IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 30, 2018, accepted December 20, 2018, date of publication December 28, 2018,
date of current version January 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2890112

P4-TPG: Accelerating Deterministic Parallel Test Pattern Generation by Preemptive, Proactive, and Preventive Schedulings

LOUIS Y.-Z. LIN, (Student Member, IEEE), CHARLES CHIA-HAO HSU, AND CHARLES H.-P. WEN, (Member, IEEE)
Department of Electrical and Computer Engineering, National Chiao Tung University, Hsinchu 300, Taiwan
Corresponding author: Louis Y.-Z. Lin (louislin.eed99g@g2.nctu.edu.tw)

ABSTRACT According to the prior research, a deterministic parallel test pattern generation (TPG) engine was realized and generated the same test pattern set the serial automatic test pattern generation does during acceleration. However, for retaining the determinism, tremendous idle time is observed when different tasks (either dependent or independent) were synchronized among threads. Therefore, a new deterministic parallel TPG engine called P4-TPG is developed and incorporates preemptive, proactive, and preventive schedulings to further save/reuse the idle time during acceleration. In P4-TPG, preemptive scheduling first modifies the thread flow and brings forward, as many latter tasks as possible, to the idle time. Next, proactive scheduling inserts prospective TPG tasks of unprocessed faults to the remaining idle time and increases the overall utilization of threads. Last, preventive scheduling dynamically skips faults incompatible with the working pattern per thread and shortens the fault list during fault compaction. The experimental results show that P4-TPG not only generates the same test pattern set as the serial TPG does but also achieves averagely 10.36× speedups, is 96.6% better than the prior research, using 12 threads on 18 benchmark circuits.

INDEX TERMS Parallel ATPG, test inflation, deterministic, dynamic compaction.

I. INTRODUCTION

Continued growth in the size and complexity of very-large-scale integration (VLSI) systems is a fueling demand for faster automatic test pattern generation (ATPG) for testing. Conventional ATPG algorithms performing on a single processor now run into a bottleneck, incapable of efficiently generating tests required for modern VLSI designs. The rapid development of multi-core processors has opened the door of ***parallel computing*** as resolution for scaled designs. The communication protocol determined the classification of parallel computing architectures as *shared-memory systems* and *message-passing systems*. Both of these parallel computing systems provide additional computing power for ATPG.

Parallel ATPG can be classified into *non-deterministic* and *deterministic* ones. A non-deterministic parallel ATPG does not need the consistency after test pattern generation sets. Most of previous parallel ATPG algorithms [1]–[13] belong to this category and parallelize three core operations in ATPG, which are *test pattern generation* (TPG), *fault compaction* (FC) and *fault simulation* (FS), for achieving better speed-up. However, a non-deterministic parallel ATPG frequently runs into the problem of test inflation caused by race condition of multiple threads. According to [14], a 5.9% increase in pattern count can lead to a 100% increase in test cost per unit (under the worst-case scenario), not to mention the additional time and related pre-silicon effort. These aforementioned works remain suffering from test inflation; none of them can guarantee zero inflation on test pattern sets even if many of them have adopted different respective techniques to avoid repeated detection of the same faults.

Unlike non-deterministic ones, deterministic parallel ATPGs always generate **the same** test pattern set regardless of the processing time and the number of threads. Yeh *et al.* [15] proposed a deterministic parallel TPG, which is termed circular pipeline processing based parallel TPG (CPP-TPG). CPP-TPG achieves a near linear speed-up (with respect to the thread count) and produces the same test pattern set regardless of required computing resources. However, this method still results in inflation proportional to the number of threads on final test pattern sets. The main cause comes from that CPP-TPG employs *static pattern compaction* in a circular fashion to avoid the race condition of multiple threads, but

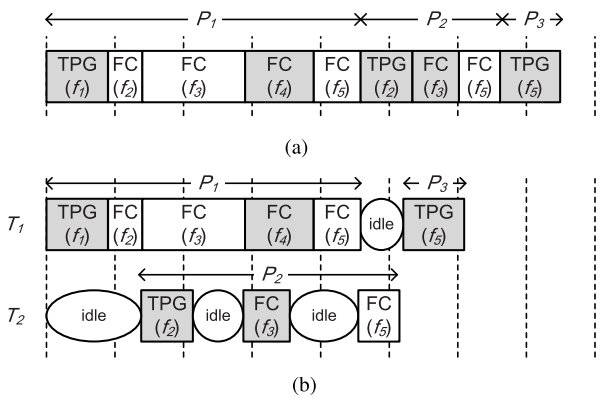

FIGURE 1. Ordering tasks in the serial ATPG and SDC-TPG [16]. (a) serial ATPG. (b) SDC-TPG.

produces a different compaction order, not in concordance with the one in the serial ATPG. Extra test patterns are thus generated.

To further guarantee the same compaction order as the serial ATPG does, Chang *et al.* [16] imposed *dynamic fault compaction* and proposed the corresponding zero-inflation parallel test pattern generator named SDC-TPG. More specifically, for maintaining determinism with a consistent compaction order, SDC-TPG utilizes a synchronization mechanism to consider dependencies of task execution (i.e. when to perform TPG, FC or FS of faults). As a result, regardless of computing resource in use, SDC-TPG always compacts faults in the same order as the serial ATPG does under dynamic fault compaction and thus produces the same test pattern set.

Fig. 1 shows two orders of tasks for the serial ATPG and SDC-TPG. P, T and f denote a pattern, a thread and a target fault, respectively. Gary boxes denote a target fault that can be detected in one task (either TPG or FC), whereas white ones denote those that fail to detect target faults in the corresponding tasks. In Fig. 1(a), all tasks (either TPG or FC) are performed serially until three patterns are successfully generated. As one can see, the overall runtime is the sum of the time on TPG and FC processes related to these three patterns. Fig. 1(b) shows the related tasks in SDC-TPG of three patterns (P_1, P_2 and P_3) with two threads (T_1 and T_2). T_1 takes charge of the related tasks for P_1 and P_3 while T_2 is responsible for P_2. As one can see, tasks in each pattern follows the exactly same order as the one in the serial ATPG. Moreover, tasks in one thread can start only if all dependent tasks in the other thread finish. For example, TPG(f_2) in T_2 executes after FC(f_2) in T_1 and FC(f_3) in T_2 executes after FC(f_3) in T_1. The similar situation happens between P_2 in T_2 and P_3 in T_1. As a result, SDC-TPG can effectively shorten the total runtime and generate the same result (i.e. the same test pattern set) as the serial ATPG does.

However, for imposing such synchronization for *dynamic fault compaction* in SDC-TPG, much idle time needs to be paid and thus the computing resources are wasted. Fig. 2 shows two cases where the idle time can be further **saved** or **reused** in SDC-TPG. Fig. 2(a) shows the case of **saving** the idle time. For example, in SDC-TPG, the start of FC(f_3) in T_2 waiting for the finish of FC(f_3) in T_1 is completely not necessary. To ensure the correctness, FC(f_3) in T_2 can start early, but needs to commit itself after the finish of FC(f_3) in T_1 once it succeeds. If it fails to detect the target fault, the next task on the same thread can start immediately without any problem. Fig. 2(b) shows the case of **reusing** the idle time. If much idle time among tasks is saved from starting latter tasks earlier in SDC-TPC, the remaining idle time can be further reused for TPG task of latter faults. In Fig 2(b), the idle time on T_2 between FC(f_3) and FC(f_5) can be used for the next TPG task (i.e. TPG(f_5)). Once T_1 needs to execute TPG(f_5) for P_3, only related information is restored in T_1. Therefore, the total runtime of SDC-TPG is reduced again.

According to the two cases mentioned above as the motivation, we proposed a deterministic parallel test pattern generation engine called *P4-TPG* that includes three different scheduling techniques (***preemptive***, ***proactive*** and ***preventive schedulings***) for acceleration. For ***preemptive scheduling***, a novel thread flow is developed to bring forward executions of TPG/FC early as many as possible for saving unnecessary idle time. For ***proactive scheduling***, pattern pre-generation is invoked to reuse the unavoidable idle time in parallel ATPG (after preemptive scheduling) and perform TPG of latter (undetected) faults. For ***preventive scheduling***, a novel thread-based skipping-fault technique is applied to rapidly shrink the size of the compaction list for the target faults on one thread. The above three scheduling techniques are implemented onto an ATPG engine (PODEM-X)[1] [17] and work correctly as a deterministic parallel ATPG to produce the same test pattern set as the serial ATPG does, meanwhile accelerating runtime. According to our experiments, *P4-TPG* successfully improves the speed-ups of SDC-TPG by averagely 76.0%, 87.2%, 94.7% and 96.6% with 1, 4, 8 and 12 threads, respectively, on 18 benchmark circuits.

The remainder of this paper is organized as follows: Section II presents the background information, including the fundamentals of ATPG and the problem of SDC-TPG as our motivation. Section III introduces the overall flow of *P4-TPG* first. Later ***preemptive***, ***proactive*** and ***preventive*** scheduling techniques are detailed, respectively. Experimental results are presented in Section IV. Finally, conclusions are drawn in Section V.

II. BACKGROUND OF DETERMINISTIC PARALLEL ATPG

In this section, we review the background information pertaining to ***deterministic parallel ATPG***. The primary concept of **automatic test pattern generation (ATPG)**, as well as its typical flow for its serial version, is first described. Next, we briefly explain the role of **fault compaction** in the context of the serial ATPG. Meanwhile, the execution order of three primary tasks (i.e. TPG, FC and FS) is also used to illustrate

[1] PODEM-X is re-implemented as an in-house tool.

