and in-depth code optimization. First, we divide the overall elaboration process into independent phases and insert them into concurrent pipeline stages. Then, we analyze our pipeline to discover computational costs and waiting (idle) times of each phase. Finally, as our computational power is not unlimited, we adopt paralleliszation enriched by low-level optimizations to balance the different steps and to dedicate more effort to the more critical phases. We strive to reduce contention, waiting times, and overheads to find the right balance in our recipe.

Notice that we work with traditional enterprise systems, organized through a centralized server, where the simulation engine stores a individual simulation file. In other words, we do not take into consideration the possibility to use distributed systems and related techniques (such as Apache Hadoop MapReduce [11]) and we do not have to consider problems associated with the network architecture, such as network topology, latency, bandwidth, and packet loss.

As the original VCD file stores signal variations sorted by increasing clock ticks, the exact logic of our application strongly depends on the testing metric we need to compute. Anyway, superficially speaking, we proceed as follows. Our process starts with a given VCD file stored by the simulator in the long-term memory unit of our system. The manager thread of our application initializes the entire process, and it runs all required working threads performing the different pipeline stages, i.e., it runs discovery, read, parse, elaborate, and write-back threads. The discovery thread partitions the file content and dispatches the resulting file partitions to the reading threads. Each reading thread stores "large enough" file partitions into the main memory. Parse threads visit these partitions and store intermediate results in a thread-local memory-efficient data structure. In parallel, elaboration threads elaborate on this intermediate information to gather the final testing metric. Write-back threads progressively take care of all post-processing operations eventually required by the computed metrics and store "finalized" local results to the long-term memory system. The entire process continues until all VCD file partitions have been read, parsed, elaborated, and written back to the long-term memory system.

Please, notice again that since the size of the VCD file maybe extremely large, not all file partitions can be read and manipulated simultaneously in the central memory. Therefore, to be as effective as possible, the system is organized as a pipeline where all stages ideally require the same amount of time and a very limited amount of partitions need to be stored at the same time into the main memory. Moreover, using more threads and computing cores allows us to leverage the time requirements of the different stages. Adopting low-level non-blocking system calls enables us to perform I/O and memory manipulation as effectively as possible. Avoiding stalls, adjusting the workload, and minimizing contentions among different running tasks, allow us to reach the appropriate equilibrium among the stages and the desired time and memory efficiency. Furthermore, notice that while the functionalities of the elaboration phase and the write-back step need to be tailored to the desired metrics and require the intervention of the tool developer, the other stages are generic and are not metric-dependent.

The experimental part shows the results collected on a server architecture that manages threads, hyper-threads, and software threads. We evaluate our solution on VCD files generated by the simulation of large industrial circuits. These files store testing metrics related to BI stress and include timing and topology information. We collect three different testing metrics, representing the single point, the single point extended, and the multiple point stress analysis, on VCD files of increasing size to almost 250 GBytes. The speedup, of over two orders of magnitude compared to the original tool, looks very consistent, and it linearly scales with the number of adopted cores. Moreover, memory usage can be maintained under control and adapted to the hardware architecture.

This paper is an extended version of the conference paper [12]. The current work extends the original one in several directions. First, we completely revise the original procedure as the new one divides the computational process into more fine-grained pipeline stages. Moreover, the new algorithm better balances the different phases as it uses more optimized intermediate data structures and further reduces contention and memory overheads. Secondly, the new work details the logic process. It reports the implementations to compute three different testing metrics (the single point, the single point extended, and the multiple point stress analysis). Finally, this new work reformulates all experimental results. These now include complete, separate, and detailed evidence on the three different test metrics, both time and memory, on a larger set of VCD files.

The paper is organized as follows. Section II introduces the related works and the necessary background on simulation and dump format. Section III introduces our approach from a high-level point of view. Sections IV, V, and VI detail the process of computing three different testing metrics, namely

the single point, the single point extended, and the multiple point stress analysis. Section VIII reports our experimental evaluation of the three previous strategies. Finally, Section IX draws some conclusions.

II. BACKGROUND AND RELATED WORKS

Figure 1 illustrates the workflow required to compute stress metrics starting from the simulation of the circuit [12]. As motivated in the introduction, simulation traces can be analyzed by ATPGs or fault simulation engines if they are sufficiently instructed. The simulation dump analysis may require ad-hoc tools to manage the selected metric values if they are not.

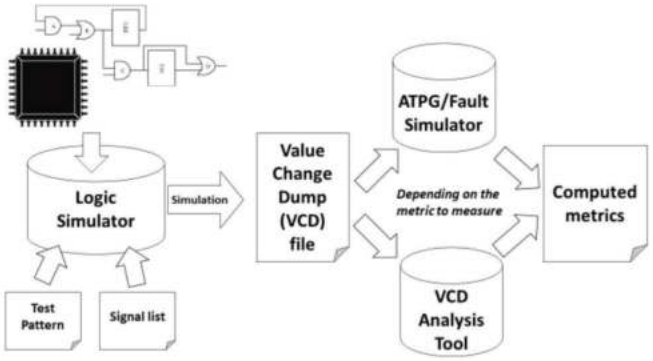

FIGURE 1. The process flow for testing metric evaluation in case an ATPG or a Fault simulator can be used, or when an ad-hoc tool is necessary.

This section provides the reader with the necessary background on the VCD file format, the most used simulation dump format, and the metrics usually evaluated to characterize a specific pattern or functional behavior.

A. VCD FILES

When a simulator must generate a trace, a commonly used file format is the so-called Value Change Dump (VCD) [8]. Its layout is composed of two logical sections:

- The first one includes a header and a declaration part. The former defines generic information on the simulation performed. The latter locates all signals into a hierarchical description, assigns an integer identifier to each signal, and specifies an initial value for each design signal.
- The second section reports all values assumed by the previously defined signals over subsequent time frames. Transitions are reported by increasing time values, such that, for each time unit in the future, only the set of all transient signals reported with the corresponding new value.

A diagrammatical representation of such a format and its high-level layout is reported in Figure 2.

The list of signals (or circuit nodes) to evaluate must be selected before the simulation runs. When we need to collect chip-level measurements, the scenario to investigate is often the one including all signals. For example, all nodes of the circuit have to be considered if the stress ability of the scan pattern/functional sequence is the objective of the analysis. It is a common practice for the evaluation of BI stress metrics, as described in Section II-B.

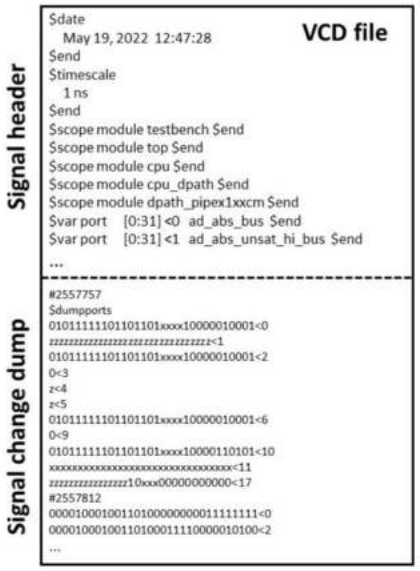

FIGURE 2. Simulation dump format: A VCD file and its high-level structure composed by the header and the simulation part.

If the device includes up to millions of gates, as automotive chips and the simulation considers a long list of patterns, VCD files may reach a size up to hundreds of GBytes.

B. TEST AND BURN-IN RELATED STRESS METRICS

The analysis of a VCD file is often perceived as a tedious, risky, and time-consuming activity. Modern state-of-the-art ATPG engines (essentially their fault simulation procedures) can digest VCD files to evaluate the toggle activity and several other fault coverage metrics (such as stuck-at, transition delay, bridges, etc.). Unfortunately, they often generate a limited number of models and metrics with inadequate statistics and information details. For example, in our particular field of application, ATPG engines cannot measure the stress metrics related to the BI process or for the activity over time of a circuit. Therefore, there are cases in which we need to use more specific tools.

Figure 3 illustrates the use of scan chain(s) to stimulate a chip. BI patterns add logical stress to the electrical burden (higher voltage supply) and the higher temperature inside the climatic chamber. Such extra stress tent to exacerbate latent defects of various nature by moving the circuit in specific "highly-stressing" logic conditions, possibly reflecting additional physical fatigue for the device under BI. This process addresses the problem of infant mortality failure, and its major purpose is to make weak devices break and screen them before they reach the market.

In our production flow, we evaluate the following BI stress metrics [12], [13]:

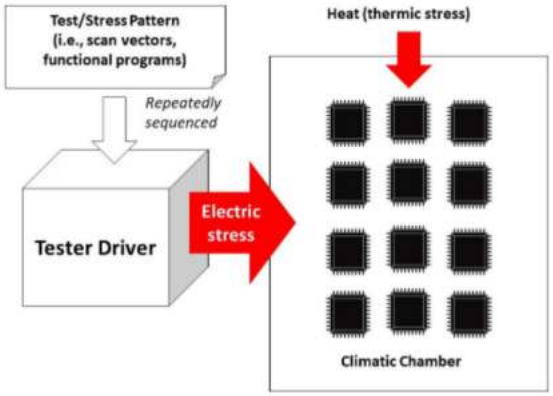

FIGURE 3. BI stress overall scenario.

- Single point stress metrics. An example of this metric is the very common "toggle activity" [14], which indicates if a circuit node assumes both logic values '0' and '1' during the simulation. If this happens, the node is covered. Within the single point stress metrics, we may identify:
  - Extended statistics. We compute the number of times a node toggles during the simulation, along with the values assumed by the nodes on each time frame. This metric is rarely included in ATPG analysis.
  - Timing-related measurement. We consider the logic behavior of the circuit along time, i.e., we compute the average toggling frequency. This metric is sometimes important to ensure similar stress for all nodes of the circuits.
- Multiple points stress metrics. This measure has similarities with the bridging fault concept as we evaluate the logic values of adjacent nodes [15]. As ATPG cannot provide time-related measures, we transform the time-centric nature of the original VCD report into a signal couple-centric one. In other words, we transform the original VCD file into a new report highlighting the couples in which the two signals hold opposite values for a time span longer than a minimum desired one (e.g., if the first element maintains the logic value '0', the second element must hold '1', and both shall be stable for at least a certain amount of time). If both signals of the couple assume both values, the couple is fully covered.

