

Automating Data Analysis and Acquisition Setup in a Silicon Debug Environment*

Yu-Shen Yang, Andreas Veneris, *Senior Member, IEEE*, and Nicola Nicolici, *Member, IEEE*

Abstract—With the growing size of modern designs and more strict time-to-market constraints, design errors can unavoidably escape pre-silicon verification and reside in silicon prototypes. Due to those errors and faults in the fabrication process, silicon debug has become a necessary step in the digital integrated circuit design flow. Embedded hardware blocks, such as scan chains and trace buffers, provide a means to acquire data of internal signals in real time for debugging. However, the amount of the data is limited compared to pre-silicon debugging. This paper presents an automated software solution to analyze this sparse data to detect suspects of the failure in both the spatial and temporal domain. It also introduces a technique to automate the configuration process for trace-buffer based hardware in order to acquire helpful information for debugging the failure. The technique takes the hardware constraints into account and identifies alternatives for signals not part of the traceable set so that their values can be restored by implications. The experiments demonstrate the effectiveness of the proposed software solution in terms of run-time and resolution.

Index Terms—Silicon debug, post-silicon diagnosis, data acquisition setup, test, Boolean satisfiability

I. INTRODUCTION

Developing modern integrated circuits consists of several synthesis stages before a silicon prototype is fabricated. To ensure that each synthesis step does not introduce any errors (e.g. timing, functional, power), a corresponding verification stage is carried out to validate the design. During the pre-silicon process, engineers test devices in a virtual environment with sophisticated simulation [1], emulation [2] and formal verification [3], [4] tools to check the correctness of the RTL model against its functional specification. However, due to the growing complexity of functionality and the size of designs, it becomes infeasible to achieve 100% verification coverage within the strict time-to-market constraints. As such, functional bugs may escape pre-silicon verification and only be discovered during in-system silicon validation where the design is exercised at speed. Consequently, silicon prototypes are rarely bug-free. For example, the work in [5] presents several functional errors in processor designs that are found after the designs are taped out. In fact, more than 60% of

design tape-outs require a re-spin where more than half of these re-spins are due to logical or functional errors not discovered by the pre-silicon verification [6]. Each re-spin due to design errors or faults that reside in the silicon dramatically increases project costs and the time-to-market. Therefore, it is important to develop a silicon debug flow that provides short turn-around time when a silicon prototype fails.

A typical silicon debug process consists of several iterative sessions, referred to as *debug sessions*. Each session can be divided into two stages: *data acquisition* and *data analysis*. In the data acquisition stage, test engineers set up the environment to obtain appropriate data from the chip under test while it is operated in real-time. Unlike pre-silicon verification, where values of any signals can be obtained through simulation, observability of internal signals in the silicon prototype is restricted. Several *Design-for-Debug (DfD)* hardware components, such as scan chains or trace buffers, are used to access the internal signals. Nevertheless, the amount of acquired data is limited by the integrated DfD hardware components. These limits greatly inhibit accurate and effective debugging analysis. During data analysis, the sparse amount of data acquired during the test is analyzed to prune the error candidates and to set up the data acquisition environment for the next debug session. This time-consuming and labor-intensive cycle continues until the root cause of the failure is determined. Clearly, the quality of the data analysis is affected by the acquired data. Hence, software solutions for silicon debug need to have the ability of identifying sets of signals, as well as cycles during the execution, that are important and helpful in narrowing down the suspects. Furthermore, this set of signals should be concise to comply with the hardware limitation.

This paper proposes an automated software-based debug methodology that complements current data acquisition hardware solutions. The proposed methodology is designed as a post-processing step after the data has been acquired. This methodology automates the data analysis process and aids the engineer in discovering the root cause of the chip failure. It identifies the potential locations of the error in a hierarchical manner, and estimates the time interval where the error is excited. When a debug engineer needs to manually investigate the errors in a design, the proposed methodology facilitates a focused approach on a smaller set of locations within more concise windows of cycles. As a result, the amount of manual work performed by the engineer can be reduced. In addition, our methodology helps refine the debugging experiments by setting up the data acquisition environment for the next debug session. It utilizes UNSAT cores to identify registers that may

Y.-S. Yang is with Vennsa Technologies Inc. Toronto, ON, M5V3B1, Canada (Email: terry.yang@utoronto.ca)

A. Veneris is with the Department of Electrical and Computer Engineering and with the Department of Computer Science, University of Toronto, Toronto, ON, M5S 3G4 (Email: veneris@eecg.utoronto.ca)

N. Nicolici is with the Department of Electrical and Computer Engineering, McMaster University, Hamilton, ON, L8S 4K1, Canada (Email: nicola@ece.mcmaster.ca)

* This work was presented in part at *IEEE Design and Test in Europe 2009*, *IEEE Int'l Symposium on Quality of Electronic Design 2010*, and *IEEE Silicon Debug and Diagnosis Workshop, 2010*.

contain useful information to prune the suspect candidates. The new data acquired is fed to the subsequent automated data analysis cycle to eventually determine the root cause. Although the methodology is described in terms of diagnosing design errors, it is later extended to handle physical defects as well. This increases its applicability and its practicality.