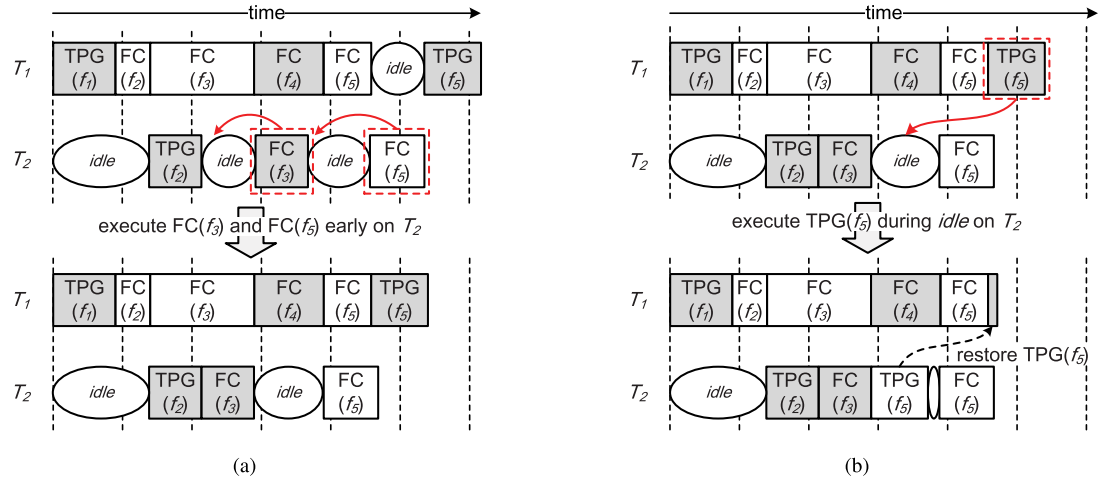

FIGURE 2. Saving/reusing idle time from early execution of later tasks in SDC-TPG by rescheduling. (a) Saving idle time. (b) Reusing idle time.

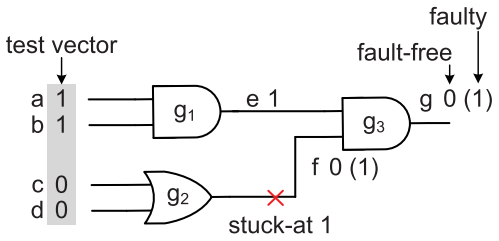

FIGURE 3. Example of a single stuck-at fault.

how compaction proceeds on faults during test generation. Finally, we introduce SDC-TPG [16], the latest work of deterministic parallel ATPG, and detail how it parallelizes ATPG with fault compaction to achieve determinism in Section II-D.

A. AUTOMATIC TEST PATTERN GENERATION

Imperfect manufacturing process may lead to defects in different ways of forms, resulting in chips that could potentially malfunction. The objective behind test generation is to produce a set of test patterns capable of uncovering any defect in every single chip. However, generating test patterns against all possible defects that could potentially occur during the manufacturing process is expensive. Automatic test pattern generator (ATPG) employ abstract representations of defects referred to as faults. One popular fault model is the single stuck-at fault, in which it is assumed that only one fault is present in the circuit under test, in order to simplify the problem of ATPG.

In the single stuck-at fault model, a fault simply denotes that a circuit node is tied to logic 1 or logic 0. Fig. 3 shows a circuit with a single stuck-at fault, in which signal f is tied to logic 1 (f/1) under the test mode. A logic 0 must be deduced under the normal mode on node f from the primary inputs of the circuit to make a difference between the fault-free (or good) circuit and the circuit with a stuck-at fault.

In Fig. 3, since g_2 is an OR gate, only $\{c=0,d=0\}$ can produce 0 on the output (f) under the normal mode, differentiating itself with stuck-at 1 under the test mode. Moreover, in order to observe the fault effect on the output g, a logic 1 must be enforced on signal e, such that fault f/1 (if it exists) can be propagated. $\{a=1,b=1\}$ is deduced accordingly from $\{e=1\}$. As a result, 1100 on the inputs is a test pattern generated for the stuck-at fault (f/1) and makes a difference on the output between the good and faulty circuits. Automatic test pattern generation (ATPG) aims at generating sufficient input patterns as tests to detect every possible fault in the circuit under test. For example, faults such as c/1, d/1, and g/0 also take turn to be targeted by ATPG.

Note that, some of the faults in the circuit can be logically equivalent, inferring that no test can be derived by which to distinguish between them. Thus, *fault collapsing* is often used to identify equivalent faults a prior in order to reduce the number of faults that must be targeted [18]–[20]. Therefore, in practical, ATPG is concerned with the generation of test patterns only for faults in the collapsed fault list.

B. FAULT COMPACTION IN ATPG

Since generating a **sufficient but small** test pattern set for the circuit under test is the objective of automatic test pattern generation (ATPG), in addition to fault collapsing, several techniques like ***pattern compaction*** and ***fault compaction*** are commonly incorporated. *Pattern compaction* is a **static** approach for reducing test counts and merges compatible patterns (before filling unspecified bits) after they are generated for respective faults. On the other hand, *fault compaction* takes a **dynamic** and incremental way to detect more faults by assigning the remaining unspecified bits in each pattern generated for one target fault in the list. Comparing to pattern compaction, *fault compaction* can typically produce test pattern sets of smaller sizes and thus is adopted in many prevailing ATPGs [21]–[23].

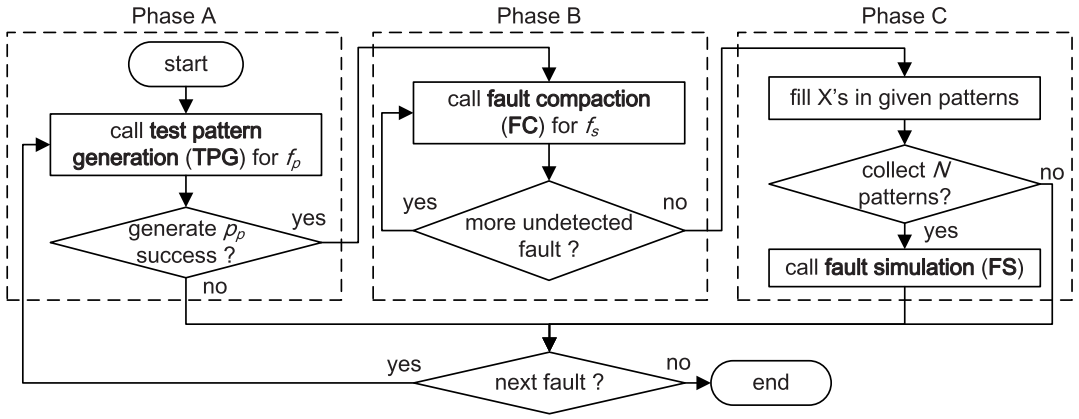

FIGURE 4. Flow of serial ATPG with fault compaction.

In fault compaction, a fault can be classified into ***primary*** or ***secondary***. One test for the *primary fault* is modified by subsequent *secondary faults* if such test after being generated still withholds certain unspecified bits (i.e. X's). The specified bits of the current test become constraints during the subsequent modification. A test for the next *secondary fault* can only be derived when it satisfies all existing constraints. Imposing constraints on the secondary faults limits the search space for test solutions and speeds up the ATPG process. If some bits for the secondary fault are newly specified, more constraints will be added to the next test modification. The search repeats until all remaining undetected faults are exhausted; in addition to the primary fault, such test detects the secondary faults as many as possible. As a result, ATPG with fault compaction effectively produce a sufficient but small test pattern set for the circuit under test.

C. TASK ORDER IN SERIAL ATPG

To clarify how ***fault compaction*** works in ATPG, we firstly introduce the typical flow of the serial ATPG and the execution order of its primary tasks. Fig. 4 shows such flow of serial ATPG with three phases containing three primary tasks: (A) *test pattern generation* (TPG), (B) *fault compaction* (FC) and (C) *fault simulation* (FS). In Phase A, a primary fault f_p is picked from the collapsed fault list, and is used as the target for pattern generation. Once f_p is detected by a generated pattern p_p, TPG stops and FC starts. Otherwise, it picks the next primary fault for TPG. In Phase B, fault compaction uses p_p as the basis to form constraints and checks if any secondary fault f_s could be detected by assigning more unspecified bits in p_p. After exhausting all undetected faults in FC, remaining unspecified bits (if any) in such pattern will be filled either 0, 1 or randomly (called X-filling) in Phase C. After X-filling, fault simulation (FS) helps to drop more undetected faults and completes one run of ATPG. A new run of ATPG will be invoked if the fault list is not empty.

Fig. 5 shows an example of the execution of multiple runs of ATPG with three primary tasks in serial TPG. f, R and P denote a target fault, an ATPG run and a pattern, respectively. Moreover, TPG(f), FC(f) or FS(f) are the respective tasks targeting fault f where a white (gray) box is referred as such task ending up with failure (success). In the first run R_1, TPG generates a pattern, P_1, for target fault f_1 and FC detects f_3, f_6 and f_{10} in Phase B. In the second run R_2, TPG picks f_2 as the primary fault from the remaining fault list and generates P_2. Under constraints imposed by P_2, FC detects more faults including f_4 and f_8 in P_2. In the third run R_3, TPG targets f_5 to generate P_3 and FC detects f_7 in P_3. Similarly, in the fourth run R_4, f_9, f_{11} and f_{13} are further detected by P_4. Once a fixed number of patterns (i.e. 4 in this example) are collected, fault simulation (FS) is invoked as shown in the fifth run R_5 and performed for dropping more faults (i.e. f_{14} and f_{16}). After R_5, fault f_{12} and f_{15} remains in the fault list. In the last run R_6, we fail to generate more patterns for f_{12} and f_{15} since these two faults are untestable. Finally, serial ATPG terminates and completes the whole process. Note that, in each run, only the remaining faults from the previous run need to be processed in the next run. For example, f_2, f_4, f_5, f_7 to f_9 and f_{11} to f_{16} left by R_1 will be targeted in either TPG or FC in R_2.