III. OUR METHODOLOGY

Although manipulating a VCD file is theoretically straightforward, current VCD files may have a size up to a few hundred of GBytes. Moreover, for some types of analysis, the generated output may even require more memory space than the original simulation trace. For example, this is true for the single point extended (or full) analysis, for which not only the final file may be huge, but it is usually impossible to store all intermediate information in the central memory. Consequently, processing a VCD file may require enormous resources [16], [17], i.e., computation servers with high memory capacity, and it may imply very long waiting times. A more accurate analysis of the existing applications processing VCD files showed that computation costs often depend on I/O operations, i.e., the ones required to read the input files and store the final result on the external memory unit. Intermediate elaborations that are necessary to compute the desired testing metrics have a cost that depends on the type of metric collected. In this section, as the desired coverage metric strongly influences the logic flow of our process, we concentrate on the former cost, highlighting our strategy from a very high-level point of view. We concentrate on the latter cost and detail the logical steps required to compute our metrics in the following sections. Our process is illustrated in Figure 4.

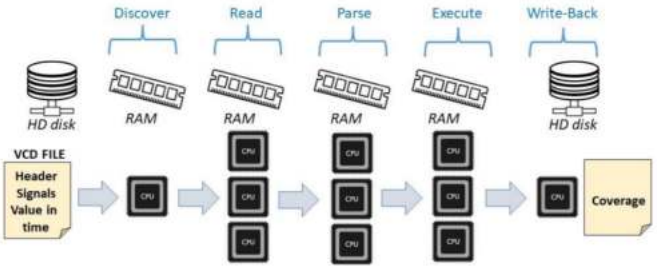

FIGURE 4. The logic flow of our approach: A carefully designed pipelined process with several scrupulously optimized parallel stages.

Generally speaking, our recipe to efficiently manipulate VCD files includes three main ingredients.

The first ingredient consists of a divide-and-conquer approach to perform the coverage computation procedure and dispatch its different phases into different pipeline stages. In the most general case, our pipeline includes five stages, which we refer to as:

- Discover
- Read
- Parse
- Execute
- Write-Back.

In specific cases, i.e., for some coverage metrics, some steps can be missing, or different steps can be merged into a unique phase.

During the Discover phase, we pre-process the file to set up the reading stage. Essentially, we read the header of the VCD file, initialize all main data structures, and partition the simulation section of the file, providing both the order of the partitions and their sizes to the Read stage.

During the Read stage, we read the simulation section of the VCD file from the external memory, and we transfer it into the main memory in pieces of proper size. The dimension of the partitions is very important as it has a strong impact on the level of parallelism we can adopt in the following phases and on the quantity of main memory used by the entire process.

We analyze the previous file pieces during the Parse phase to extract compact information from which to derive our coverage metrics. This phase is the most memory-critical one, and great care has to be applied in designing the intermediate data structures used to store the required information.

The Execute phase computes the final metric or re-organizes the data in a proper internal (RAM-based) representation.

The last Write-Back stage stores our results back onto the external memory storage. In some cases, we also perform some final manipulation on the data generated by the Execute phase during this step.

Although this logic scheme appears very simple, we had to implement the different phases with great care and keep them as balanced as possible to reach the desired efficiency and scalability. Our pipeline is organized as shown in Figure 5.

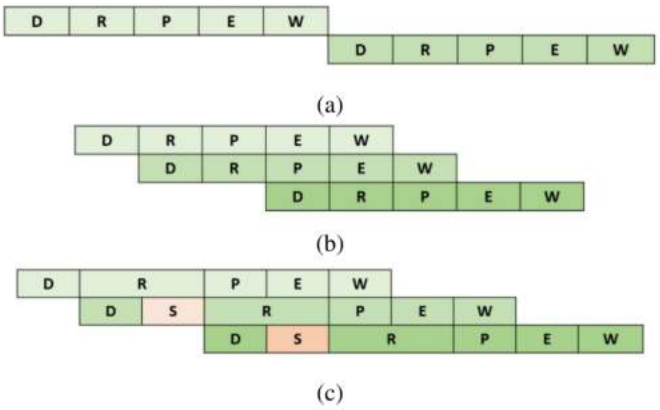

FIGURE 5. Our process pipeline: We try to parallelize the use of the different hardware units (disk, RAM, CPUs, etc.) as much as possible. Letters D, R, P, E, and W stand for the Discover, Read, Parse, Execute and Write-Back stages, respectively. S stands for Stall. (a) The original purely sequential execution flow. (b) The balanced pipeline execution with all stages having the same time length. (c) The unbalanced pipeline execution with unbalanced stages and the necessity to stall or parallelize the slower phase.

Figure 5a represents the sequence of our five process stages without pipelining (letters D, R, P, E, W, and S stand for Discovery, Read, Parse, Execute, Write-Back, and Stall, respectively). Pipelining emerges in the other two representations. In Figure 5b, all stages are supposed to have the same complexity and last the same amount of time. This is not the case, and the real situation is the one depicted in Figure 5c. In this case, the Read phase is supposed to be much longer than the others and it drastically unbalances the pipeline, inserting long stall phases.

To rectify this problem, or at least reduce it to the minimum possible one, we resort to our second ingredient. As "programming" today is "parallel programming" [18]–[24], we resort to many-core parallelization to reduce the elaboration time and memory usage of the more complex stages [25], [26]. From the architectural point of view, parallelization implicitly balances the different algorithmic stages of the pipeline of Figure 5b. Thus, we run a proper number of threads during each phase to minimize the waiting times for the following stage. Threads are organized into thread pools to minimize thread overhead. In turn, pools are in charge of a queue of tasks, usually manipulated following a producer and consumer approach. Overall, the multi-thread divide-and-conquer process has highly balanced tasks and low contention (thus, overhead) among threads.

Our third and last ingredient is code optimization. In this case, we resort to features delivered by modern languages and the latest operating systems. More specifically, we minimize synchronous I/O, waiting times, and dynamic memory manipulation.

In the following three sections, we specialize the approach described in Figure 4 to compute the testing metrics introduced in Section II-B, that is, statistically analyze the behavior of single signals, perform an extended analysis on the toggle activity of single signals, and run a statistical study on the strain stress of adjacent signals. In these sections, we clarify each testing metric's implication on our parallel implementation, and we report all implementation-related details.

IV. THE SINGLE POINT STRESS ANALYSIS

The single-point stress analysis is the simplest among our testing metrics and requires less memory and time resources. We evaluate the number of times a signal toggles during the application of the pattern sequence. The target is to produce a condensed report including the overall amount of toggling events for every signal under consideration and the average toggling events over the whole set of signals.

On the one hand, storing the number of variations for all signals requires a single-column table (which we call the signal or result table) with several rows equal to the number of signals. Consequently, the quantity of memory necessary to store temporary the number of signals bounds information. Moreover, this quantity is not influenced by the time length of the simulation or by the number of toggling events. On the other one, since it is not necessary to store time information about the toggle activity of the signals, different file sections can be managed in parallel without any constraints on the manipulation order. Thus, threads do not need any sort of synchronization, and it is sufficient to update the signal table in mutual exclusion. Notice that, for this metric, the Execute stage is directly performed by the Parse phase. The above considerations drastically simplify our concurrent process, speed up the entire process, and make temporary memory usage reasonable compared to the original VCD file.

Following the previous ideas, our procedure is described in Figure 6. At the very beginning, our application initializes the following data structures and logic processes:

- A single thread DT performing the Discover phase.
- Two First-In First-Out (FIFO) queues, namely Q_1 and Q_2, are of finite size and store different intermediate results (namely partition extremes and partition references) along the process.
- A thread pool RT, with N_R reading threads, taking care of the Read phase. These threads elaborate the tasks stored into Q_1 and store tasks into Q_2.

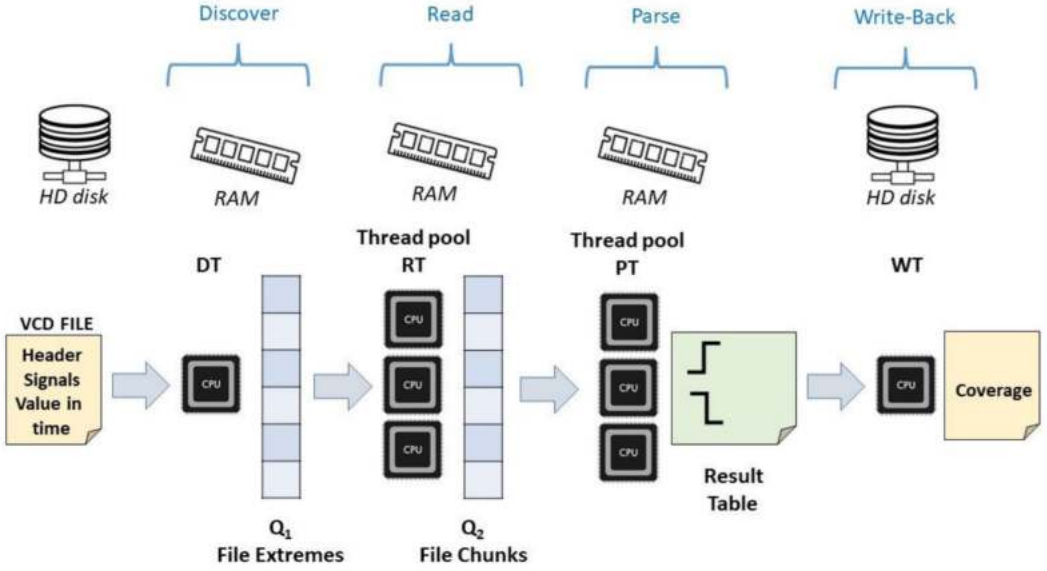

FIGURE 6. The architectural pipeline of our application for the single point stress analysis.