When trace buffers are used, not all registers can be accessed in practice. To comply with these hardware constraints, a search algorithm is presented to find alternatives such that their value can restore the value of registers of interest that cannot be traced by the hardware. The proposed search algorithm is memory efficient because only a small window of the complete test trace is analyzed. Finally, because only one traceable register group can be traced in each debug session, a simple ranking system is suggested to prioritize the traceable register groups according to the result from the proposed analysis.

Experiments on OpenCores.org and ISCAS'89 circuits are conducted. Results show that our methodology successfully determines the locations of the error or defect. It also specifies with accuracy the time interval in which the error is excited. Even with the hardware constraints considered, the methodology reduces, on average, 31% of suspects that the engineer needs to investigate with only 8% to 20% of registers traced. To the best of our knowledge, this is the first study that presents a comprehensive analysis of the data acquired with modern in-silicon hardware like trace buffers to reduce the number of iterations during silicon debug.

The remainder of the paper is organized as follows. Section II summarizes prior work on hardware and software solutions for silicon debug, as well as the background material. Section III presents the proposed software solution to silicon debug, while Section IV illustrates the searching algorithm for selecting alternatives for non-traceable registers, and presents a simple ranking system of traceable register groups. The extension of the proposed methodology to deal with physical defects is explained in Section V. Finally, the experimental results and conclusion are given in Sections VI and VII, respectively.

II. BACKGROUND

As mentioned in the introduction, the main difficulty of silicon debug is the lack of access to the internal signals. In this section, two data acquisition hardware components used to enhance the observability of internal signals in chips are discussed. Next, previous automated data analysis algorithms are reviewed. Finally, we summarize the background material for the proposed methodology.

A. Design for Debug Hardware Solutions

The behavior of internal signals in a chip can only be observed if they are routed to external pins. Since numbers of available pins on a chip are limited, this approach may not provide sufficient information to perform debugging. To improve observability of internal signals, two DfD solutions are mainly used in practice: scan chains and trace buffers.

a) *Scan chains*: provide a means to take a snapshot of the registers at a specific cycle. In the test mode, values of all scanned registers can be serially shifted out with a slower shift clock. This operation is referred to as *scan dump*. However, the scan dump operation interrupts the execution of the chip because the functional clocks of the design are halted and the values stored in the registers are destroyed. In order to resume the execution from the same point, the environment needs to be reset and restarted from the initial setup. Note that although non-destructive scans (i.e., scan cells that consist of an additional element for debugging purpose) provide mechanism to resume the execution from the same point, a new state capture cannot occur until previous scan dump has been completed. Hence, it is not practical to acquire state values for several consecutive cycles using scan chains [7].

b) *Trace buffers*: record internal signals in an on-chip memory in real-time. The trace buffer size, which for today's designs typically ranges from a few kilobytes to a few hundreds of kilobytes, is determined by its *width* and its *depth*. The width constrains the number of signals to be probed, while the depth limits the number of samples that can be stored. A trace buffer contains control logic, called trigger logic (e.g., embedded hardware assertions), employed for on-line monitoring of circuit behavior. The logic values of the selected signals are recorded when the trigger condition is asserted. Subsequently, the recorded data is read via a low-bandwidth interface, such as a boundary scan. The advantage of trace buffers is that the values of signals can be collected for consecutive cycles, while scan chains only provide the value in a specific cycle. However, due to the limited size of the embedded memory, only a small set of pre-selected signals can be traced. Those pre-selected signals are divided into groups and connected to the on-chip memory through a multiplexer. During execution, only one group can be traced at a time. The traceable signals are typically manually selected by the designer. Recently, several algorithms have been developed to automate the selection process [8], [9], [10], [11]. In those works, the authors try to select a small set of signals such that their values have a higher chance of restoring a significant amount of untraceable states.

B. Related Work on Data Analysis

Although DfD hardware enhancement increases the observability of internal signals, there is a lack of techniques that automate the data analysis process on the acquired data. Recently, there has been an effort to develop methodologies to aid the engineer in this part of the silicon debug process as summarized in the following.

The method proposed in [12] relies on scan dumps collected at multiple consecutive cycles to determine failing registers at each time frame. Next, it conducts back-tracing from those failing registers to identify the fault propagation paths and suspect registers at each cycle. Finally, it performs a forward-tracing from the suspect registers to further narrow down the root cause candidates.

Yen et al. [13] propose an approach that isolates the critical cycles using a binary search paradigm based on the comparison between the observed data and the simulation results. A

critical cycle is the first cycle in which the state elements show a discrepancy between the expected responses and the actual ones. Then, this method identifies suspect registers with a simple path-tracing method [14] and simulating the golden model with faulty values injected at each suspect candidates.

A formal approach that restores state values of a design in a failing trace is proposed in [15]. It starts from the crash state and computes backward in time. Signatures, computed with additional hardware structures, are captured during the chip execution and stored in the trace buffer. Later, those signatures are used to determine a unique or a small set of possible predecessor states that lead to the crash state.

C. Boolean Satisfiability and UNSAT Cores

This work models the debugging problem into a Boolean Satisfiability (SAT) instance and utilizes the use of UNSAT cores extracted from it to guide the silicon debug data acquisition setup. A brief overview of these methods are given in this section.