D. SYNCHRONIZED DYNAMIC COMPACTION TEST PATTERN GENERATION (SDC-TPG)

To ensure consistency of test pattern generation, a deterministic parallel TPG with dynamic fault compaction is proposed in [16]. The key idea of SDC-TPG is to ensure the same order of tasks as the serial ATPG does during parallelization. Fig. 6 shows an example of SDC-TPG on 4 threads with a time perspective where R_1 to R_4 in Fig. 5 are concurrently assigned to different threads. Note that f, T and P represent a target fault, a thread and a pattern, respectively. When generating P_1, T_1 does not idle, regardless of whether the task is confirmed to be a successful case (gray box) or a failing case (white box). Therefore, T_1 will pick a fault for the next task until all undetected faults in the list are exhausted. As a result, T_1 generated P_1 to detect the primary fault f_1 and three secondary faults (f_3, f_6 and f_{10}), For P_2, T_2 idled

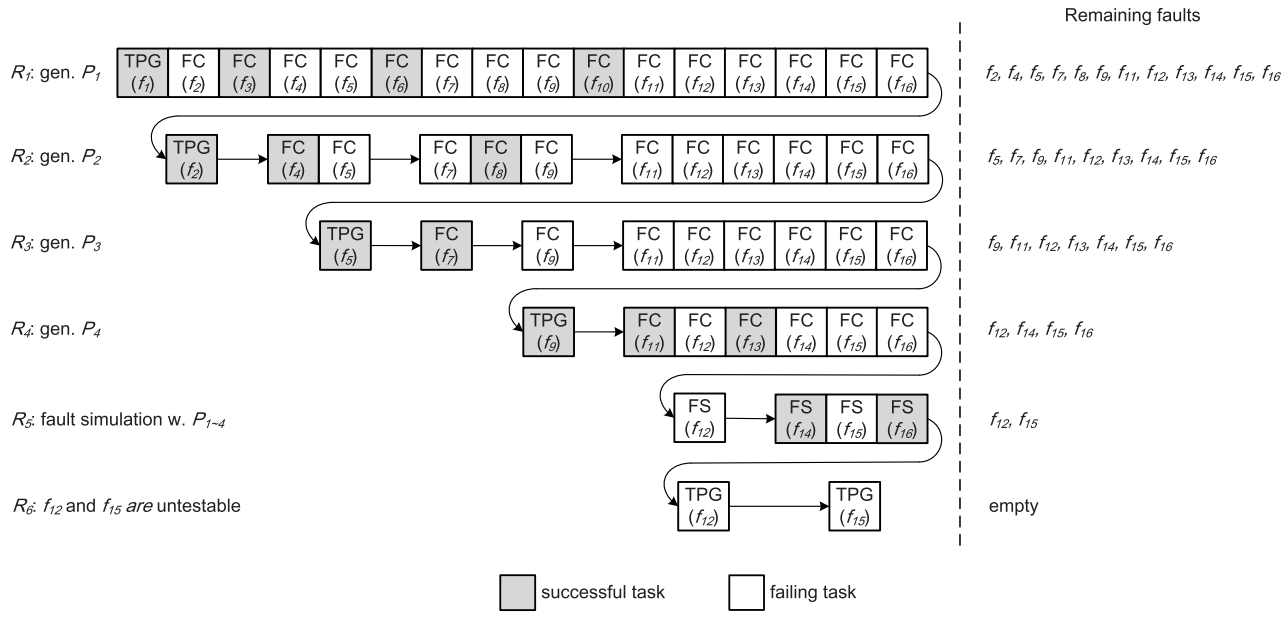

FIGURE 5. Example of a serial ATPG with fault compaction.

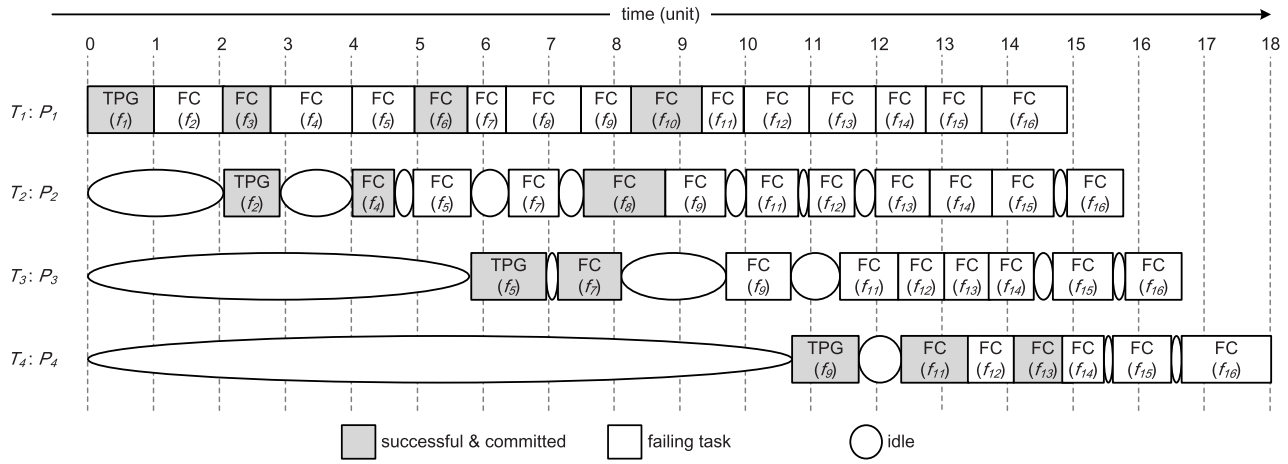

FIGURE 6. Example of executing SDC-TPG by four threads.

at the beginning until FC(f_2) was confirmed as a failing task on T_1. After f_2 cannot be detected by P_1, T_2 picked fault f_2 as the target and performed TPG on f_2. Unlike T_1, T_2 needed to wait until f_4 was committed as a FC failure on T_1 and then FC(f_4) succeeded on T_2. Similarly, T_2 performed FC(f_5) right after FC(f_5) failed on T_1. To give another example, T_2 idled again after FC(f_7) was shown as a failing case. After FC(f_8) was confirmed as failure on T_1, T_2 could start to perform FC(f_8). As a result, T_2 generated P_2 to detect faults (f_2, f_4, and f_8). Following the strategy of SDC-TPG, T_3 generated P_3 to detect f_5 and f_7; T_4 generated P_4 to detect f_9, f_{11} and f_{13}.

Among four threads, only the thread invoked first (i.e. T_1) did not idle at the beginning of its execution. For the other threads (i.e. T_2, T_3 and T_4), TPG in the respective execution can start until a fault is determined undetected by FC in the previous thread. For example, T_3 started to invoke TPG(f_5) after FC(f_5) failed on T_2. T_4 started after FC(f_9) failed on T_3. As one can see in Fig. 6, the idle time to invoke TPG on the latter threads became longer. Moreover, longer idle time also appeared in fault compaction (FC) for the latter threads. For example, FC(f_4) on T_2, FC(f_9) on T_3 and FC(f_{11}) on T_4 needed to idle until the failure of corresponding tasks on T_1, T_2 and T_3, appeared, respectively. Therefore, although SDC-TPG ensures the consistency of results after test pattern generation, much waste of computing resources during the idle time becomes a big obstacle in achieving better

acceleration, motivating us to develop a new deterministic parallel ATPG engine named ***P4-TPG*** with three novel scheduling techniques.

III. P4-TPG: PREEMPTIVE, PROACTIVE AND PREVENTIVE SCHEDULING IN Parallel TPG

This section is dedicated to the proposed deterministic parallel ATPG engine ***P4-TPG*** and provides the details about the three novel scheduling techniques embedded in *P4-TPG*. First, the thread flows of SDC-TPG and *P4-TPG* are compared and contrast where three scheduling techniques are annotated in Fig. 7. Second, ***preemptive scheduling*** is elaborated and includes ***fault commitment checking*** (***FCC***) and ***concurrent task interruption*** (***CTI***) to save idle time. Third, ***proactive scheduling*** uses ***pattern pre-generation*** (***PPG***) to reuse the remaining idle time left by ***preemptive scheduling***. Last, ***preventive scheduling*** invokes ***fault polarity check*** (***FPC***) and ***backward unpropagatale fault removal*** (***BUFR***) to dynamically skip a large number of faults for acceleration.

A. THREAD FLOW OF P4-TPG

As SDC-TPG is described in section II-D, its thread flow for one pass of ATPG is shown in Fig. 7(a). One thread picks an undetected fault, f_t, from the fault list at the beginning. If the fault list is not *empty*, the thread picks the first undetected fault as the target f_t. Then, test pattern generation TPG(f_t) or fault compaction FC(f_t) will be followed. The result of TPG(f_t) (or FC(f_t)) is either a success or a failure. If TPG(f_t) (or FC(f_t)) succeeds, it is regarded as a successful task. Otherwise, such TPG(f_t) (or FC(f_t)) is regarded as a failing task. Whether TPG/FC succeeds or not, the thread returns to the waiting mode for the next fault. One pass of ATPG in SDC-TPG stops when all undetected faults in the list are processed; then, a test pattern is successfully generated. As you can see, each thread needs to pick a fault during either TPG or FC, fault dependence makes much idle time appear as Fig. 6 shows. Although SDC-TPG ensures determinism and achieves acceleration, tremendous idle time is retained before a thread performs TPG or FC on a target fault; consequently, the computing resource becomes thriftless during ATPG.

The proposed *P4-TPG* adopts a similar thread flow for one pass of ATPG as SDC-TPG does but performs more jobs. For dealing with the idle time in SDC-TPG, three novel techniques (i.e. ***preemptive***, ***proactive*** and ***preventive*** schedulings) are developed and integrated into *P4-TPG*. First, in *P4-TPG*, ***preemptive scheduling*** is applied on each thread and steals idle time to perform the corresponding TPG/FC task of the target fault as early as possible. Once the fault fails to be detected by the previous pattern on other thread, the successful TPG/FC result on this thread can be committed immediately. So that the idle time on each thread is not wasted. Second, if no task can be preemptively executed on one thread, ***proactive scheduling*** re-utilizes the remaining idle time and invokes pattern pre-generation (PPG) on the next target fault. The PPG result is saved beforehand and will be restored as the TPG task is invoked by the thread later. Third, considering the constraints enforced by the base pattern generated during the TPG/FC process, some of the remaining undetected faults in the list will become incompatible. To avoid performing FC in vain, ***preventive scheduling*** checks the legitimacy of the picked fault and eliminates those that disagree with the current pattern. As a result, a new thread flow shown as Fig. 7(b) is developed accordingly for effectively producing the same test pattern set but shortening overall execution time of *P4-TPG*.