- A thread pool PT, including N_P parsing threads. PT deals with the data enqueued into Q_2 and it updates the final signal (or result) table.
- A single thread WT performing the Write-back phase.

Initially, the discovery thread DT reads the file to compute each file partition's initial and final addresses. These addresses are stored in the first queue Q_1. Q_1 is a FIFO queue, and it is used to manipulate file partitions sorted by increasing simulation times. All queues are always accessed by adopting a producer and consumer scheme. DT acts as a single producer for Q_1; the reading threads in the RT pool act as consumers. Each consumer extracts elements from Q_1 and reads the corresponding file section from the file. Thanks to the pipeline strategy, reading threads may start running as soon as at least one partition has been inserted in Q_1, without any need to wait until the entire file is read and partitioned by DT.

Due to hardware constraints, and especially when running on very simple hardware architectures, very few reading threads may be sufficient because operations on the secondary memory are often natively synchronous with limited possibility of optimization. However, partitions are read from the file using low-level system calls optimized to perform reading operations in the fastest possible way. File partition references are inserted into a second FIFO queue Q_2. Notice that, to minimize the cost of dynamic memory allocation as much as possible, we also use a pool of pre-allocated data structures to store file partitions. In this way, when a thread within the PT pool dequeues data from Q_2, an RT thread can upload a new file partition into the main memory without losing time to allocate it. Threads belonging to the Parse pool PT start running as soon as Q_2 is not empty. When a file partition is available on the queue Q_2, a parsing thread analyzes that specific file partition. It updates the corresponding toggle activity in the single-column result table previously described. Once all partitions have been completely parsed, the final table is written back to the secondary memory by the writing thread WR. Notice that for the writing phase and for this metric, a single Write-back thread is sufficient.

As already observed in the introduction, to limit the quantity of central memory to be used to store temporary information and to avoid unbalanced pipeline stages are very important factors for our approach. The partition size strongly influences memory usage, and it can be dynamically adjusted by the discovery thread DT. We normally worked with file partitions of about 8 MBytes as this value constitutes a good compromise between memory and time requirements. The most important parameter to balance the pipeline is the number N_P of threads within the Parse pool. We usually set the size of the queue Q_2 to 40 entries. To avoid time overheads for memory allocation, we use several pre-allocated partitions five times larger than the number of entries of Q_2. As a consequence, we need about 1.6 GBytes of allocated memory for the entire procedure.

The pseudo-code of the previous process is reported by Algorithm 1. It follows the previous logic, and it is quite self-explanatory. In lines 1-3, our process initializes all required data structures, and it runs all working threads belonging to all thread pools. The Discover, Read, Parse, and Write-Back pipeline stages are implemented by DT, RT, PT, and WT. As previously mentioned, the Execute stage has been merged to the Parse one for this metric computation. Notice that all phases can run in parallel, except for the Write-Back stage that can be executed only once the entire file has been parsed. In the pseudo-code, this effect is obtained using the synchronization barrier reported in line 30.

V. THE SINGLE POINT EXTENDED (Full) ANALYSIS

As in the single point stress analysis, we consider the behavior of single signals during the full analysis. Nevertheless, we aim to collect extended (or full) statistics, i.e., we want

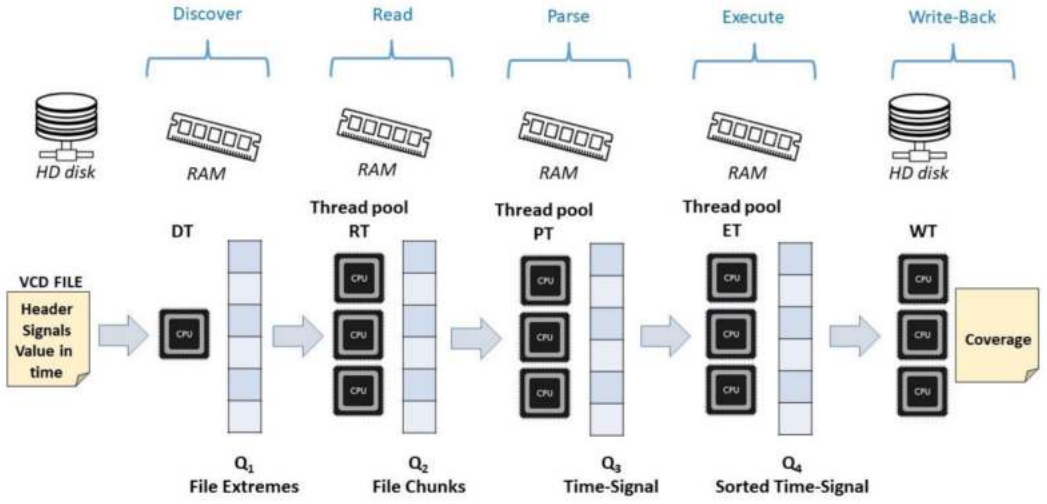

FIGURE 7. The architectural pipeline of our application for the single point full analysis.

to store in the final report, and for each signal, the complete list of clock ticks in which the signal toggles. In other words, we want to perform a sort of matrix transposition, as we move from the original VCD time-based information (which lists the variations of all signals in each time unit) to the final signal-based information (which lists for each signal its entire toggle activity during the simulation). For this reason, the final result for this metric may require more disk space than the original VCD file. As a consequence, great attention must be taken to optimize all intermediate data structures. Notice, however, that even if this metric may generate cumbersomely and bulky final result, we consider it a crucial phase in specific corner-case circumstances.

The main characteristics of our process, illustrated by the diagrammatical representation of Figure 7 and the pseudo-code of Algorithm 2, are the following. The first two pipeline stages, i.e., the Discover and the Read phases, proceed as analyzed in Section IV. The main difference in this first part of the process is that the Discover thread partitions the VCD file into chunks, including entire (one or more) time frames. As for the previous metric, the extremes of these file partitions are enqueued into queue Q_1.

Read threads within the thread pool RT proceed exactly as for the previous metric, i.e., they extract file extremes from Q_1,read entire file partitions from disk, and store the partition references into the queue Q_2. Since the Read stage is one of the slowest stages in the pipeline, we optimized the memory allocation process. In particular, Q_2 holds references to a pool of pre-allocated file partitions. This pool, on which RT is the only writer and PT is the only reader, has a size two times the maximum size of the queue Q_2. We do not need a separate queue to manage the selection of the pre-allocated parts since PT does not perform writing operations on it and file parts are not overwritten during their elaboration, as the number of pre-allocated file parts is large enough This procedure allows us to save time on memory allocations and deallocations. At the same time, we do not have to perform any expensive selection of the pool partition, which is managed as a circular buffer.

Each thread within the parse pool PT extracts a file partition from the queue Q_2 and parses it. Nevertheless, Parse threads must create the data structure to collect the full analysis is drastically different from the one used in Section IV. For this metric, each Parse thread creates a list of records for each time frame. Each record stores a signal change (the signal identifier and its value) that occurred in that time frame. Thus, every time a Parse thread (parsing a specific file partition) encounters a signal activity, it just appends the newly discovered activity on one of its signal lists.

At the end of this process, the toggle activity is time-sorted (as each parse thread takes care of entire time frames) but not signal-sorted. For that reason, each list is eventually enqueued into queue Q_3 managed by the pool ET of Execute threads. Each Execute thread extracts a list from Q_3, sorts it by increasing signal identifiers, and stores it into queue Q_4. As soon as a single sorted list is present within Q_4, Write-Back threads within the thread pool WT start to work. To save memory, Write-Back threads store sorted lists on files and, at the same time, they merge lists, or sets of lists, representing contiguous time frames. This repeated process, briefly sketched by function WT in the pseudo-code of Algorithm 2, leverage computation times and memory usage.

To sum up, a few main aspects differentiate this procedure from the one described in Section IV:

- We do not have to store a summary of the toggle activity but the entire list of variations. It implies using one list of modifications for each signal instead of a unique and "relatively" compact table.
- An Execute thread must sort each list of variations before being stored in the output file.
- Write-Back threads leverage computation time and memory usage by adopting an iterative approach.

Algorithm 1 Pseudo-Code for the Single-Point Stress Analysis.

```
SinglePointStressAnalysis ()
 1: Init queue Q1 and Q2
 2: Init threads DT and WT
 3: Init thread pools RT and PT
 4: synch_barrier (wait for all threads to terminate)

 5: DT ()
 6: while (read_file() != EOF) do
 7:    insert signal name in signal table
 8:    define partition limits
 9:    enqueue (limits, Q1)
10: end while

11: RT ()
12: for each thread in RT in parallel do
13:    while (Q1 ≠ ∅) do
14:       limit = dequeue (Q1)
15:       content = read_file (limit)
16:       enqueue (content, Q2)
17:    end while
18: end for

19: PT ()
20: for each thread in PT in parallel do
21:    while (Q2 ≠ ∅) do
22:       partition = dequeue (Q2)
23:       while (partition is not parsed completely) do
24:          {s, t} = get toggle activity (signal s, time t)
25:          update (table, {s, t})
26:       end while
27:    end while
28: end for

29: WT ()
30: synch_barrier (wait for all PT threads to terminate)
31: write (table, file)
```

They store sorted lists on temporary files and merge previously-stored files with new incoming sorted lists until a unique sorted list stores the desired results.

VI. THE MULTIPLE POINT STRESS ANALYSIS

In the adjacent nodes strain stress analysis, we consider multiple points and evaluate the number of times adjacent nodes take opposite values for a user-defined amount of time. Storing all signal variations would make this analysis very similar to the one of Section V, with a huge amount of memory dedicated to store temporary information. Moreover, storing all changes would make the size of the intermediate data structure strongly dependent on the number of signal variations, with a strong impact on the logical organization of our dynamic data structures and memory contention among threads to access them.