SAT proves or disproves whether a Boolean formula Φ has a satisfiable assignment, i.e., the formula is evaluated to true. If such an assignment exists, the formula Φ is said to be satisfiable; otherwise, it is unsatisfiable. For most modern SAT solvers, Boolean formula is presented in *Conjunctive Normal Form* (CNF), which consists of a conjunction of *clauses* where each clause is a disjunction of literals. A *literal* is an instance of the variable or its negation.

If the formula is unsatisfiable, any subset of clauses in the instance that is also unsatisfiable is referred to as an *UNSAT core*. Modern SAT solvers [16], [17], [18] can produce UNSAT cores as a result of proving unsatisfiability. For example, an UNSAT core of the formula, $\Phi = (a+b) \cdot (a+c) \cdot (\bar{b}+\bar{c}) \cdot (\bar{a}) \cdot (\bar{c})$, is $\{(a+c), (\bar{a}), (\bar{c})\}$.

An unsatisfiable SAT instance can have multiple UNSAT cores. Each represents a situation where the instance is unsatisfied. Additional UNSAT cores can be obtained by eliminating a previously found UNSAT core, as described in [19]. In summary, each clauses in an UNSAT core is augmented with a distinct relaxation variable. Additional clauses are added to the CNF formula to ensure one and only one relaxation variable is true. Consequently, this additional formulation removes the constraints applied to Boolean variables and breaks the unsatisfied condition.

D. SAT-based Diagnosis

Modelling the problem of logic and fault diagnosis in SAT is first presented in [20]. Given a circuit and a set of test traces that cause the design to fail, the problem is formulated in a CNF instance such that the SAT solver returns solutions that correspond to error location(s). This is achieved by inserting a multiplexer at every gate (and primary input) such that when the select line (s) of the multiplexer is inactive, the original design is maintained; otherwise, a new unconstrained primary input variable (w) drives the output of the multiplexer. Next, the design is translated into a CNF formula [21] and duplicated for each vector sequence and for each cycle of the test sequence. Note that multiplexers that are driven by the

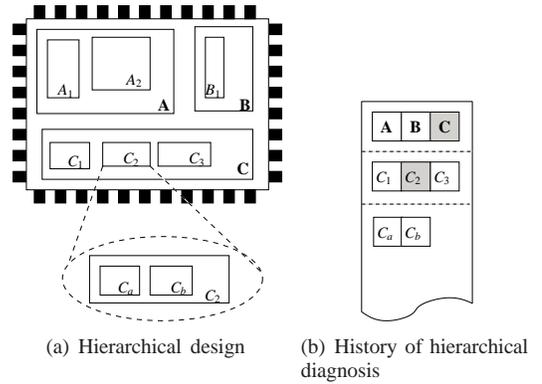


Fig. 1. Hierarchical diagnosis

same gate in each unrolled copy of the circuit share the same select line. This is because these gates represent the same gate in the original sequential circuit. If one of them is the error source, all the remaining should be selected as well. This construction is later constrained with the input vector and the expected output response. Readers can refer to [20] for more information on this SAT-based debugging methodology.

The SAT-based diagnosis algorithm from [20], also used here, performs *model-free* diagnosis [22]. That is, it does not make any assumption on the behavior of the fault/error. This is a desirable fit to silicon debug since silicon prototypes can fail test for various reasons. The engineer can utilize the values of the unconstrained variables w to determine the type of the fault that has occurred.

More recently, unsatisfiability is also used for debugging. In [23], authors utilize UNSAT cores to determine suspects in an erroneous gate-level design. The work in [24] uses MaxSAT to identify the time in which the bug is activated in a trace.

E. Hierarchical Diagnosis

Ali et al. [25] extend the SAT-based diagnosis to debug designs in a hierarchical manner to improve the performance and resolution of logic debugging. The debug process consists of several iterations. In each iteration, the method only considers modules in the same hierarchical level. The procedure starts from the top-level of the design and goes deeper into the design hierarchy. Suspect candidates for debugging in each iteration are sub-modules of the modules that are determined to be suspects in the previous iteration. The procedure is repeated until the lowest level of the design hierarchy is reached.

The formulation of hierarchical diagnosis in SAT is the same as the basic construction of SAT-based diagnosis described in the previous section except that multiplexers are inserted at the output of coarse-grain modules rather than simple gates. Additionally, all multiplexers at the output of the same module share the same select line. Consequently, the SAT solver selects a module as a suspect if it can assign values to some outputs of this module such that the behavior of the design matches the constrained output response for the particular test vector.

Figure 1 illustrates the concept of debugging using hierarchical information. Figure 1(a) shows the hierarchical structure

of a design. A situation in which hierarchical diagnosis is applied to this design with two iterations is shown in Figure 1(b). Diagnosis starts with three top modules. In the first iteration, module C (grey box) is diagnosed to be the suspect. Hence, diagnosis, in the second round, only considers the sub-modules of C , namely, C_1 , C_2 , and C_3 , as candidates. At that round, C_2 is identified as the suspect. As a result, the suspect candidate list for the third round consists of C_a and C_b only.