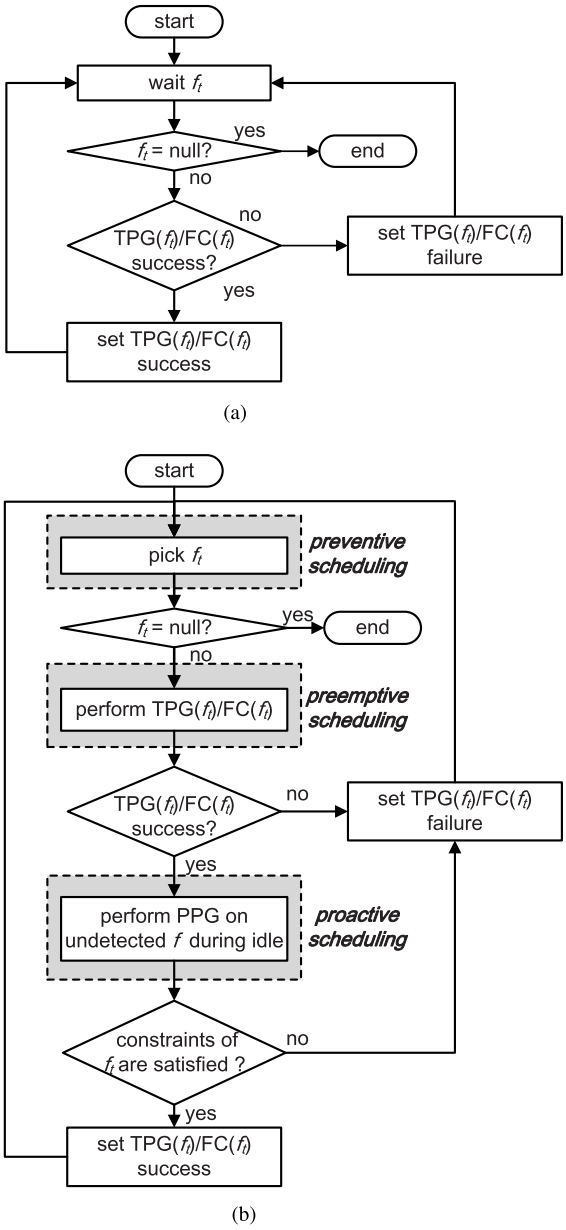

FIGURE 7. Thread flows for one pass of ATPG in SDC-TPG and *P4-TPG*. (a) SDC-TPG. (b) *P4-TPG*.

B. PREEMPTIVE SCHEDULING IN P4-TPG

To save trenmendous idle time caused by task dependence between threads, ***preemptive scheduling*** is proposed and

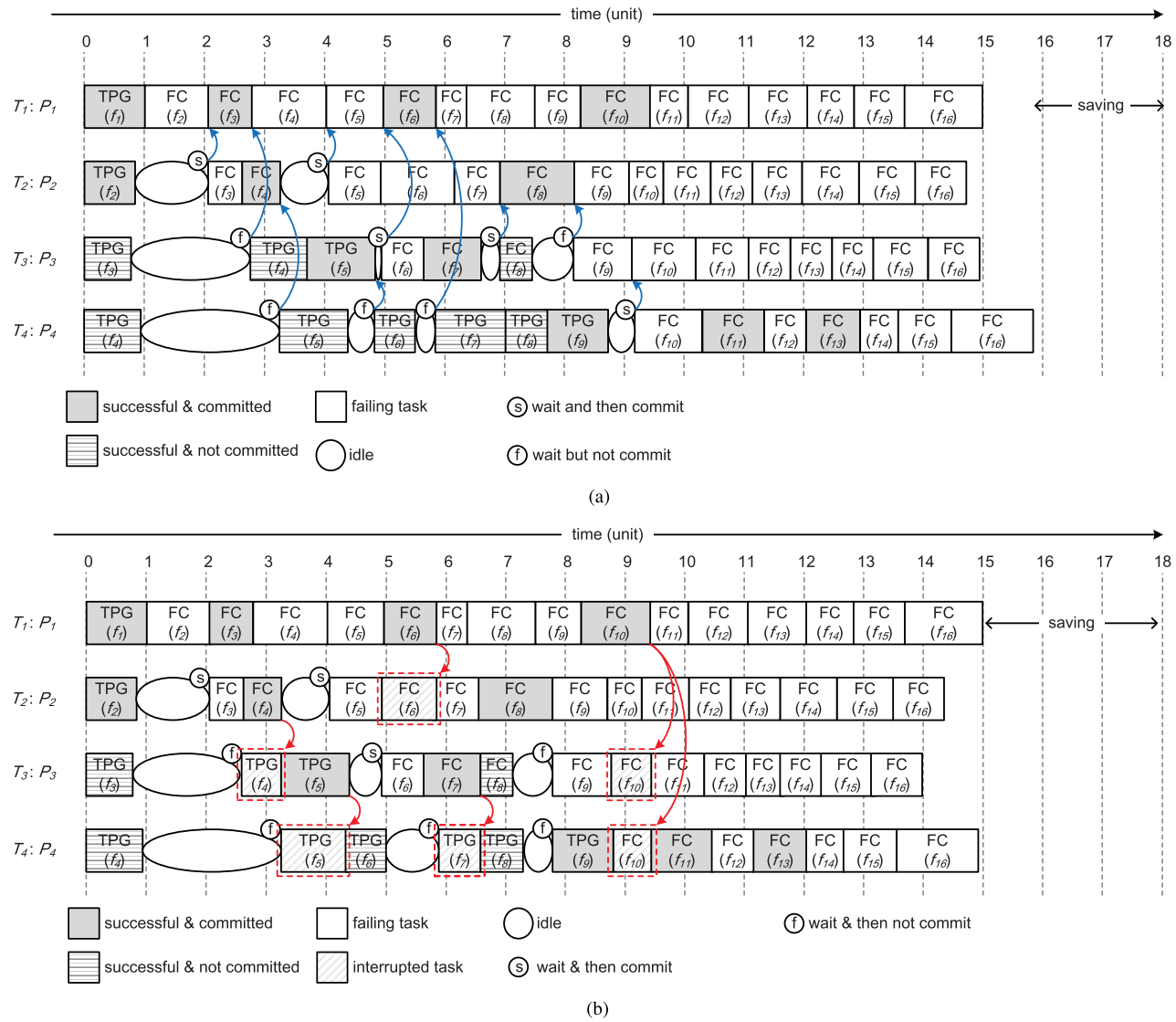

FIGURE 8. Example of ***preemptive scheduling***. (a) ***Fault commitment checking (FCC)***. (b) ***Concurrent task interruption (CTI)***.

brings forward execution of TPG/FC tasks *as early as possible* in ATPG but waits for committing results only if necessary. Fig. 8 shows the result after applying ***preemptive scheduling*** to the example in Fig. 6. Note that since some tasks are brought forward, some augmented tasks are also generated. For example, thread T_3 performs the following extra augmented tasks, TPG(f_3), TPG(f_4), FC(f_6), FC(f_8) and FC(f_{10}), during the pass of ATPG but also moves forward TPG(f_5) and FC(f_9). Similar phenomenon occurs on other threads (T_2 and T_4), too.

Due to these augmented tasks, two particular components, (1) ***fault commitment checking*** (***FCC***) and (2) ***concurrent task interruption*** (***CTI***) are also participated in ***preemptive scheduling*** to help retain determinism but meanwhile suspend useless computation. For retaining determinism, ***fault commitment checking*** (***FCC***) ensures the same detection order of all faults in *P4-TPG* for generating the same test pattern set. For avoiding useless computation, ***concurrent task interruption*** (***CTI***) is responsible for early terminating the corresponding tasks whose target faults are detected by the tasks in the previous patterns on other threads and thus no more needed. To further clarify details of ***FCC*** and ***CTI***, Fig. 8 illustrates the results after applying (1) ***fault commitment checking*** (***FCC***) and (2) ***concurrent task interruption*** (***CTI***) to the example shown in Fig. 6 by 4 threads where T, P and f represent a thread, a pattern and a fault, respectively.

1) FAULT COMMITMENT CHECKING (FCC)

To guarantee determinism in parallel ATPG, the detection order of a fault is the key. In *P4-TPG*, before any fault that can be declared as *detected* by one pattern running on thread, ***fault commitment checking*** (***FCC***) needs to be invoked and checks task dependency of all other threads running for preceding patterns (those with the smaller pattern indexes).

According to the proposed thread flow in Fig. 7(b), once a fault can be detected by FC on the basis of the current pattern, the running thread will then invoke ***FCC*** to check the commitment condition. That is, ***FCC*** on thread T checks the status of the same target fault f on any other thread T' who has a smaller pattern index than T has. If all threads T' running for preceding patterns fail to detect fault f, the current TPG/FC task on thread T can commit the result and is regarded as a *successful & committed* case of fault detection.

Fig. 8(a) shows the complete result after applying ***FCC*** to the SDC-TPG example in Fig. 6. Most of TPG/FC tasks of one fault on one thread do not overlap with the tasks of the same fault on other threads and thus ***FCC*** does nothing. However, only 10 particular ***FCC*** events highlighted by the arrows involves task dependency among threads as shown in Fig. 8(a) and can be classified into four cases: (1) TPG waits and then commits, (2) TPG waits but does not commit, (3) FC waits and then commits, (4) FC waits but does not commit. As an example for case (1), fault f_2 is detected by pattern P_2 after TPG(f_2) on thread T_2. Later, T_2 invokes ***FCC*** to commit f_2. However, T_2 has to wait and can only commit the current P_2 until FC(f_2) on T_1 fails. For case (2), TPG(f_4), TPG(f_5) and TPG(f_6) on thread T_4 are all examples. Although all these TPG tasks succeed, the corresponding results cannot be committed because the target faults are also detected by preceding patterns. For example, TPG(f_5) cannot commit the current pattern P_4 on thread T_4 because TPG(f_5) in the preceding pattern P_3 on T_3 detects f_5. FC(f_7) on thread T_3 is an example for case (3). After FC(f_7) fails on T_2, FC(f_7) on T_3 can be committed. Similar but with the opposite result for case (4), after T_3 waits, FC(f_8) still cannot be committed because FC(f_8) on T_2 commits at the end. In summary, ***FCC*** ensures the same detection order of faults as that from the serial ATPG and thus retain determinism in ***preemptive scheduling***.