Algorithm 2 Pseudo-Code for the Single Point Full Analysis.

```
SinglePointFullAnalysis ()
 1: Main
    {See Algorithm 1}

 2: DT()
 3: while (read_file() in VCD header) do
 4:    insert signal name in signal table
 5:    partition_start = find (time)
 6: end while
 7: while (read_file() != EOF) do
 8:    partition_end = find (time)
 9:    enqueue([partition_start, partition_end], Q1)
10:    partition_start = partition_end
11: end while

12: RT ():
    {See Algorithm 1}

13: PT ()
14: for each thread in PT in parallel do
15:    while (Q2 ≠ ∅) do
16:       Init list of changes
17:       partition = dequeue (Q2)
18:       while (partition is not parsed completely) do
19:          {s, t, v} = get toggle activity (signal s, time t,
             value v)
20:          update (list, {s, t, v})
21:       end while
22:       enqueue (list, Q3)
23:    end while
24: end for

25: ET ()
26: for each thread in ET in parallel do
27:    while (Q3 ≠ ∅) do
28:       list = dequeue (Q3)
29:       sort list per signal ID
30:       enqueue (list, Q4)
31:    end while
32: end for

33: WT ()
34: for each thread in ET in parallel do
35:    while (Q4 ≠ ∅) do
36:       list = dequeue (Q4)
37:       merge list with existing file if it exists
38:       write result in new temporary file
39:    end while
40: end for
```

To rectify the above problems, we reason as illustrated in Figure 8. The top part of the picture depicts a situation in which we suppose 4 threads, namely $\{T_1, \ldots, T_4\}$, manipulate several file partitions, namely $\{F_1, F_2, \ldots, \}$. The time

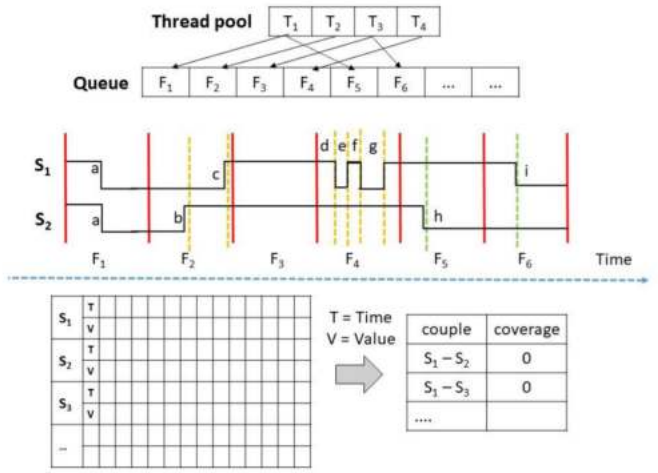

FIGURE 8. Collapsing information on a partition by partition basis to collect adjacent nodes stress-related statistics.

plot is an example of the possible time behavior of two signals, S_1 and S_2, along with the entire simulation analysis. As in the previous cases, each thread manipulates a file partition. Thus, thread T_4 may read file partition F_4 and it will detect 4 variations $\{d, e, f, g\}$ for signal S_1. At the same time, thread T_4 will detect that signal S_2 remains to 1 during the entire period associated with the partition. The core idea to avoid storing the entire signal history is to realize that, in the adjacent node analysis, signal changes need to be saved only when they are meaningful concerning the time defined by the user to perform the analysis. For example, let's consider file partitions that span a simulation time smaller than (or at most equal to) the user-defined time. We can store for each signal and each partition only the first and the last transition activity. Due to time-limit constraints, these transitions can cover a signal only if they can be positively combined with the transitions eventually detected on adjacent file partitions. This process has three advantages:

- It drastically limits the number of signal changes we have to store (just two for each signal and each file partition).
- It allows us to allocate table blocks statically and avoid extra run-time for dynamic allocation.
- It avoids any sort of contention among threads as each thread updates only the two dedicated entries within the global table.

Consequently, for the multiple point stress analysis, the discovery thread *DT* parses the entire VCD file to return partitions of constant time-length and variable size. Partitions extremes are enqueued into Q_1. As each partition does not span a time length sufficient to be significant for our stress test analysis, two signals must assume different values for a time-length spanning contiguous partitions. As a consequence, each thread stores only the first and the last toggle activity for each partition and to evaluate the stress activity, we consider adjacent partitions. We put together the toggle activity of each partition with the one detected on the partition on the left (the one with shorter simulation times) and the one on the right (with larger simulation times). This can be done using a two-dimensional matrix, where each row represents a signal, and the number of columns is strictly bounded by the number of partitions manipulated in parallel. Once at least two adjacent partitions have been parsed and the table reports the related toggle activity, we can move into the next phase to collect statistics. Statistics are gathered resorting to a double-buffering scheme, i.e., we use two tables, and we create a perfect pipeline: Parse threads PT store data on a table and Execute threads ET collect statistics from the other one.

Following the previous logic, our implementation is described in Figure 9 and by Algorithm 3. To compute the multiple point stress analysis, we need:

- A single discovery thread *DT*.
- Several queues, namely $\{Q_1, Q_2, Q_3, Q_4\}$, of finite size, each one to store different partial results.
- A set of table blocks, each one coupled with a file partition. In each table block, we store its first and last toggle activity for each signal in a specific time frame.
- Different thread pools, namely $\{RT, PT, ET\}$, including N_R Read, N_P Parse, and N_E Execute threads, respectively.
- A single Write-Back thread *WT*.

As far as the discovery thread and the reading threads are concerned, they proceed as described in Section V even if the Discovery thread detects partitions of a defined simulation length (*min_time*, line 11) instead of a defined size. The first main difference can be noticed during the Parse phase. As in the previous analysis, the FIFO queue Q_2 temporarily stores the reference to file partitions. However, to avoid serious slow-down due to memory allocation, we pre-allocated a set of table blocks and stored their references into Q_3. Unlike what we indicated in Section V, we need the queue Q_3, because Q_4, which stores the references to the data held by Q_3 can be written by both *PT* and *ET* threads. As a consequence, each parsing thread *PT* runs when a file partition is available in Q_2 and a free table block in Q_3. Once this happens, the parsing thread parses that specific file partition, and it stores the first and last toggle activity of each signal in that specific table block. Once the partition has been completely parsed, the reference to the table block is inserted into queue Q_4. Unlike all previous queues, Q_4 is a priority queue as it is better to manipulate partitions sorted by (increasing) simulation times. When at least two partitions in queue Q_4 are related to contiguous time frames, one thread in the thread pool *ET* merges the results available in the two blocks into a unique table block. This block is stored back into Q_4 since it might be re-used soon, whereas the empty block is stored into Q_3, as a free partition to be updated by *PT*. Following merging activity finally produces a unique table block. This last block is written back to the external unit by the writing thread *WT*.

Overall, the main aspect that differentiates this procedure from the one used in Section IV and V is the work done by the Execute threads *ET*. These threads extract table blocks from the queue Q_4 and compute the final testing metrics as

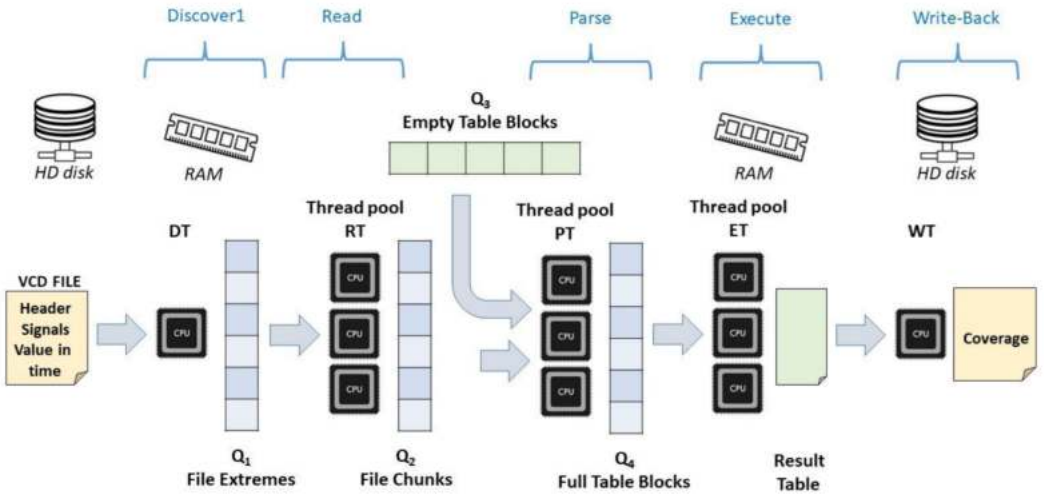

FIGURE 9. The architectural pipeline of our application for the multiple point stress analysis.

highlighted in our pseudo-code of Algorithm 3 (lines 27–40). They merge contiguous timetable blocks until there is only one block remaining. In this case, the last *ET* thread running computes the final statistics and copy them in the result table for the writing-back thread, which generates the output file. Since all the couples involved are independent of one another, the thread pool performs a parallel pattern (line 28). In this way, as more threads are involved in this pool, the speedup increases linearly, as shown in the experimental section.

VII. DESIGN OPTIMIZATIONS

As previously stated, we paid great attention to how to balance the different pipeline stages, avoid useless thread contention and reduce memory overheads as much as possible. To reach these targets, we heavily optimize the program using appropriate C and C++ functions and efficient low-level system calls:

- To have a very efficient signal manipulation, we adopt a fast direct access table with one entry for each signal. Moreover, we store only strictly required information to save memory space. For example, we optimize memory allocation by avoiding padding, and we also convert information to its binary format when this is memory efficient and does not slow down the computation. We also avoid using pointers and extra information (related to C++ collection libraries) as long as possible.
- We minimize I/O waiting times due to lazy (on-demand) loading and storing operations. We adopt memory mapping capabilities and binary block-based I/O operations. We also trim the block size with the different testing metrics to improve efficiency and minimize memory usage in all critical situations. We finally extend these features with multiple-buffering techniques based on the producer and consumer paradigm.
- We minimize thread-contention using local-accessible data structure on all Abstract Data Types (ADTs) are accessed with a high frequency. On the remaining ADTs, the ones accessed relatively at low frequency, we implement standard critical sections protection strategies using the more appropriate paradigm in each situation.