With the hierarchical information of the design, diagnosis can start with a coarse-grain global analysis and the search can be directed to local areas after each iteration. Such a procedure reduces the runtime and memory requirement, since there are fewer candidates that need to be analyzed.

III. AUTOMATED DATA ANALYSIS

A silicon debug process is different from an RTL debug process in many important aspects. First, silicon debug needs to utilize the DfD hardware components in the design to acquire values of internal signals, whereas, in RTL diagnosis, values of internal signals can be obtained through simulation. Second, due to the vast complexity of the silicon debug problem, a software solution should be designed appropriately to take advantage of the debug hardware available to the engineer to reduce the iterations of the process. Finally, because silicon prototypes are operated at-speed during test, the test trace for debugging can be orders of magnitude longer compared to the one usually available during RTL diagnosis. As such, it is important to identify the segment of the trace that really matters to aid the future iterations of the debug process and simplify the analysis.

A. Assumptions

In this work, the following assumptions are used to make the silicon debug problem more feasible to solve, while still reflect realistic practice concerns.

- The erroneous silicon behavior is deterministic. That is, errors replicate their behavior with the same set of test vectors. This is the case if the circuit is debugged on the tester or on an application board where the input is controlled synchronously. This assumption is necessary to replicate experiments and to obtain the values of multiple state elements at different cycles. It is also the fundamental underlying assumption of a silicon debug environment described in the introduction.
- Scan chains and trace buffers (Section II-A) are utilized to obtain the values of internal states. In this scenario, the design is fully scanned and trace buffers can be programmed to capture the value of specific state elements. Those values are compared with the expected values to determine whether the error is observed.
- The golden model, such as a high-level behavioral model, is available to provide the correct responses of the design. Note that although this behavioral model may not provide access to the data on every single net in the implementation, the important information on the data and address buses, as well as the essential control signals that steer the data through the data-path, can be monitored.

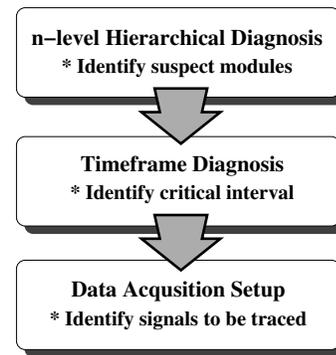


Fig. 2. A single debug analysis session

- All discrepancies are due to a single error present in the RTL representation. Since most test vectors target specific functionality of the design, it is realistic to conclude that a failing test vector is due to a single error [26].

B. Proposed Methodology Overview

The methodology in the following discussion deals with functional errors (bugs) in the design. Examples of functional bugs that escape to silicon can be found in [5]. The extension of the methodologies on physical defects is discussed later.

An overview of the methodology is given here with the details of the implementation described in the remaining subsections. The complete flow of the methodology is summarized in Figure 2. The objective of the proposed methodology has three main goals: to identify the suspect modules that contain the error, to find the critical interval of the error, and to find the registers that may contain helpful information about the error. A *critical interval* is a window of cycles that contains the critical cycle. Unlike for RTL debug, the above objective must be achieved with a conscious usage of the on-chip debug hardware resources. This objective is unique to silicon debug and it motivates the key contributions in this work.

The objectives are achieved in three steps. First, it diagnoses the circuit in a hierarchical manner as described in Section II-E. The algorithm takes in the RTL representation of the erroneous design, failing input test vectors and the expected (i.e., correct) output responses to build the Boolean satisfiability instance. Here, only test vectors that can excite the error and cause discrepancies observed at some primary outputs are examined. Hence, the algorithm identifies design components that are potentially responsible for the functional failure observed during functional test. It has been shown that it is effective to use the design hierarchy information when searching between different components of a design [25]. Unlike in [25] where the debugging algorithm iterates the procedure until the lowest hierarchical level is reached, the algorithm in the proposed flow would only expand at most n hierarchy levels from the level ended in the last session during each debug session. This is referred to as *n-level hierarchical diagnosis*. For example, if $n = 2$ and the maximum hierarchy depth of the design is 10, the algorithm goes deeper in the hierarchy by two levels. As a result, at most five sessions will be performed. Then, timeframe diagnosis is carried out to find

Algorithm 1 Timeframe Diagnosis

```

1:  $\widehat{M}_{List} :=$  list of suspect modules
2:  $k :=$  size of time interval
3:  $T_b(T_e) :=$  beginning(end) cycle of the trace
4: procedure TIMEFRAMEDIAGNOSIS( $\widehat{M}_{List}, k, T_b, T_e$ )
5:    $\widehat{TM}_{sol} :=$  the timeframe diagnosis solutions
6:   for all  $M \in \widehat{M}_{List}$  do
7:      $\widehat{TM}_{List} :=$  the new list containing timeframe modules
8:     for  $t = T_b$  to  $T_e$  incremented by  $k$  do
9:        $TM_{new} :=$  A new timeframe module consists of
           $\{M^t \dots M^{t+k}\}$ 
10:      Add  $TM_{new}$  to  $\widehat{TM}_{List}$ 
11:    end for
12:    Debug with candidates from  $\widehat{TM}_{List}$  and add solutions to
           $\widehat{TM}_{sol}$ 
13:  end for
14:  Critical interval  $(T'_b, T'_e) \leftarrow \bigcup_{TM_i \in \widehat{TM}_{sol}} (T_{bi}, T_{ei})$  where  $TM_i$ 
      is defined over the interval  $(T_{bi}, T_{ei})$ 
15:  return  $(T'_b, T'_e)$ 
16: end procedure