2) CONCURRENT TASK INTERRUPTION (CTI)

As mentioned previously, ***preemptive scheduling*** brings forward execution of possible TPG/FC tasks on one thread and relies upon ***FCC*** for committing the detection of faults. Moreover, once a TPG/FC task completes the commitment of fault detection on one thread, all running TPG/FC tasks on the other threads may also be terminated earlier since they are generating succeeding patterns (i.e. patterns with bigger index numbers) to the committed one. Accordingly, ***concurrent task interruption*** (***CTI***) is proposed and suspends straightway those useless computation of the same faults on the other threads. That is, if a thread (T) performs a TPG/FC task on a fault (f) successfully and commits the detection of f simultaneously, then T invokes ***CTI*** to interrupt tasks of the same fault on all the other threads that deal with succeeding patterns.

Fig. 8(b) shows the example again after applying ***CTI*** and three different types of cases can be found: (1) TPG interrupts TPG, (2) FC interrupts TPG, (3) FC interrupts FC. As an example for case (1), f_5 is detected by TPG on T_3. Later, T_3 invokes ***CTI*** to interrupt T_4, which runs the TPG task on the same fault. Second, for case (2), T_2 interrupts TPG(f_4) on T_3 once FC(f_4) commits the detection of f_4 on T_2. Similarly, T_3 interrupts TPG(f_7) on T_4 after FC(f_7) commits the detection of f_7 on T_3. Last, FC(f_6) and FC(f_{10}) on T_1 are examples for case (3). Once FC(f_6) commits the detection of f_6 by P_1, T_1 interrupts the task for the same fault on T_2. Similarly, FC(f_{10}) commits the detection of f_{10} by P_1, ***CTI*** suspends the corresponding FC(f_{10})'s on two other threads (i.e. T_3 and T_4). In summary, ***CTI*** suspends a number of redundant tasks and shortens computational time in ***preemptive scheduling***.

Note that ***preemptive scheduling*** incorporates ***FCC*** for ensuring the detection order of faults as the serial ATPG and also utilizes ***CTI*** to interrupt useless tasks for shortening computational time. According to our experimental results, *P4-TPG* with ***preemptive scheduling*** can improve the speed-ups of SDC-TPG[2] by averagely 10.2%, 13.4% and 13.9% using 4, 8 and 12 threads, respectively, on 18 benchmark circuits.

C. PROACTIVE SCHEDULING IN P4-TPG

After applying ***preemptive scheduling***, different numbers of tasks will be brought earlier on the threads. However, much idle time still exists in *P4-TPG* as shown in Fig. 8(b). To further shorten the overall runtime of ATPG, the remaining idle time can be further reused by ***pattern pre-generation*** (***PPG***). As a result, ***PPG*** brings forward the latter TPG tasks to these idle time but does not change the commitment order of fault detection, retaining the determinism as the serial ATPG does.

Before detailing ***PPG***, let us revisit the example in Fig 5. From R_1 to R_5, f_{12} is processed by fault compaction (FC) 4 times and by fault simulation (FS) 1 time. Until TPG(f_{12}) finishes on R_6, fault f_{12} can be confirmed as *untestable*. Similarly, the same phenomenon can be observed on fault f_{15}. Therefore, if f_{12} and f_{15} are determined as *untestable* earlier, much futile computation can be saved. As a result, the execution time can be further reduced whereas the final test pattern set remains the same.

In the following, the mechanism of ***pattern pre-generation*** (***PPG***) is detailed. As one thread enters the idle mode, it will invoke ***PPG*** for an undetected fault in the list as the target. After ***PPG***, two different results may yield: (1) if the target fault is detected by ***PPG***, such fault will be reserved in the list and the generated pattern will be stored in advance for the possible access by a future TPG task executed on a certain thread; (2) if the target fault is not detected by ***PPG***, such fault is known *untestable* and thus can be safely removed from the fault list. Either of these two results help ATPG reduce the total runtime.

Fig. 9 shows an example for applying ***proactive scheduling*** in *P4-TPG* where possible PPG tasks will be invoked whenever threads are idle. For example, long idle time appears on

[2]SDC-TPG is an in-house re-implementation according to [16].

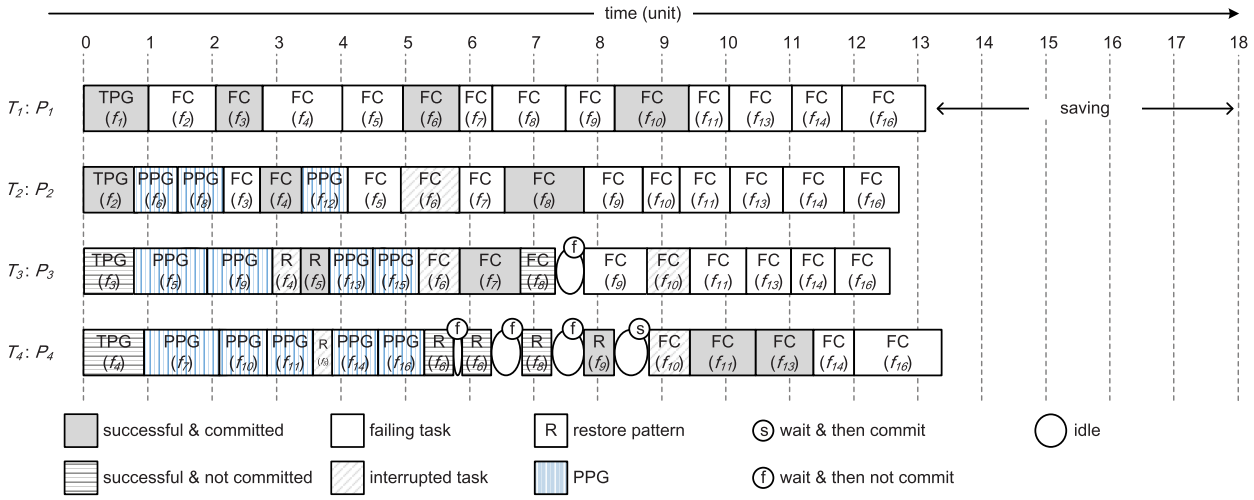

FIGURE 9. Example of ***proactive scheduling*** in ***P4-TPG***.

thread T_2 before FC(f_3) starts. Therefore, ***proactive scheduling*** inserts PPG(f_6) and PPG(f_8) during such idle time and check if these faults can be detected or not and stored the corresponding patterns if they are successful. Similarly, thread T_3 invokes PPG(f_6) and PPG(f_9) at the first idle time and inserts PPG(f_{13}) and PPG(f_{15}) at the second idle time. In particular, given that f_{12} and f_{15} are untestable faults, after PPG(f_{12}) and PPG(f_{15}) fail, f_{12} and f_{15} are also removed from the fault list. Accordingly, all FC(f_{12}) and FC(f_{15}) tasks disappear on all threads, shortening the total runtime of ATPG.

As a matter of fact, ***proactive scheduling*** has a side effect. That is, although a target fault can be pre-computed by PPG after ***proactive scheduling***, a restoration task, R, needs to be invoked to replace the original TPG task for such fault. However, empirically, one R task takes about 100X smaller time than one TPG/FC task does and thus is relatively negligible. Therefore, the advantages of ***proactive scheduling*** in *P4-TPG* can be summarized into: (1) reusing idle time for TPG and (2) safely removing untestable faults earlier. According to our experimental results, applying ***proactive scheduling*** can improve the speed-ups of *P4-TPG* by averagely 1.6%, 3.8% and 5.7% using 4, 8 and 12 threads, respectively, on 18 benchmark circuits.

D. PREVENTIVE SCHEDULING IN P4-TPG

After applying ***preemptive*** and ***proactive scheduling***, *P4-TPG* achieves determinism and improves SDC-TPG. However, in pursuit of better performance, ***preventive scheduling*** is developed and utilizes the idea of skipping *redundant* faults under a *dynamic* environment during the pattern generation to accelerate *P4-TPG*. In fact, ***preventive scheduling*** consists of two stages, (1) ***fault polarity check*** (***FPC***) and (2) ***backward-unpropagatable fault removal*** (***BPFR***), and avoids futile computation during fault compaction.

In the following, we will explain how two stages work for examining the status of fault and dynamically eliminating redundancy during the development of a pattern in ***preventive scheduling***.

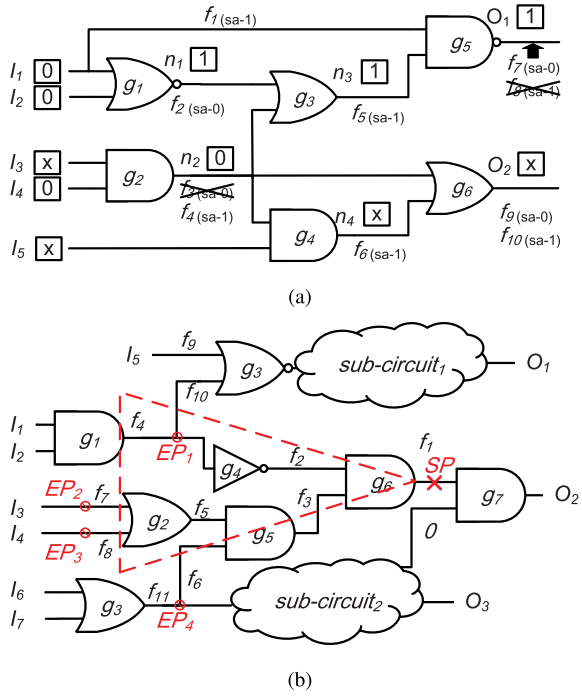

FIGURE 10. Two techniques of ***preventive scheduling***. (a) ***Fault polarity check (FPC)***. (b) ***Backward-unpropagatable fault removal (BUFR)***.