Overall, the parameters that have more influence on the performances of our applications are the size of the file partitions, the number of working threads in each phase, and the number of time frames present in the source VCD file. Partitions have to be large enough to be I/O efficient. Increasing their size also prevents the reading threads *RT* from iterating too often on the queue Q_1. Anyway, as each partition resides in the main memory, the quantity of memory used increases with the dimension of the file partitions. The Discovery thread leverages these two aspects by selecting the file partition size at the beginning of the entire process. We can adjust the memory used, setting the level of parallelism of the parsing pool *PT*. When we run more Parse threads, we increase the quantity of memory used, as we have more partitions stored in the main memory simultaneously. The number of threads also strongly influences the time the threads stall on the different stages. In the experimental section, we perform an accurate analysis of the execution and stall times of the different stages using different levels of parallelism.

The partition size also has substantial implications on the elaboration phase. When we manipulate different file partitions with different threads in parallel, we collect signal statistics (e.g., timings) randomly placed into different time frames. Unfortunately, at the end of the process, each signal should be analyzed sequentially in time. Thus, when we analyze in parallel a more significant number of small partitions, we must increase the effort finally required to reconstruct any time-dependent information properly. Moreover, notice that the discovery phase is also helpful to make partition starting synchronously with simulation time frames. If we

Algorithm 3 Pseudo-Code for the Multiple Point Stress Analysis.

```
MultiplePointStressAnalysis ()
Allocate table blocks
Init queues Q1, Q2, Q3, and Q4
insert table references in Q3
Init thread DT
Init thread pools RT, PT, and ET
Init thread WT
synch_barrier (wait for all threads to terminate)

DT (min_time)
while (read_file() != EOF) do
   insert signal name in signal table
   define partition limits (time difference < min_time)
   enqueue (limits, Q1)
end while

RT ()
{See Algorithm 1}

PT ()
for each thread in PT in parallel do
   while (Q2 ≠ ∅ AND Q3 ≠ ∅) do
      partition = dequeue (Q2)
      block = dequeue(Q3)
      while (partition is not parsed completely) do
         {s, t} = get toggle activity (signal s, time t)
         update (block, {s, t})
      end while
      enqueue (block, Q4)
   end while
end for

ET ()
for each thread in ET in parallel do
   while (Q4 ≠ ∅) do
      block1 = dequeue (Q4)
      if (block1 is last block) then
         update (result table, block1)
      else
         block2 = dequeue (Q4)
         block = merge (block1, block2)
         enqueue (block, Q4)
         enqueue (free block, Q3)
      end if
   end while
end for

WT ()
{See Algorithm 1}
```

had partitioned the file blindly, we would have obtained three different possible situations:

- The thread finds the beginning of a new time frame on the first line of its partition.
- The thread does not find the beginning of a new time frame on the first line of its partition, but it finds it inside it.
- The thread does not find the beginning of a new time frame in its partition at all.

To avoid the second and third situations, we force the discovery thread to store partitions into the queue Q_1 starting synchronously with simulation times. Anyhow, as partitions are not manipulated sequentially, each parsing thread does not know the initial value of the signals it must analyze. Each parsing thread considers the initial values of all signals as undefined to reconstruct the correct toggle activity. Then, when the toggle activity collected along different partitions is concatenated to find the overall behavior, the time information is reconstructed, managing all toggle activities sorted in time. In this way, it is possible to reconstruct the correct time sequence for each signal collecting single or multiple point metrics.

VIII. EXPERIMENTAL RESULTS

In this section, we show the effectiveness and scalability of our tool in terms of computation time and memory usage. More specifically, as we focus on multi-thread applications, we report the wall-clock time and the central memory used by all threads to perform our analysis. We compare these values with the ones gathered from the purely sequential version. Time and memory usage are measured with a third-party profiling tool to be as fair as possible. We limit the maximum central memory to 120 GBytes. Moreover, we run each experiment 5 times, and we present average data on all these executions. Tests have been performed on a machine equipped with a CPU AMD EPYC 7301, including 16 cores at 2.2 GHz with hyperthreading and 128 GBytes of RAM. All software versions run under Linux CentOS 7.

A. BENCHMARKS

The considered target device is the 40 nm Automotive Microcontroller described by Appello *et al.* [13]. The SoC includes three dual-issue processors connected to a set of peripheral cores. Overall, it includes about 28 million circuit nodes in the logic parts. The chip is equipped with a configurable scan chain, and all features to perform a BI stress by operating on the scan inputs. Like many other devices available in the automotive market, it requires a BI step and must undergo several hours of electrical stress to exacerbate all possible faulty behavior. For this reason, it is essential to evaluate all significant stress metrics before production. It is common to evaluate patterns generated by an ATPG engine (targeting some coverage, i.e., delay faults to ensure node toggling) and streams of bits generated on the fly by the tester logic. In our experimental analysis, we considered the stress metrics analyzed in Sections IV, V, and VI as ATPG and fault can hardly compute them simulation engines. In our experiments, we consider the whole circuit, measuring values and computing statistics on over 28 million nodes. In all cases, our tool can generate compact and extended reports.

TABLE 1. Single point stress timing analysis. Running times for the different pipeline stages of our chain, with an increasing number of threads for the smaller VCD file, i.e., the one of 10 GBytes. All times are reported in seconds. – means that the data is meaningless in that experiment.

#THREADS	DISCOVER	READ	PARSE	WRITE-BACK	TIME SUM	WALL-CLOCK
Sequential	–	–	–	–	–	231
1	0	7	221	0	229	222
4	0	6	63	0	70	63
8	0	8	33	0	41	33
16	0	11	23	0	35	23
32	0	20	20	0	40	21
64	0	45	12	0	58	47

The files from 10 to 57 GBytes are produced by pattern simulation; the ones from 80 to 243 GBytes are generated performing functional analysis. The functional analysis leads to VCD files storing a more significant number of time frames, each one with a reduced toggle activity with respect to the ATPG pattern analysis. In general, this characteristic implies longer computational times for the entire Execute stage of the multiple point stress metric.

B. THE SINGLE POINT STRESS TIMING ANALYSIS

Table 1 reports the wall-clock times of all pipeline stages required to compute the toggle activity on the 10 GBytes VCD file. For the toggle activity (compact) statistics, we produce a report including the overall amount of toggling events for every signal and the average toggling events over the whole set of considered signals. Table 1 also reports the sum of these times and the wall-clock times of the entire process. Please, notice that within the sequential version of the program, we do not have distinct phases, which are merged into the sequential file reading and parsing. Thus, for this implementation, we report only the total computation time. For all other cases, due to our pipeline implementation, the overall wall-clock time is quite close to the time required to solve the slower stage. On the contrary, the sum of the wall-clock times of all stages is usually more prominent than the processing time. We define the *pipeline efficiency* (P) of our application as:

$$P = \frac{Wall\ Clock}{Time\ Sum} \tag{1}$$

P provides a metric showing the efficiency of our pipeline approach, i.e., the smaller the value, the larger the time saved with respect to a hypothetical, fully sequential version of the program. The lowest bound of P is $1/N_{stages}$ for a perfectly balanced pipeline. We report the pipeline efficiency P in Table 3, but Table 1 already shows how our pipeline reduces the computational time. When we use a small number of threads, the most critical phase is represented by the Parse stage. On the contrary, when we increase the number of parsing threads in the thread pool PT, the most critical stage is the Read one. It is also noticeable that 32 parsing threads lead to the lowest computational time. Beyond this value, the advantages of the parallel approach are balanced by the computational overhead required to manage more threads. Anyway, the difference between 16, 32, and 64 parsing threads becomes smaller when processing larger files.

Table 1 shows that the most balanced configuration is the one with 32 parsing threads. This is true for most of the cases, except for the file of 179 GByte, for which the configuration with 16 parsing threads is about 50 seconds faster. Table 2 reports the running times on VCD files of increasing size using the configuration with 32 parsing threads with this exception in mind. The larger performance gap is the one between the file of 57 and the one of 80 GBytes. As already observed, this is due to the different nature of the VCD files, which have fewer time frames with more signal variations up to 57 GBytes, and more time frames with a reduced toggle activity beyond 80 GBytes. In any case, our application scales well in both scenarios. For the sake of readability, Figure 10 plots the timing trends with different thread configurations in two and three-dimensional graphics. The two plots better show the linear behavior of the total elapsed time with increasing VCD files. As suspected, our analysis also shows how the Read stage is the most expensive one. The Read time is pretty close to the entire process wall-clock time, clearly showing the performance of our pipeline is strictly related to the slower phase. For this metric computation, the Discover and the Write-Back times are irrelevant in all experiments. The first one is unimportant because, in this phase, our tool simply provides to the Read phase the limits to partition the VCD file nicely. The last one is insignificant as in the stress test analysis during the Write-Back stage, and we store a minimal amount of statistical information on disk. Using a faster disk (such as an SSD) would likely improve the performance of both the Parse and the Reading stages, providing shorter execution times and better balance.