```

a greater precision estimate for the window of clock cycles in which the error may be excited. This interval can further reduce the time interval where the design needs to be analyzed in the next debug session. The trace can also be truncated to start at the same cycle as the begin of the returned interval. The idea is that the segment of the trace before the critical cycle can be safely removed for debugging analysis since it does not contain information related to the error observed (which is excited at the critical cycle). The value of state elements w.r.t. the truncated trace can be initialized with the value of scan dump at this new starting cycle. Moreover, during the test, signals only need to be traced within the new reduced window. Finally, the design with the location of the potential suspects is analyzed to determine a set of registers that can provide more information about the actual bug. The above information feeds back to the proposed analysis flow which iterates the three steps in Figure 2 in the next debug session to aid in further root cause analysis.

Note that, after debugging, engineers still need to inspect each suspect to determine which one is the real error source and fix it. This process can be time-consuming. Hence, the final suspects that require manual inspection should be as few as possible. Furthermore, to ease this process for the engineer, the algorithm should also indicate the time interval wherein the error is excited. In such a way, the test engineer can study a much smaller segment of the complete trace to determine the actual cause of the failure and, consequently, rectify the error. The first step, hierarchical diagnosis, only provides the location but it gives no information about this time frame. For this reason, the second and third steps of the proposed flow are used to improve the resolution of the debugging result in terms of further screening of the error locations and the time frames that they are excited.

The details of timeframe diagnosis and data acquisition setup are discussed in the following subsections.

C. Timeframe Diagnosis

In silicon debug, the depth of the trace buffer limits the number of samples that are acquired in one debug experiment. Once the buffer is full, the older data is overwritten by the new samples. Hence, if the cycle in which the error is exercised can be estimated, the buffer can be utilized more effectively. This unique constraint motivates timeframe diagnosis.

A timeframe diagnosis pass narrows down the critical interval. This result can help to set up the next debug experiment, such that data acquisition starts at the right cycle(s), i.e., the one(s) as close to the critical cycle as possible. Note, the test still runs from the beginning of the test vector sequence. The trace buffer is programmed to begin the capture at a later cycle.

In this work, sequential circuits are modelled in the *Iterative Logic Array* (ILA) representation. The design is unfolded over time to maintain the combinational functionality. Throughout this paper, the superscript of a symbol refers to the cycle of the unfolded circuit. For instance, χ^2 represents the set of the primary inputs in the second cycle. Furthermore, $INPUT(M)$ ($OUTPUT(M)$) denotes the input (output) nets of module M .

Definition 1: Consider an ILA representation of a sequential design. A **timeframe module** TM for a single module M over a set of cycles $\{T_n \dots T_{n+k}\}$ is a conceptual entity that contains the instances $M^{T_n} \dots M^{T_{n+k}}$ of module M over this set of cycles such that $INPUT(TM) = \bigcup_{t=T_n}^{T_{n+k}} INPUT(M^t)$ and $OUTPUT(TM) = \bigcup_{t=T_n}^{T_{n+k}} OUTPUT(M^t)$

Timeframe diagnosis is a SAT-based algorithm as described in Section II-D. Recall, in basic SAT-based diagnosis the same gates in each unrolled time frame form one suspect candidate (since their multiplexers share the same select line). One may think that it is a timeframe module defined over the complete trace. Hence, instead of considering suspects in one timeframe module that is defined over the complete trace, timeframe diagnosis examines suspects in timeframe modules that are sets-of-cycles.

Pseudo-code to identify the critical interval is described in Algorithm 1. Timeframe diagnosis divides the trace into several intervals of width k and constructs a timeframe module for each suspect module returned by hierarchical diagnosis in each interval. That is, the suspect modules in each cycle of the interval are collectively considered as a single suspect by timeframe diagnosis (lines 8–11). Consequently, timeframe diagnosis selects suspects from this new set. In this scenario, a timeframe module is selected if the SAT solver can assign values to the outputs of the timeframe module such that the applied constraints are satisfied (line 12). The final critical interval is the union of intervals wherein all selected timeframe modules are defined (line 14). The union of two time intervals, $(T_{b1}, T_{e1}) \cup (T_{b2}, T_{e2})$, is a new time interval, (T'_b, T'_e) , where $T'_b = \min(T_{b1}, T_{b2})$ and $T'_e = \max(T_{e1}, T_{e2})$.