1) FAULT-POLARITY CHECK (FPC)

In stage 1, **FPC** checks if or not the polarity of the picked secondary fault $\phi_{s.f.}$ is compatible to the net assignment (ϕ_n) during fault compaction. If $\phi_{s.f.}$ is equal to ϕ_n, then ***FPC*** skips such fault and picks the next secondary fault. On the other hand, the secondary fault is compatible with the current pattern and can be used as a target for fault compaction (FC). Until all undetected faults in the list are examined, fault

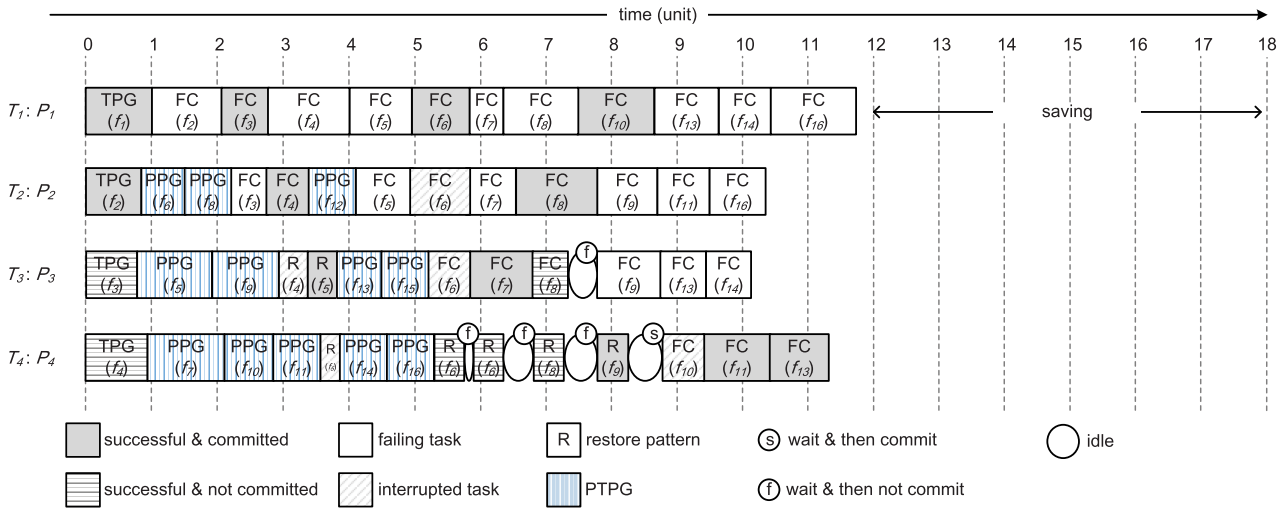

FIGURE 11. Example of ***preventive scheduling*** in ***P4-TPG***.

compaction (FC) in the pass of ATPG finishes and a test pattern is generated.

FPC includes two different cases of polarity comparison:

- (1) $\phi_{s.f.} = \phi_n$: the secondary fault f requires the opposite assignment (e.g. a stuck-at-1 fault requests a logic 0) on the net and conflicts with the current one; thus, f is dynamically redundant and can be skipped.
- (2) $\phi_{s.f.} \neq \phi_n$ or $\phi_n = X$: the secondary fault f requires the assignment on the net same as the current one or the current net has not yet assigned with a value; thus, f could be compacted by FC with the current pattern.

Fig. 10(a) shows an example for ***FPC*** where f, g, I, O and n denote a fault, a gate, an input, an output and a net, respectively. Assume that the current pattern is 00X0X after applying TPG on the primary fault f_7 (stuck-at-0). The logic value of n_1, n_3 and O_1 are assigned 1 whereas n_2 is assigned 0; n_4 and O_2 remain X. For case (1), the secondary faults, f_3 (stuck-at-0) and f_9 (stuck-at-1), with the same polarity as n_2 (logic-0) and O_1 (logic-1), respectively, will request the opposite logic values on n_2 and O_1, which are impossibly satisfied, are thus eliminated. The remaining undetected faults belong to case (2) and can be picked for future fault compaction. Note that since the pattern and the net assignment change after each round of fault compaction, FPC may be performed again on those nets with new value assignments.

2) BACKWARD-UNPROPAGATABLE FAULT REMOVAL (BUFR)

In stage 2, ***BUFR*** is developed and more aggressively skips unnecessary tasks in *P4-TPG* for achieving better acceleration. Unlike conventional *forward propagation*, which the fault effect is propagated through one or more paths towards a *primary output* of the circuit, ***BUFR*** is a backward search and deletes faults that reside along the unpropagatable paths from the undetected list.

BUFR starts from a node and applies the backward search to remove faults until two stopping criterion (hitting either (1) a *branching* end node or (2) a *PI* node) are met. Fig 10(b) presents an example for *BUFR* where f, g, I, O, SP and EP denote a fault, a gate, an input, an output, a start node and an end node, respectively. First, ***BUFR*** finds that the secondary fault, f_1, on the output of gate g_6 cannot be detected by FC because the other pin of gate g_7 is assigned with a controlling value 0. Therefore, no other faults propagating through gate g_6 can be detected on O_2. Accordingly, ***BUFR*** applies *depth-first search* (*DFS*), starting from the output of gate g_6 and backward removes all faults in the fan-in cone. ***BUFR*** will stop searching and removing faults until reaching the end point EP_1, which belongs to condition (1). This is because fault f_4 may have an alternative to be propagated to PO O_1 through a different path (i.e. $g_3 \rightarrow sub\text{-}circuit_1 \rightarrow O_1$). EP_4 also belongs to case (1) and fault f_{11} will not be removed from the list since f_{11} can avoid gate g_7 and propagates toward O_3 through the path ($sub\text{-}circuit_2 \rightarrow O_3$).

For case (2), ***BUFR*** will stop searching at EP_2 and EP_3 since I_3 and I_4 are PIs of the circuit under test and no more fault can be found. As a result, seven faults (including f_1, f_2, f_3, f_5, f_6, f_7 and f_8) are removed from the undetected list since these faults can impossibly be propagated through gate g_7 and thus become *untestable* under this circumstance.

Since ***BUFR*** removes numerous undetected faults by DFS until reaching a stopping criterion, the corresponding FC tasks related to these removed faults will disappear during the pass of ATPG. That is to say, ***BUFR*** effectively shortens the runtime of *P4-TPG* (in particular, during the first few passes of ATPG) and achieves better acceleration. Fig. 11 shows an example of applying ***preventive scheduling*** in *P4-TPG*. Many faults are removed according to the current pattern of each thread. For example, based on P_1, f_9 and f_{11} cannot

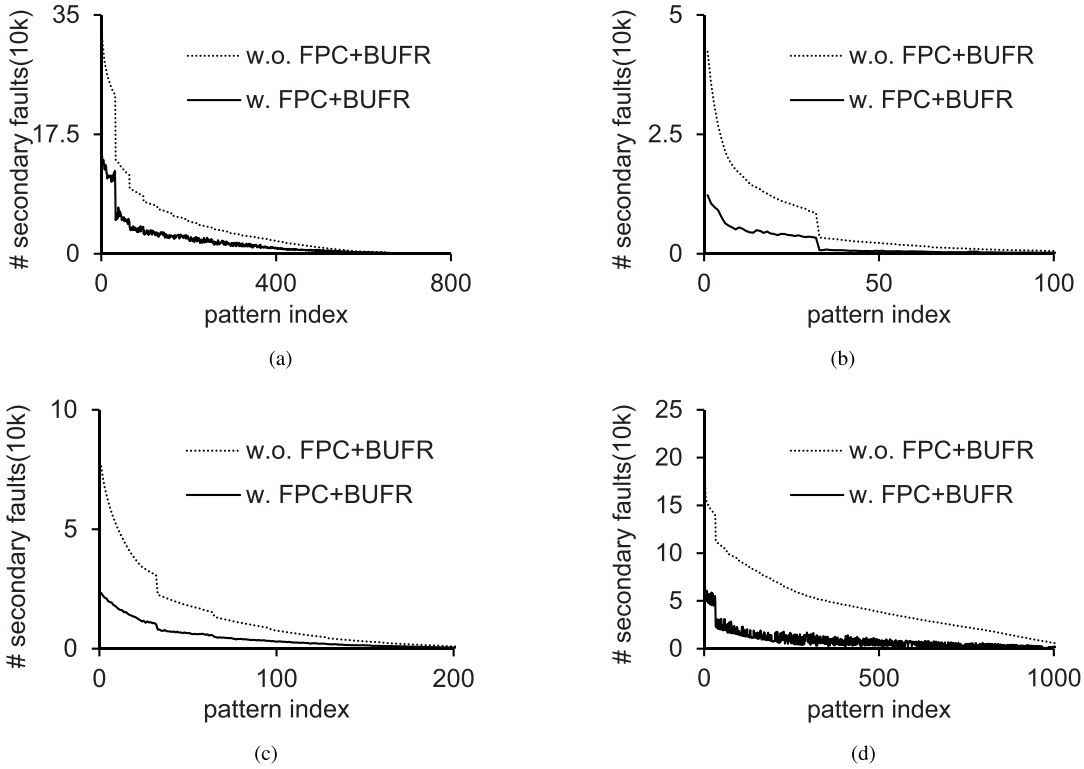

FIGURE 12. The number of secondary fault on four benchmarks.

be activated or propagated to the PO. Therefore, these faults are removed from fault list for T_1. Similarly, T_2, T_3 and T_4 can skip faults that are incompatible with the corresponding pattern, respectively.

Fig. 12 compares the number of faults remaining in the undetected list of each pattern after using ***FPC*** and ***BUFR*** in *P4-TPG*. Our experimental result clearly indicates that ***FPC*** and ***BUFR*** indeed reduce a large number of secondary faults on those patterns with relatively smaller indexes. Before performing the first round of fault simulation (FS) (as 32 patterns are accumulated), the reduction percentage of secondary faults are 57.8%, 65.7%, 66.2% and 65.5% on circuit *b*19, *pci_bridge*32, *bench*4 and *vga_enh*, respectively. As a result, the total runtime of ATPG are significantly reduced. More results will be presented in the experimental section.