Table 3 focuses on waiting times, pipeline efficiencies, and speedups, again reporting data for VCD files of increasing size. The waiting times of the essential stages (i.e., Read and Parse) are computed as the sum of the idle times of all threads performing those phases. Reading threads wait mainly for parsing threads to manipulate the file chunks that have been uploaded, whereas parsing threads mainly wait for reading threads. Our data confirm that the most critical stage is the reading one, as it forces the parse phase to wait for a good percentage of the overall wall-clock time. In the last two columns, the table reports the speedups obtained for the Parses stage and for the entire process. The parsing stage has almost a linear speedup (14.35 on average with 32 threads) for all VCD files. On the contrary, the speedup of the entire

TABLE 2. Single point stress analysis. Running times for the different pipeline stages, with 32 parsing threads (our most balanced configuration) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or in hours (when explicitly stated).

VCD Size [GBytes]	Single threaded Time sum	Pipeline + Parallel + Opt					
		Discover	Read	Parse	Write-Back	Time Sum	Wall-Clock
10	229	0	20	20	0	40	21
20	470	0	34	34	0	69	36
38	1,132	0	123	45	0	169	126
57	2,656	0	324	129	0	454	327
80	3,634	0	731	149	0	880	735
100	3,985	0	928	218	0	1,146	932
124	5,461	0	1,156	258	0	1,415	1,162
140	5,680	0	1,305	283	0	1,589	1,311
156	4,438	0	1,459	346	0	1,805	1,463
179	1.77 h	0	1,661	391	0	2,053	1,668
207	1.99 h	0	1,867	481	0	2,349	1,875
243	2.49 h	0	2,286	565	0	2,852	2,296

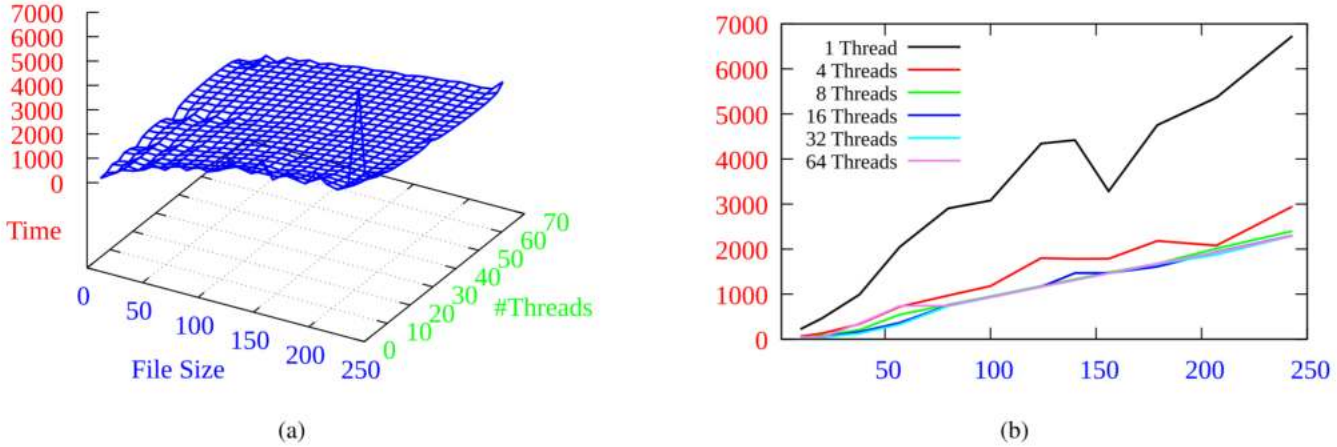

FIGURE 10. Single point stress analysis. Overall Wall-Clock Times as a function of the VCD file size and the number of threads. Three-dimensional plot (a) and corresponding two-dimensional graph (b).

TABLE 3. Single point stress analysis. In-depth analysis of our approach: Waiting times for all threads of the two most critical phases (read and parse stages), pipeline efficiency P (Equation 1), and speed-ups for the Parse phase and the entire process. All times are reported in seconds.

VCD Size [GBytes]	Waiting Times		Pipeline Efficiency	Speed-ups	
	Read Stage	Parse Stage		Parse Stage	Entire Process
10	0	1	0.54	10.90	10.50
20	0	1	0.52	13.25	13.05
38	0	45	0.75	21.85	8.94
57	0	197	0.72	15.72	8.11
80	0	585	0.83	19.46	4.94
100	0	713	0.81	14.07	4.27
124	0	903	0.82	16.81	4.70
140	0	1027	0.83	15.58	4.33
156	0	1118	0.81	9.47	3.03
179	0	1276	0.81	12.12	3.82
207	0	1392	0.80	11.11	3.82
243	0	1729	0.81	11.88	3.90

process is drastically limited by the most expensive Read phase. During this stage, we face a substantial limitation in improving the speedup due to the physical constraints of the hardware architecture on which our application is executed.

Memory usage is almost constant for all thread configurations, and it mostly depends on the number of signals to analyze and the number of file chunks present in the Parse queue. Overall, memory is not critical in this type of analysis. For this reason, we discuss memory issues more accurately for the following metrics, for which memory has a more substantial impact on the performances.

C. THE SINGLE POINT FULL TIMING ANALYSIS

In the single point full timing analysis, we are interested in the entire toggle activity of all signals, including all of the toggles and the time frames at which they happened.

Table 4 reports the wall-clock times of all pipeline stages required to compute the toggle activity on the file of

TABLE 4. Single point full timing analysis. Running times for the different pipeline stages, with an increasing number of threads, for the smaller VCD file, i.e., the one of 38 GBytes. All times are reported in seconds. – means that the data is meaningless in that experiment.

#THREADS	DISCOVER	READ	PARSE	WRITE-BACK	TIME SUM	WALL-CLOCK
Sequential	–	–	–	–	–	1793
1	452	172	948	219	1792	1047
4	140	214	289	227	871	329
8	130	176	164	234	687	302
16	390	609	82	128	1211	662
32	124	185	35	215	561	293
64	393	665	22	171	1253	677

TABLE 5. Single point full timing analysis. Running times for the different pipeline stages, with 32 threads (our most efficient configuration) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or hours (when explicitly stated).

VCD SIZE [GBYTES]	SINGLE THREADED TIME SUM	PIPELINE + PARALLEL + OPT					
		Discover	Read	Parse	Write-Back	TIME SUM	WALL-CLOCK
10	355	18	50	9	62	141	83
20	711	35	92	17	109	255	150
38	1792	124	185	35	215	561	293
57	5094	654	735	55	487	2933	916
80	5.50 h	2.06 h	1.92 h	48	107	4.03 h	2.08 h
100	6.95 h	2.65 h	2.47 h	58	151	5.17 h	2.67 h
124	6.40 h	2.92 h	2.72 h	72	295	5.74 h	2.94 h
140	8.39 h	3.59 h	3.34 h	78	245	7.11 h	3.61 h
156	9.21 h	3.71 h	3.45 h	88	285	7.26 h	3.72 h
179	10.30 h	4.66 h	4.34 h	102	435	9.14 h	4.68 h
207	11.77 h	4.58 h	4.26 h	118	423	9.00 h	4.60 h
243	14.12 h	5.50 h	5.12 h	140	618	10.84 h	5.52 h

TABLE 6. Single point full timing analysis. In-depth analysis of our approach: Waiting times for the two most critical phases (the Read and Parse stages), pipeline efficiency, and speedups for the Parse phase and the entire process. All times are reported in seconds or hours (when explicitly stated).

VCD SIZE [GBYTES]	WAITING TIMES		PIPELINE EFFICIENCY	SPEED-UPS	
	Read Stage	Parse Stage		Parse Stage	Entire Process
10	26	16	0.59	12.61	2.80
20	49	14	0.59	12.46	2.72
38	65	24	0.52	26.55	6.10
57	162	616	0.47	36.67	5.56
80	10	2.05 h	0.52	31.12	2.65
100	18	2.64 h	0.52	32.71	2.61
124	36	2.90 h	0.51	28.99	2.18
140	29	3.57 h	0.51	29.87	2.32
156	52	3.67 h	0.51	29.47	2.48
179	54	4.62 h	0.51	28.85	2.20
207	50	4.54 h	0.51	28.99	2.56
243	66	5.44 h	0.51	28.71	2.56

38 GBytes, the sum of these times, and the wall-clock time of the entire application. We selected this VCD file because it shows a typical performance profile, whereas for smaller files, eight parsing threads provide the best performance. From the table, it is noticeable that with 16 and 64 parsing threads, the time for the Discover and Read stages significantly increase. This is common to all other VCD files and, as mentioned in the introduction, verified on different runs performed after a considerable time gap. We have been unable to understand this behavior even if we fully suspect it is due to a combination of the characteristics of the VCD files and the hardware platform used to run the experiments.

Table 5 shows all running times collected with the most balanced version, i.e., the one using 32 parsing threads. Notice that computing the full analysis is between 4 and 10 times slower than evaluating the single-point statistic metric. Anyway, as for this stress metric, our analysis scales well with increasing file size. For the sake of space, we avoid reporting the corresponding 2D and 3D plots. The time behavior is similar to the one analyzed in Section VIII-B considering the file size, with the functional simulation requiring more analysis time than the pattern simulation. For the full analysis, the Discover stage is the most time-consuming. This is because the Discover stage performs a more complex file parsing than in the last metric to make each new file partitions start with a time declaration.

Table 6 reports the measured waiting times for the Read stage and the Parse phase is waiting for each other. We minimized those times during our trimming process, balancing the different pipeline stages. Thus, even if the waiting

times keep increasing with the file size (except for the file of 207 GBytes), the pipeline efficiency improves as it decreases to 0.51 for the larger files. As for the single point stress timing analysis, a faster storage unit would likely reduce all Parse and Reading times. Our data also shows that our pipeline process is more balanced with the full analysis than with the stress metric. Excluding the two smaller files, the speedup we obtained for the Parse stage is relatively steady and has a value of about 29 with 32 threads. The overall speedup varies from 2.18 to 6.10 times, and it is mainly limited by the performances of the Discover stage, in which we parse the file sequentially.