The formulation of the timeframe diagnosis problem is similar to the one described in Section II-D, except the insertion of multiplexers. In timeframe diagnosis, multiplexers are inserted at the outputs of each timeframe module and share one select line. Timeframe modules are selected as suspects if timeframe diagnosis can assign values at the outputs of those timeframe modules such that the output response of the design

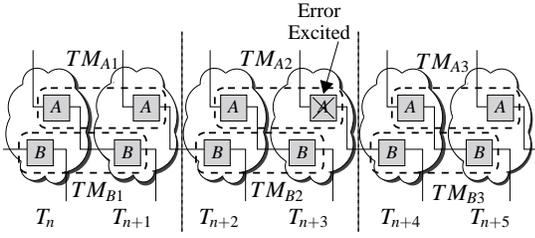


Fig. 3. Timeframe diagnosis

matches the expected response. The following theorem states that one of the selected timeframe modules must be defined over the interval that contains the critical cycle.

Theorem 1: Timeframe diagnosis is guaranteed to select the timeframe module with the respective interval that contains the critical cycle.

Proof: Timeframe diagnosis divides the trace into consecutive intervals of cycles. As such, the cycle when the actual error is triggered must be in one of the intervals. This implies that one of the timeframe modules must contain the critical cycle. Assume, toward contradiction, that the timeframe module that contains the critical cycle (TM_c) is not selected by timeframe diagnosis. This means that the SAT solver cannot re-adjust the value at $OUTPUT(TM_c)$ to make the design comply with the expected output response for the given input vector. However, because the error is excited during the time interval defined by TM_c , the outputs of TM_c must contain erroneous values and, correspondingly, there must be correct values. It follows that the SAT solver can assign the correct values at $OUTPUT(TM_c)$ to eliminate the error effect and make the instance satisfied. Hence, by construction, TM_c must be one of the solutions returned by timeframe diagnosis. ■

It is worth to note that the suspects returned by hierarchical diagnosis may not be the actual error source. They can be modules driving the actual error source or propagating the error effect to the primary outputs. To be more accurate, in addition to the timeframe module containing the critical cycle, the solution also includes timeframe modules defined over the intervals that are (a) before the critical cycle and (b) between the critical cycle and the cycle in which the erroneous effects are observed. In the former cases, the timeframe module is selected because the condition to excite the error can be eliminated, whereas, in the latter cases, the error effect can be masked. Therefore, the resulting critical interval is the union of time intervals in which selected timeframe modules are defined. The following example demonstrates the behavior of timeframe diagnosis.

Example 1: Consider a test vector interval between cycles T_n and T_{n+5} , as shown in Figure 3. From hierarchical diagnosis, it is known that modules A and B, shown in that figure as grey boxes, are suspects. To improve the estimate for the time interval where the error is excited, timeframe modules that consider two cycles at a time (i.e., $k = 2$) are created. These timeframe modules are shown in dotted rectangles (e.g., TM_{A1} consists of $\{A^{T_n}, A^{T_{n+1}}\}$). Assume that the error is excited in module A at cycle T_{n+3} , that is, the grey box marked with an \times . As such, timeframe diagnosis returns solutions consisting

of TM_{A2} and TM_{B3} . Hence, timeframe diagnosis can deduce that the critical interval is (T_{n+2}, T_{n+5}) as defined by TM_{A2} and TM_{B3} .

Since the algorithm guarantees that one of the selected timeframe modules contains the critical cycle, the subsequent analysis can focus on the trace within the critical interval returned by timeframe diagnosis. In Example 1, because TM_{A2} and TM_{B3} are selected, cycles between T_{n+2} and T_{n+5} are analyzed in the next debug session. The value of k defines a trade-off between performance and resolution. The more timeframe modules one has to examine, the more candidates that need to be considered at every iteration of the algorithm. In early debugging sessions, a larger value for k may be more preferable for some coarse-grain analysis. Since failing test vectors can contain many cycles, short timeframe modules will introduce a lot of candidates that take more time to screen. On the other hand, having excessively long timeframe modules intervals may not always be a good practice at later stages.

D. Data Acquisition setup

Due to the insufficient observability of internal signals, determining which set of signals to observe is a key step in the silicon debug process. Trace buffers provide the engineer great flexibility in the choice of traced signals. However, the buffers can only trace a limited subset of signals. In most real-world designs, only a small set of hard-wired signals can be traced during the execution.

Among all traceable registers, the engineer wants to select ones that are related to the error source or provide valuable information to aid in pruning suspects. A simple approach to identify those registers is using X-simulation [22], which simulates the design with logic unknown at the output of the suspects to capture all possible paths for error propagation. Then, any registers that store logic unknown are the candidates for tracing. Because X-simulation is a pessimistic process, it may return too many registers to make the information useful. To improve resolution and accuracy, another selection algorithm that utilizes the proof of unsatisfiability generated by SAT solvers is presented.

As discussed in Section II-C, an UNSAT core of an unsatisfiable SAT problem is a subset of clauses that is also unsatisfiable. Given an erroneous circuit, C , the input vector sequence, v , and the correct output response, y , the CNF formula of the ILA representation of the circuit $\cup_{i=1}^L (C^i \cdot v^i \cdot y^i)$, where L is the length of the sequence, is unsatisfiable due to the contradiction between the erroneous output response and the correct output response. Intuitively, the contradiction can occur at any signals along the paths from the actual fault location to the output where discrepancies are observed. Therefore, signals associated with clauses in the UNSAT cores can be potential locations for tracing and provide information about the behavior of the failure.