IV. EXPERIMENTAL RESULTS

P4-TPG is implemented in C/C++ with Pthreads and runs on a Linux machine with 20 processors and 64GB RAM. Experiments are conducted on seven benchmarks from ISCAS'89, five benchmarks from IWLS'05 and five benchmarks from industrial circuits provided by the Industrial Technology Research Institute of Taiwan (ITRI) [24]. Table 1 lists basic information about 18 benchmark circuits. #pttn, FC and RT refer to the numbers of patterns, fault coverage and runtime, respectively. Columns 1 to 3 denote circuit names, the numbers of gates (#gate) and the numbers of stuck-at faults (#SA-fault). Columns 4 to 6 denote #pttn, FC and RT, which are obtained by a commercial tool,[3] respectively. Columns 7 and 8 denote #pttn and FC derived by *P4-TPG*. In particular, the last two columns compare total runtime (RT) of SDC-TPG and *P4-TPG*.

P4-TPG is built upon the ATPG package, PODRM-X, which is a re-implementation of the Path-Oriented Decision-Making algorithm [17]. In particular, *P4-TPG* further incorporates three techniques, *preemptive scheduling*, *proactive scheduling* and *preventive scheduling* into the PODEM-X engine for improving runtime performance. For ***preemptive scheduling***, *P4-TPG* executes tasks TPG/FC earlier as many as possible for saving idle time. For ***proactive scheduling***, *pattern pre-generation* (*PPG*) is invoked to reuse the remaining idle time left by ***preemptive scheduling***. ***Preventive scheduling*** consists of *fault polarity check* (*FPC*) and *backward unpropagatable fault removal* (*BUFR*) to dynamically shrink the size of the list for remaining undetected faults. Experimental results show that *P4-TPG* effectively retain determinism (generating the same test pattern set) and successfully improve the speed-ups of SDC-TPG. Moreover, how the number of threads affects the speed-ups of *P4-TPG* is also investigated in our experiments.

[3] In our experiments, the merge effort is set high and automatic compression is specified with run_atpg

TABLE 1. Information of benchmark circuits.

			Tool[1]			Our experiments (single-thread)			
								SDC-TPG	P4-TPG (only w. ***preemptive scheduling***)
circuit	#gate	#sa_Fault	#pttn	FC (%)	RT (sec)	#pttn	FC (%)	RT (sec)	RT (sec)
$b15$	9953	21096	738	99.98	5.76	654	98.11	129.65	123.53
$b17$	30700	65532	766	99.99	18.03	721	98.94	832.28	754.76
$b18$	86157	187998	878	99.95	75.57	787	99.25	4145.39	4170.30
$b19$	163033	348168	1351	99.42	301.68	765	97.26	12934.19	13072.70
$b20$	11693	24756	463	99.89	2.43	524	99.64	189.42	172.81
$b21$	13084	28244	567	100.00	3.55	664	99.62	196.93	177.60
$b22$	18577	42006	488	99.99	6.61	577	99.62	263.43	263.61
$RISC$	23477	41762	323	100.00	2.88	313	100.00	9.54	9.36
mem_ctrl	10685	19564	414	100.00	1.00	418	99.99	11.33	11.16
$pci\_bridge32$	28435	51492	225	100.00	3.22	213	100.00	15.00	11.95
$ethernet$	91553	169840	1055	100.00	36.00	989	99.99	741.55	701.74
vga_enh	135580	264523	1474	100.00	84.82	1638	99.13	2057.26	1982.91
DMA	77827	145382	1793	99.98	29.00	1765	99.82	2161.67	2145.32
$bench2$	104968	209520	630	100.00	26.39	459	100.00	286.80	271.05
$bench3$	219276	445794	1621	100.00	101.00	1300	100.00	2672.57	2766.44
$bench4$	41966	83154	430	100.00	6.55	295	100.00	31.55	28.49
$bench7$	776946	1580100	505	100.00	785.37	375	100.00	13768.49	13790.70
$bench8$	103869	205780	1534	100.00	33.43	1185	100.00	421.91	403.01
avg.			847	99.96	84.63	758	99.52	2270.50	2269.86

[1,2] Difference of pattern count, fault coverage, and runtime come from the ATPG algorithms and their settings.

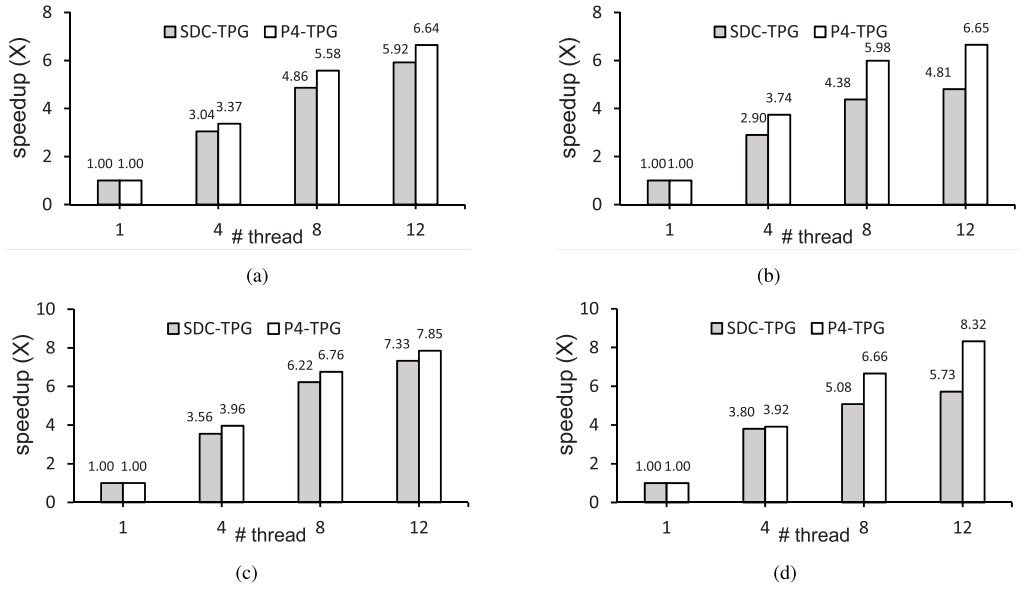

FIGURE 13. Comparison of speedup between SDC-TPG [16] and *P4-TPG* only with preemptive scheduling.

A. COMPARISON OF SPEEDUP UNDER DIFFERENT NUMBERS OF THREADS ON PREEMPTIVE SCHEDULING

Preemptive scheduling makes *P4-TPG* execute PG/FC tasks as early as possible. Fig. 13 shows the comparison of speed-ups using different numbers of threads on four benchmarks in SDC-TPG and in *P4-TPG* (only with ***preemptive scheduling***). After applying ***preemptive scheduling***, the runtime overhead under one thread is found negligible (i.e. <1%). Meanwhile, ***preemptive scheduling*** increases the

TABLE 2. Performance of speedup between SDC-TPG and *P4-TPG*.

	SDC-TPG [16]				*P4-TPG*			
	speedup (x)				speedup (x)			
#thread	1	4	8	12	1	4	8	12
$b15$	1.00	1.73	2.14	2.59	1.09	2.12	2.75	2.82
$b17$	1.00	2.55	3.44	4.16	1.20	3.12	4.24	4.96
$b18$	1.00	2.86	4.20	5.46	1.02	3.35	5.26	6.86
$b19$	1.00	3.04	4.86	5.92	1.04	3.85	6.32	7.88
$b20$	1.00	2.91	4.33	5.08	1.09	3.28	5.25	6.47
$b21$	1.00	2.42	3.21	3.76	1.13	3.52	4.83	5.51
$b22$	1.00	2.09	2.79	3.28	1.04	2.76	3.63	4.25
$RISC$	1.00	3.23	5.75	6.77	1.04	3.72	6.08	8.78
mem_ctrl	1.00	2.90	3.79	4.46	1.18	3.82	6.36	7.51
$pci\_bridge32$	1.00	2.90	4.38	4.81	1.51	4.79	7.77	9.21
$ethernet$	1.00	2.92	3.85	3.86	1.98	4.92	5.62	4.76
vga_enh	1.00	3.56	6.22	7.33	3.73	12.51	19.02	22.41
DMA	1.00	3.03	3.59	4.68	2.52	7.07	10.35	11.88
$bench2$	1.00	3.58	6.01	6.59	2.45	8.61	14.42	17.43
$bench3$	1.00	3.55	5.98	7.30	2.30	8.11	14.11	17.69
$bench4$	1.00	3.80	5.08	5.73	2.42	8.57	13.79	16.03
$bench7$	1.00	3.43	5.64	5.70	2.42	8.74	14.47	15.07
$bench8$	1.00	3.59	6.10	7.25	2.42	8.42	14.21	16.92
avg.	1.00	3.01	4.52	5.26	1.76	5.63	8.80	10.36

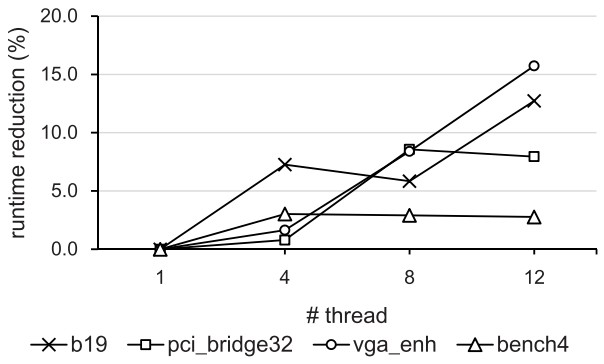

FIGURE 14. Runtime reduction of ***proactive scheduling*** in ***P4-TPG*** on four benchmarks.

average speed-up by 12.6%, 21.7% and 23.9% under 4, 8 and 12 threads, respectively, when compared with *SDC-TPG* using the same number of threads. This clearly indicates that ***preemptive scheduling*** in *P4-TPG* effectively saves idle time and shortens total runtime in ATPG. Note that the numbers of pattern (#pttn) and the fault coverages (FC) between SDC-TPG and *P4-TPG* are the same as shown in Table 1 because both are deterministic TPG engines.