For the full analysis, memory usage strongly depends on the number of signals. Moreover, in the case of smaller files, the writing queue Q4 tends to be filled with more data to be saved since a larger number of changes is read for each time frame. On larger files, the data queue tends to use less memory as there is a lower number of changes for each time frame. For this type of analysis, memory usage is more critical than for the previous metric. To keep the amount of used memory under control, we store intermediate information into temporary files on the external memory unit as soon as they become useless for the following computations. This strategy, implemented by the Write-Back stage, maintains the memory usage under control.

D. THE MULTIPLE POINT STRESS METRIC

In the adjacent nodes strain stress analysis, we consider multiple points and evaluate the number of times adjacent nodes take the opposite values for a user-defined amount of time. Our experiments selected 20 million pairs of adjacent nodes to analyze the files of size smaller than 60 GBytes. For larger files (starting from 80 GBytes), we consider 2 million pairs to avoid extremely long running times. We also set the minimum time to 1 ns, focusing on a very bad scenario since almost all signal changes need to be considered.

Table 7 shows the running times for all pipeline stages for the VCD file of 10 GBytes. In this case, we present our results varying both the number of parsing and executing threads. The first line reports the result of the sequential execution. As for the other metrics, the wall-clock time is shorter than the sum of all times because of our pipeline architecture. In the configuration with 1 Parse and 64 Execute threads, we drastically reduce the time required by the Execute phase, the more critical one. This leads to the parsing stage being the slowest along our pipeline. With 32 Parse and 32 Execute threads (the fourth line of the table), we better balance these two expensive stages. However, this leads to the execute stage being the slowest one. This situation does not change even when we further increase the number of Execute threads. Even with 32 Parse and 64 Execute threads (line 5 of the table), the Execute phase is the slowest one, with a relatively small improvement with respect to the previous case. Since the execution stage represents the critical path, we can reduce the number of parsing threads. Thus, the last configuration, adopting 16 Parse and 64 Execute threads are the fastest due to an improved pipeline balance and a smaller thread overhead.

Table 8 shows the improvements of our fastest configuration over a set of VCD files of increasing size. Notice that the multiple point stress metric is about 40 times slower than the single point stress metric and about four-time slower than the full analysis. The sequential version needs extremely long times with files larger than 60 GBytes. Our fastest version shows speedups of over two orders of magnitude. Since the duration of this analysis is strongly dependent on the number of time frames analyzed, for the files beyond 80 GBytes, our tool shows much longer computation times. For this metric, the most critical stage is the Discover phase. This criticality is partially due to the number of signal pairs considered. We can also see that with the increasing number of time frames, the Execute and the Discover times increase significantly. Moreover, for large files, the parsing stage tends to be slower than the execution stage. This is not an issue since the Discovery stage is effectively on the critical path, and it tends to be much slower than the parsing stage.

Table 9 shows the waiting times, the pipeline efficiency, and the speedups for all VCD files. As in the previous analysis, the pipeline efficiency P (defined in Section VIII-B) provides a valuable hint of how well the pipeline is working, and its value goes down to 0.35-0.36 for the larger files. Speedups are very significant when we compute this metric for both the Execute stage and the entire process, as they reach a value of about 40 (with 16 Parse and 64 Execute threads).

Memory occupation, in the multiple point metric computation, strongly depends on the number of signal pairs considered. To avoid allocating too much memory, we use a pool storing all signal pair changes. The memory used is strongly influenced by the size of this pool, which also depends on the number of parsing threads since the pool's size depends on it to avoid deadlocks. To be more precise, the pool needs to be as large as the number of threads if we want a performance improvement. We selected it to be 2.5 times larger than the number of parsing threads since any size beyond this value does not allow significant time improvements.

E. SPEEDUPS, COMPLEXITY, AND SCALABILITY

For the sake of completeness, Figure 11 plots the speedups already reported in Tables 3, 6, and 9. Blue graphics illustrate the speed-ups gathered for the single point analysis, red ones for the single point full analysis, and the green ones for the multiple point stress analysis. For each of these, we consider the contributions obtained thanks to pipelining (Figure 11) and parallelism (Figures 11b and 11c), separately. We compute the speedups due to pipelining as (1/(pipeline efficiency)), where the pipeline efficiency is reported in Tables 3, 6, and 9. Figures 11b take into consideration the most expensive phase, whereas Figure 11c considers the entire process. Overall, our speedups range from about 3 to over 40 for the entire process. Our data shows that the single point full analysis is the least parallelizable computation, whereas the most parallelizable one is the multiple point

TABLE 7. Multiple Point Stress Analysis. Running times for the different pipeline stages, with different thread configurations, for the smaller VCD file, i.e., the one of 10 GBytes. All times are reported in seconds. – means that the data is meaningless in that experiment.

#Threads		Pipeline Stages					Time Sum	Wall-Clock
Parse	Execute	Discover	Read	Parse	Execute	Write-Back		
Sequential	Sequential	–	–	–	–	–	–	5884
1	1	15	36	1453	4372	0.0	5877	4385
1	64	17	43	1321	481	0.0	1863	1818
32	32	21	95	64	306	0.0	487	320
32	64	34	135	83	284	0.0	537	309
16	64	29	1451	152	269	0.0	596	292

TABLE 8. Multiple Point Stress Analysis. Running times for the different pipeline stages. with the 16/64 threads configuration (the most efficient one in our experiments) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or hours (when explicitly stated).

VCD Size [GBytes]	Single threaded time sum	Pipeline + Parallel + Opt						
		Discover	Read	Parse	Execute	Write-Back	Time Sum	Wall-Clock
10	5887	29	145	152	269	0	596	292
20	5.32 h	60	285	294	481	0	1121	524
38	12.32 h	118	526	564	868	0	2077	957
57	18.23 h	174	684	811	1374	0	3044	1473
80	391.36 h	9.04 h	4.53 h	6.50 h	5.81 h	0	25.89 h	9.02 h
100	524.45 h	10.62 h	5.14 h	7.64 h	6.74 h	0	30.14 h	10.71 h
124	528.31 h	12.76 h	6.34 h	9.15 h	7.96 h	0	36.21 h	12.84 h
140	587.56 h	14.56 h	7.55 h	10.54 h	9.77 h	0	42.42 h	14.67 h
156	716.79 h	15.61 h	7.41 h	11.20 h	9.68 h	0	43.91 h	15.70 h
179	801.82 h	17.72 h	8.39 h	12.70 h	11.06 h	0	49.87 h	17.80 h
207	957.45 h	19.86 h	9.81 h	14.21 h	11.79 h	0	55.67 h	19.94 h
243	1076.56 h	23.65 h	11.14 h	16.97 h	15.06 h	0	66.82 h	23.74 h

TABLE 9. Multiple Point Stress Analysis. In-depth analysis of our approach: Waiting times for all threads of the two most critical phases (read and parse stages), pipeline efficiency, and speed-ups for the Parse phase and the entire process. All times are reported in seconds or hours (when explicitly stated).

VCD Size [GBytes]	Waiting Times		Pipeline Efficiency	Speed-ups	
	Read Stage	Parse Stage		Execute Stage	Entire Process
10	72	128	0.49	16.24	20.12
20	166	218	0.47	32.59	36.51
38	335	381	0.46	42.68	46.29
57	684	653	0.48	37.27	44.53
80	4.33 h	2.61 h	0.35	42.32	43.13
100	5.26 h	3.06 h	0.36	45.45	49.32
124	6.13 h	3.67 h	0.35	39.61	41.76
140	6.69 h	4.09 h	0.35	38.78	40.93
156	7.83 h	4.47 h	0.36	44.33	46.23
179	8.90 h	5.07 h	0.36	42.96	45.62
207	9.55 h	5.69 h	0.36	45.46	48.73
243	11.9 h	6.73 h	0.36	44.89	45.98

stress test analysis. This is mainly due to the fact that for the single point full analysis, the reading and the writing stage take most of the time, and these steps are intrinsically sequential with strong architectural and hardware limitations. If we do not consider reading and writing costs and just focus on the core computation, the profile for Figure 11c would be quite close to the one in Figure 11b with speedups ranging from 10 to 50.

The results gathered with our profiling tool can be completed with an analysis of Amdahl's law to understand the theoretical limit of our speedups. We concentrate on the single-point stress timing analysis. On the one hand, for this metric, our profiler could compute that our tool runs in parallel for about 75% of the overall wall-clock time. Variations of this value are very limited with the different VCD files. On the other hand, Amdahl's law is described by the following equation:

$$S = \frac{1}{(1-p) + \frac{p}{N}}$$

where S is the theoretical speedup of the execution of the whole task, p is the proportion of execution time that benefits from parallelization of degree N. Thus, as p = 0.75, we can compute a theoretical speedup of 3.36 with 16 threads. This observed speedup does not significantly differ from the ones shown by the experimental analysis of Table 3.

However, it is difficult to estimate the exact degree of parallelization using Amdahl's law within our pipeline. Stalls are not deterministic, and they often depend on the size of the VCD file. Then, we also used the pipeline efficiency, as defined in Section VIII-A, to analyze the impact of our pipeline on the speedup of the entire process.

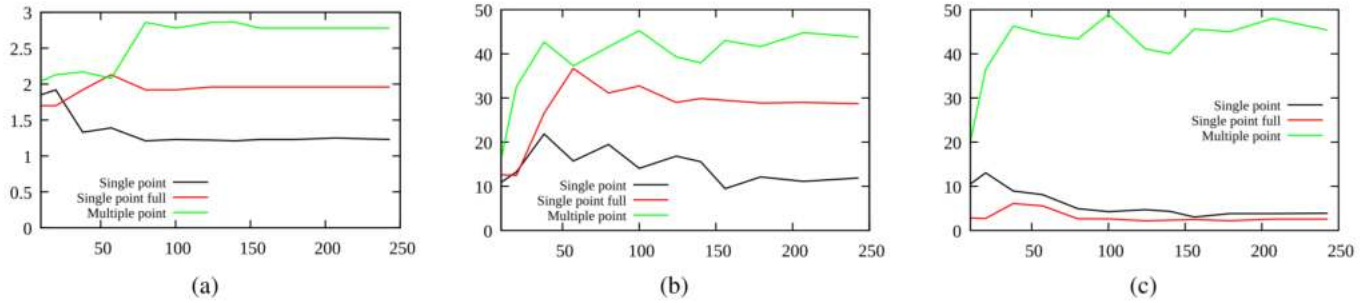

FIGURE 11. The three graphics report the speed-ups obtained using (only) our pipelining approach (Figure (a)), the ones gathered for the slower phase of each metric (Figure (b)), and the ones measured for the entire process (Figure (c)). For each graphic, we report the speedups for the single point (blue), single point full (red) and multiple points (green) analysis.