Example 2: Consider the circuit shown in Figure 4(a). Assume the error is at i , where the correct implementation is $i = \text{AND}(a, b)$. The test vector and the correct/erroneous response are shown in Figure 4(b). Since the circuit is erroneous, the CNF formula, $\Phi = \cup_{i=1}^4 (C^i \cdot v^i \cdot y^i)$, is unsatisfiable. Due

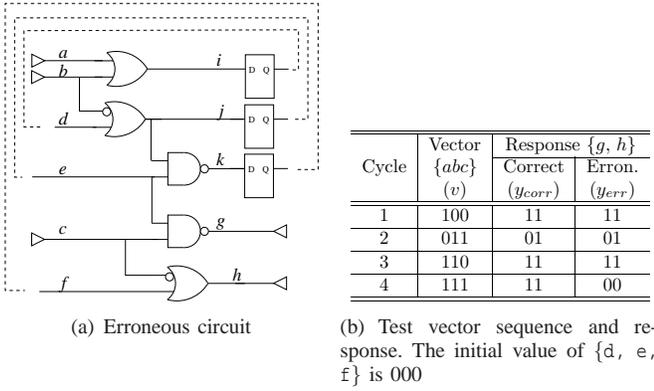


Fig. 4. Example erroneous circuit. The correct implementation of gate $i = \text{OR}(a, b)$ is $i = \text{AND}(a, b)$

to the space limitation, the formulation of Φ is not shown. However, the construction can be done in linear time as shown in [21]. Given Φ to the SAT solver (e.g., MiniSAT [18]), an UNSAT core of the instance can be extracted from the proof of unsatisfiability provided by the solver as shown below.

$$\{(j^3 + e^4) \cdot (\overline{c^4} + \overline{e^4} + \overline{g^4}) \cdot (\overline{d^3} + j^3) \cdot (i^2 + d^3) \cdot (\overline{b^2} + i^2) \cdot (c^4) \cdot (g^4) \cdot (b^2)\}$$

By examining the UNSAT core, variables that represent registers can be extracted: d^3 (from the clause $(\overline{d^3} + j^3)$) and e^4 (from the clause $(j^3 + e^4)$). Therefore, signals that should be traced are d at cycle 3 and e at cycle 4.

The overall algorithm is shown in Algorithm 2. The goal is to identify as many UNSAT cores as possible and extract registers from each UNSAT core. Since each UNSAT core is one potential error propagation path, registers involved with these UNSAT cores are potentially on the error propagation paths. To obtain multiple UNSAT cores, the algorithm iteratively eliminates UNSAT cores until the problem is satisfied.

The procedure of the algorithm is as follows. It starts by obtaining the initial UNSAT core (\mathcal{U}_{init} in line 7). Then, the algorithm tries to obtain more UNSAT cores through relaxation, as summarized in Section II-C. First, it relaxes clauses in \mathcal{U}_{init} that represent input vectors (line 10) until the problem is satisfied. Next, it repeats for clauses in \mathcal{U}_{init} that represent output responses (line 16). Since each UNSAT core can represent different error propagation paths, different signals can be included. To ensure that all paths are considered, the union of all UNSAT cores is taken, as shown in line 12 and line 17 in the algorithm. Finally, if the corresponding variables of registers appear in any UNSAT cores, these registers are the potential locations for tracing.

Example 3: Continue from Example 2, another UNSAT core can be obtained by relaxing (g^4) , which is an output constraint. Let r_1 be the new relaxation variable. To relax the constraint, (g^4) is replaced with $(g^4 + r_1)$. Moreover, an additional clause (r_1) is added to the original Φ . This results in a new formula which is still UNSAT and a new UNSAT core can be obtained as follows:

Algorithm 2 UNSAT-core-based register selection

```

1:  $C :=$  The erroneous design
2:  $\mathcal{V} :=$  Input vectors
3:  $\mathcal{O} :=$  Output vectors
4:  $\Phi := C \cdot \mathcal{V} \cdot \mathcal{O}$ 
5: procedure IDENTIFYTRACEDSIGNALS( $\Phi$ )
6:    $\Phi_{init} \leftarrow \Phi$ 
7:    $\mathcal{U}_{init} :=$  Solve  $\Phi$  and extract the UNSAT core
8:    $\mathcal{U} \leftarrow \mathcal{U}_{init}$ 
9:   while  $\Phi$  is unsatisfiable do
10:    relax on clauses  $\{c | c \in \mathcal{U}_{init} \text{ and } c \text{ is an input vector unit clause}\}$ 
11:     $\mathcal{U}_{new} \leftarrow$  solve  $\Phi$  and extract the UNSAT core
12:     $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{U}_{new}$ 
13:  end while
14:   $\Phi \leftarrow \Phi_{init}$ 
15:  while  $\Phi$  is unsatisfiable do
16:    relax on clauses  $\{c | c \in \mathcal{U}_{init} \text{ and } c \text{ is an output response unit clause}\}$ 
17:     $\mathcal{U}_{new} \leftarrow$  solve  $\Phi$  and extract the UNSAT core
18:     $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{U}_{new}$ 
19:  end while
20:   $\mathcal{R} \leftarrow$  extract registers in  $\mathcal{U}$ 
21:  return  $\mathcal{R}$ 
22: end procedure