B. COMPARISON OF RUNTIME UNDER DIFFERENT NUMBERS OF THREADS ON PROACTIVE SCHEDULING

For ***proactive scheduling***, ***PPG*** brings forward the latter TPG tasks to the remaining idle time after applying ***preemptive scheduling***. Fig. 14 compares the runtime between *P4-TPG* with ***PTPG*** and without ***PPG*** on four benchmarks and shows our reduction. As shown in Fig. 14, ***PPG*** decreases the average runtime by 0%, 3.2%, 6.4% and 9.8% using 1, 4, 8 and 12 threads, respectively. Note that the reduction is 0% under 1 thread because no idle time appears in ATPG. However, the runtime reduction increases along with the number of threads. As a result, *PPG* efficiently reuses the remaining idle time and avoids untestable faults in advance for acceleration. In particular, close to zero time can be reduced on benchmark circuit *bench*4 because there is almost no untestable faults and thus *PPG* is barely invoked. On the contrary, another benchmark circuit *vga_enh* exhibits a different result, in which the runtime improves with the increasing number of threads.

C. COMPARISON OF SPEEDUP FOR P4-TPG ON PREVENTIVE SCHEDULING

For preventive scheduling, the two novel skipping techniques (***FPC+BUFR***) are proposed to shrink the fault list dynamically during the generation of each pattern. Table 2 presents the comparison of speed-ups using the different numbers of threads. Column 1 denotes the name of the circuits. Column 2 to 5 list the speed-ups under the different numbers of threads, on the basis of the runtime obtained by one-thread SDC-TPG. After applying ***preventive scheduling***, *P4-TPG* achieves an average speed-up of 1.76X, 5.63X, 8.80X and 10.36X when using 1, 4, 8 and 12 threads, respectively. Moreover, *P4-TPG* improves the acceleration by 76.0%, 87.2%, 94.7% and 96.6% under 1, 4, 8 and 12 threads, respectively, compared to SDC-TPG using the same numbers of threads.

In particular, the speed-ups on several cases like *DMA*, *vga_enh*, *bench*2, *bench*3, *bench*4, *bench*7 and *bench*8 even become supper-linear (i.e. the speed-up exceeds the number of threads). Since ***preventive scheduling*** can promptly skip potentially redundant faults with respect to the current pattern, a larer number of effortless tasks can be avoided. As a result, ***preventive scheduling*** further facilitates acceleration of *P4-TPG*.

V. CONCLUSION

Although existing deterministic parallel TPGs like SDC-TPG guarantee the consistency of the test pattern set as the serial ATPG does, tremendous idle time is wasted on synchronizing independent tasks (either independent or dependent) between threads. For saving and reusing the idle time, we propose *P4-TPG*, a deterministic parallel test pattern generator, and incorporate ***preemptive***, ***proactive*** and ***preventive*** schedulings for accelerating ATPG. For ***preemptive scheduling***, ***fault commitment checking*** (***FCC***) and ***concurrent task interruption*** (***CTI***) are developed for reducing the idle time between dependent tasks and terminating the redundant tasks among threads as much as possible. For ***proactive scheduling***, ***pattern pre-generation (PPG)*** moves forward the TPG task of the latter (undetected) faults to the remaining idle time and thus either a fault can be identified *untestable* early or the generated pattern can be restored sooner whenever the TPG task is invoked. For ***preventive scheduling***, ***fault-polarity check*** (***FPC***) and ***backward-unpropagatable fault removal*** (***BUFR***) dynamically skip faults that are incompatible with the current pattern on each thread and thus the compaction lists of faults are also greatly reduced.

Experimental results demonstrate the effectiveness of three schedulings on 18 benchmark circuits. First, comparing to SDC-TPG, ***preemptive scheduling*** of *P4-TPG* accelerates the average runtime by 10.7%, 15.0% and 17.9% under 4, 8 and 12 threads, respectively. Second, under the best case (*vga_enh*), ***proactive scheduling*** further shortens 15.7% total runtime of SDC-TPG when using 12 threads. Last, ***preventive scheduling*** outperforms SDC-TPG by reducing 76.0%, 87.2%, 94.7% and 96.6% more runtime under 1, 4, 8 and 12 threads, respectively. In the end, *P4-TPG* not only achieves the determinism on the test pattern set as the serial TPG, resulting in zero inflation, but also is superior to SDC-TPG in speed-ups under different numbers of threads.

REFERENCES

[1] R. H. Klenke, R. D. Williams, and J. H. Aylor, "Parallel-processing techniques for automatic test pattern generation," *Computer*, vol. 25, no. 1, pp. 71–84, Jan. 1992.

[2] M. J. Aguado, E. de la Torre, M. A. Miranda, and C. Lopez-Barrio, "Distributed implementation of an ATPG system using dynamic fault allocation," in *Proc. IEEE Int. Test Conf.*, Oct. 1993, pp. 409–418.

[3] R. H. Klenke, R. D. Williams, and J. H. Aylor, "Parallelization methods for circuit partitioning based parallel automatic test pattern generation," in *11th Annu. VLSI Test Symp. Dig. Papers*, Apr. 1993, pp. 71–78.

[4] J. M. Wolf, L. M. Kaufman, R. H. Klenke, J. H. Aylor, and R. Waxman, "An analysis of fault partitioned parallel test generation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 5, pp. 517–534, May 1996.

[5] R. Butler, B. Keller, S. Paliwal, R. Schoonover, and J. Swenton, "Design and implementation of a parallel automatic test pattern generation algorithm with low test vector count," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2000, pp. 530–537.

[6] D. Krishnaswamy, E. M. Rudnick, J. H. Patel, and P. Banerjee, "SPITFIRE: Scalable parallel algorithms for test set partitioned fault simulation," in *Proc. 15th IEEE VLSI Test Symp.*, Apr./May 1997, pp. 274–281.

[7] K.-W. Yeh, M.-F. Wu, and J.-L. Huang, "A low communication overhead and load balanced parallel ATPG with improved static fault partition method," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, 2009, pp. 362–371.

[8] X. Cai, P. Wohl, J. A. Waicukauski, and P. Notiyath, "Highly efficient parallel ATPG based on shared memory," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2010, pp. 1–7.

[9] X. Cai and P. Wohl, "A distributed-multicore hybrid ATPG system," in *Proc. IEEE Int. Test Conf. (ITC)*, Sep. 2013, pp. 1–7.

[10] X. Cai, P. Wohl, and D. Martin, "Fault sharing in a copy-on-write based ATPG system," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2014, pp. 1–8.

[11] J. C. Y. Ku, R. H.-M. Huang, L. Y.-Z. Lin, and C. H.-P. Wen, "Suppressing test inflation in shared-memory parallel automatic test pattern generation," in *Proc. IEEE 19th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2014, pp. 664–669.

[12] S. Hadjitheophanous, S. N. Neophytou, and M. K. Michael, "Utilizing shared memory multi-cores to speed-up the ATPG process," in *Proc. 21st IEEE Eur. Test Symp. (ETS)*, May 2016, pp. 1–6.

[13] L. Y.-Z. Lin and C. H.-P. Wen, "Unleashing parallelism with minimal test inflation in multi-threaded test pattern generation," *IEEE Access*, vol. 6, pp. 49269–49281, 2018.

[14] I. D. Dear, C. Dislis, A. P. Ambler, and J. Dick, "Economic effects in design and test," *IEEE Des. Test Comput.*, vol. 8, no. 4, pp. 64–77, Dec. 1991.

[15] K.-W. Yeh, J.-L. Huang, H.-J. Chao, and L.-T. Wang, "A circular pipeline processing based deterministic parallel test pattern generator," in *Proc. IEEE Int. Test Conf. (ITC)*, Sep. 2013, pp. 1–8.

[16] C.-H. Chang, K.-W. Yeh, J.-L. Huang, and L.-T. Wang, "SDC-TPG: A deterministic zero-inflation parallel test pattern generator," in *Proc. 24th IEEE Asian Test Symp. (ATS)*, Nov. 2015, pp. 43–48.

[17] P. Goel and B. C. Rosales, "PODEM-X: An automatic test generation system for VLSI logic structures," in *Proc. 18th Design Autom. Conf. (DAC)*, Jun./Jul. 1981, pp. 260–268.

[18] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing & Testable Design*. Piscataway, NJ, USA: IEEE Press, 1994, p. 672.

[19] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Boston, MA, USA: Springer, 2005.

[20] N. K. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge, U.K.: Cambridge Univ. Press, 2003.

[21] I. Pomeranz, L. N. Reddy, and S. M. Reddy, "COMPACTEST: A method to generate compact test sets for combinational circuits," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 1991, pp. 194–204.

[22] I. Hamzaoglu and J. H. Patel, "Test set compaction algorithms for combinational circuits," in *IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD) Dig. Tech. Papers*, Nov. 1998, pp. 283–289.

[23] S. Neophytou, S. Hadjitheophanous, and M. K. Michael, "On the impact of fault list partitioning in parallel implementations for dynamic test compaction considering multicore systems," in *Proc. 8th IEEE Design Test Symp.*, Dec. 2013, pp. 1–6.

[24] (2011). *Industrial Technology Research Institute*. [Online]. Available: http://www.itri.org.tw/chi/

LOUIS Y.-Z. LIN received the B.S. degree from the Department of Electrical Engineering, National Chung Cheng University, in 2010. He is currently pursuing the Ph.D. degree in electrical and computer engineering with NCTU. His research focused on parallel computing (especially on automatic test pattern generation and circuit reliability in nanometer technologies) and applying data mining and machine learning techniques to system-on-chip design.

CHARLES CHIA-HAO HSU received the M.S. degree in electrical and computer engineering from National Chiao Tung University, in 2017. He has joined the Design Technology Platform, Synopsys, where he focuses on developing testing flow in advance process. His research focuses on enhancing automatic test pattern generation through the deployment of parallel computing.

CHARLES H.-P. WEN (M'07) received the Ph.D. degree in very-large-scale integration verification and test from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 2007. He is currently an Associate Professor with National Chiao Tung University, Hsinchu, Taiwan, where he is also a Specialist in computer engineering. His research has focused on applying data mining and machine learning techniques to system-on-chip designs (especially on statistical soft error rates and circuit diagnosability in nanometer technologies) and cloud computing (especially on performance analysis and architecture design of large-scale datacenters). He received best paper awards from 2012 ASP-DAC and 2014 SASIMI, and the Distinguished Young Scholar Award from the Taiwan IC Design Society.