IX. CONCLUSION AND FUTURE WORKS

We propose a methodology to reduce time and memory costs to analyze BI-related metrics that current ATPG tools cannot evaluate.

We work in a framework in which we firstly generate simulation dumps using the VCD format, and then we use a post-processing phase to gather detailed or statistical results on the simulation phase. Our methodology is based on three main ingredients, namely pipelining, parallelization, and low-level system-call optimizations.

Our experimental analysis focuses on an automotive device, including about 28 million gates and generating simulation traces occupying up to almost 250 GBytes. On this device, we have been able to reduce the evaluation time by more than two orders of magnitude compared to the original sequential tool. We also keep the memory usage under control with the right mixture of the three ingredients previously introduced.

Among the future works, we would like to mention the necessity to push forward our analysis on even larger VCD files, close or beyond to 1 TBytes. This would imply long experimental sessions on powerful and remote server units. Moreover, one of the requirements we may foresee is to enrich our application with some level of auto-trimming capability, at least sufficient to leverage the different pipeline stages using an appropriate level of parallelization.

REFERENCES

[1] H. Fujiwara and S. Toida, "The complexity of fault detection problems for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-31, no. 6, pp. 555–560, Jun. 1982.

[2] R. K. Roy, T. M. Niermann, J. H. Patel, J. A. Abraham, and R. A. Saleh, "Compaction of ATPG-generated test sequences for sequential circuits," in *Proc. IEEE Int. Conf. Comput.-Aided Design (ICCAD)*, Jan. 1988, pp. 382–385.

[3] C.-J. J. Chang and T. Kobayashi, "Test quality improvement with timing-aware ATPG: Screening small delay defect case study," in *Proc. IEEE Int. Test Conf.*, Oct. 2008, p. 1.

[4] S. Biswas and B. Cory, "An industrial study of system-level test," *IEEE Des. Test*, vol. 29, no. 1, pp. 19–27, Feb. 2012.

[5] Y. K. Malaiya and R. Narayanaswamy, "Modeling and testing for timing faults in synchronous sequential circuits," *IEEE Des. Test Comput.*, vol. 1, no. 4, pp. 62–74, Nov. 1984.

[6] K.-H. Tsai, R. Guo, and W.-T. Cheng, "A robust automated scan pattern mismatch debugger," in *Proc. 17th Asian Test Symp.*, Nov. 2008, pp. 309–314.

[7] K. Marcinek and W. A. Pleskacz, "AGATE—Towards designing a low-power chip multithreading processor for mobile software defined radio systems," in *Proc. IEEE 15th Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2012, pp. 26–29.

[8] *IEEE Standard for Verilog Hardware Description Language*, IEEE Standard 1364-2005 (Revision of IEEE Standard 1364-2001), 2006, pp. 1–590.

[9] A. Vassighi, O. Semenov, M. Sachdev, A. Keshavarzi, and C. Hawkins, "CMOS IC technology scaling and its impact on burn-in," *IEEE Trans. Device Mater. Rel.*, vol. 4, no. 2, pp. 208–221, Jun. 2004.

[10] J. H. Cha and M. Finkelstein, "Burn-in for systems operating in a shock environment," *IEEE Trans. Rel.*, vol. 60, no. 4, pp. 721–728, Dec. 2011.

[11] *The Apache Hadoop MapReduce Guide*. Accessed: Oct. 2, 2022. [Online]. Available: https://hadoop.apache.org/

[12] D. Appello, P. Bernardi, A. Calabrese, S. Littardi, G. Pollaccia, S. Quer, V. Tancorre, and R. Ugioli, "Accelerated analysis of simulation dumps through parallelization on multicore architectures," in *Proc. 24th Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2021, pp. 69–74.

[13] W. Ruggeri, P. Bernardi, S. Littardi, M. S. Reorda, D. Appello, C. Bertani, G. Pollaccia, V. Tancorre, and R. Ugioli, "Innovative methods for burn-in related stress metrics computation," in *Proc. 16th Int. Conf. Design Technol. Integr. Syst. Nanosc. Era (DTIS)*, Jun. 2021, pp. 1–6.

[14] C. He, "Advanced burn-in—An optimized product stress and test flow for automotive microcontrollers," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2019, pp. 1–6.

[15] Z. Tokei, P. Roussel, M. Stucchi, J. Versluijs, I. Ciofi, L. Carbonell, G. P. Beyer, A. Cockburn, M. Agustin, and K. Shah, "Impact of LER on BEOL dielectric reliability: A quantitative model and experimental validation," in *Proc. IEEE Int. Interconnect Technol. Conf.*, Jun. 2009, pp. 228–230.

[16] P. Bernardi and M. S. Reorda, "An novel methodology for reducing SoC test data volume on FPGA-based testers," in *Proc. Design, Autom. Test Eur.*, Mar. 2008, pp. 194–199.

[17] D. Appello, P. Bernardi, G. Giacopelli, A. Motta, A. Pagani, G. Pollaccia, C. Rabbi, M. Restifo, P. Ruberg, E. Sanchez, C. M. Villa, and F. Venini, "A comprehensive methodology for stress procedures evaluation and comparison for burn-in of automotive SoC," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 646–649.

[18] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Reading, MA, USA: Addison-Wesley, Sep. 2004.

[19] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[20] M. E. Lalami, D. El-Baz, and V. Boyer, "Multi GPU implementation of the simplex algorithm," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Sep. 2011, pp. 179–186.

[21] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 117–128, doi: 10.1145/2145816.2145832.

[22] G. Lowe, "Concurrent depth-first search algorithms based on Tarjan's algorithm," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 2, pp. 129–147, Apr. 2016, doi: 10.1007/s10009-015-0382-1.

[23] A. Malapert, J.-C. Régin, and M. Rezgui, "Embarrassingly parallel search in constraint programming," *J. Artif. Intell. Res.*, vol. 57, pp. 421–464, Nov. 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3176748.3176758

[24] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, and J. Trimble, "Sequential and parallel solution-biased search for subgraph algorithms," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, L.-M. Rousseau and K. Stergiou, Eds. Cham, Switzerland: Springer, 2019, pp. 20–38.

[25] A. Garbo and S. Quer, "A fast MPEG's CDVS implementation for GPU featured in mobile devices," *IEEE Access*, vol. 6, pp. 52027–52046, 2018.

[26] S. Quer and A. Calabrese, "Graph reachability on parallel many-core architectures," *Computation*, vol. 8, no. 4, p. 103, Dec. 2020. [Online]. Available: https://www.mdpi.com/2079-3197/8/4/103

D. APPELLO received the degree in electronics engineering from the Università di Pavia, Pavia, Italy. He is currently a Product-Engineering Director for the automotive digital products of STMicroelectronics, Agrate Brianza, Italy, where he is involved in testability and testing. He is active within TTTC and TTEP groups of the IEEE.

PAOLO BERNARDI (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science, in 2002 and 2006, respectively. He is currently an Associate Professor with the Politecnico di Torino, where he works with the Electronic CAD and Reliability Research Group. His current interests include system-on-chip test and reliability, especially in the direction of high-quality automotive devices. He is the General Chair of the Test Technology Educational Program (TTEP) and the Program Chair of the Automotive Reliability and Test (ART) Workshop held in conjunction with the International Test Conference. He was recently acting as a Topic Chair of the European Test Symposium (ETS), the Design and Diagnosis of Electronic Circuits Symposium (DDECS), and the International On-Line Test Symposium (IOLTS).

ANDREA CALABRESE (Graduate Student Member, IEEE) graduated in computer science from the Politecnico di Torino, in 2020. He is currently pursuing the Ph.D. degree with the Politecnico di Torino. His main research interests include software optimization and parallel algorithms.

G. POLLACCIA received the degree in electronics engineering from the Università di Palermo, Palermo, Italy. He is currently a Burn-In Test Engineer for the automotive digital products with STMicroelectronics, where he is involved in testability and testing.

STEFANO QUER (Member, IEEE) received the M.S. degree in electronic engineering, in 1991, and the Ph.D. degree in computer engineering, in 1996. He has been a Visiting Faculty with the Department of Electronic Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, USA. He has been an Intern with the "Advanced Technology Group," Synopsys Inc., Mountain View, CA, USA, and with the Alpha Development Group, Compaq Computer Corporation, Shrewsbury, MA, USA. He has been a Compaq Computer Corporation Consultant. He is currently a Professor with the Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy. His main research interests include systems and tools for CAD for VLSI, formal methods for hardware and software systems, and embedded systems. Other activities focus on the development of sequential and concurrent algorithms and on optimization techniques able to achieve acceptable solutions with limited resources.

V. TANCORRE received the B.S. degree in electrical engineering from the Politecnico di Bari. He is currently a Design-to-Test Engineer with the Testing Technology Center, STMicroelectronics. His research interests include test-related process monitoring using diagnostic solutions for memories and unstructured logic.

R. UGIOLI graduated in electronic engineering from the Politecnico di Torino, in 1990. He gathered a 30 years experience in design for testability and silicon validation of ASIC devices. Since 2018, he has been a Senior Hardware Product Engineer in charge of system level test equipment for volume production tests of ADAS devices at STMicroelectronics Automotive Division.