```

$$\{(\overline{a^1} + i^1) \cdot (\overline{i^1} + d^2) \cdot (\overline{d^2} + j^2) \cdot (\overline{j^2} + e^3) \cdot (\overline{b^2} + i^2) \cdot (i^2 + d^3) \cdot (\overline{d^3} + j^3) \cdot (\overline{e^3} + \overline{j^3} + \overline{k^3}) \cdot (\overline{c^4} + \overline{h^4} + f^4) \cdot (k^3 + f^4) \cdot (a^1) \cdot (b^2) \cdot (c^4) \cdot (h^4)\}$$

In the new UNSAT core, variables that represent registers are $\{d^2, d^3, e^3, f^4\}$. Hence, the new list of registers-to-be-traced contains d at cycles 2 and 3, e at cycles 3 and 4, and f at cycle 4.

Note that the proposed algorithm may not explore all propagation paths. If an error is propagated to a primary output by following various paths, it is possible that only some of the paths are explored by the algorithm. However, since the purpose of this step is to help the engineer to select registers for tracing during the data acquisition stage, a complete set of solutions may not be necessary.

IV. ALTERNATIVE SIGNAL SEARCHING

The algorithm IDENTIFYTRACEDSIGNALS from Section III-D selects a list of registers that may contain useful information about the behavior of the faulty chip. One way to obtain the values of those registers is through the use of scan dumps, if they are scannable. Nevertheless, this approach can be impractical. As explained in Section II-A, to acquire data at different cycles with the scan dump operation, test needs to be reset and started over again after each dump, a process that can be time inefficient.

Another approach is tracing these registers with trace buffers. Recall, not all registers can be traced with the trace buffer. In this case, one can try to obtain the value of non-traceable registers indirectly by implication using other traceable registers.

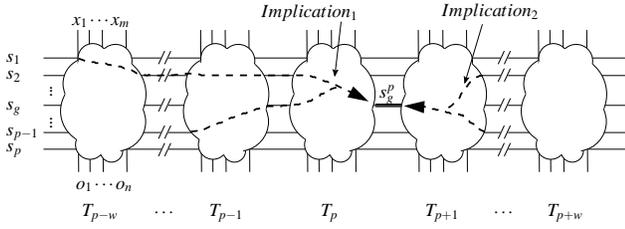


Fig. 5. An unfolded circuit from T_{p-w} to T_{p+w}

Consider a circuit modelled in the ILA representation shown in Figure 5. Let s_g^p denote the untraceable register s_g at cycle T_p of which the values are desired.

Given a set of traceable registers, s_t , referred to as *candidate registers*, the goal is to find a subset of traceable registers, $\{s'_t \subseteq s_t\}$, such that the values of the registers in s'_t can imply the value of s_g^p . Therefore, instead of tracing s_g , registers in s'_t are traced. Then, the value of s_g^p is restored with the values of the registers in s'_t . The restoration can be due to forward implications, backward justifications or both.

A SAT instance is formulated to identify these implications. The instance is satisfied if the SAT solver can assign values to a subset of candidate registers that, together with the input and output trace, imply the value of the target register. Consequently, the alternative for the target register consists of those selected candidate registers. The details of the formulation are given in the following subsections.

A. Problem Formulation

The basic problem formulation is presented in this section. The formulation consists of two components. The first component models the circuit between $\{T_{p-w} \cdots T_{p+w}\}$. Variable w is user-defined and referred to as *window_size*. This interval constrains the search space where the SAT solver can search for implications to the target register. The second component of the formula limits the number of candidate registers used for generating implications.

Candidate registers are traceable registers within the interval $\{T_{p-w} \cdots T_{p+w}\}$. In order to indicate whether a candidate register is selected for generating an implication, new variables, called *select variables* and denoted as $\mathcal{L} = \{l_1, l_2, \dots\}$, are added for every candidate register at each cycle. When a select variable is assigned with logic 1, it indicates that the corresponding candidate register is used to produce the implication.

If the formula is satisfied, each solution to the problem is one possible implication for the target register under the given input vector. Candidate registers wherein the select variable, l , is active are the necessary registers to generate the implication. Because traceable registers in each cycle have one unique select variable, the algorithm identifies not only the registers, but also cycles where those registers are located in order to generate the implication.

In detail, the SAT instance can be expressed as follows:

$$\Phi = \left[\prod_{j=p-w}^{p+w} \Phi_c^j(\mathcal{L}^j, \mathcal{V}^j, \mathcal{Y}_{obv}^j, S_{known}^j) \right] \cdot E_N \left(\bigcup_{j=p-w}^{p+w} \mathcal{L}^j \right) \quad (1)$$

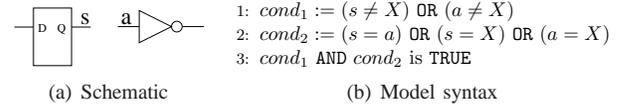


Fig. 6. The model of target registers

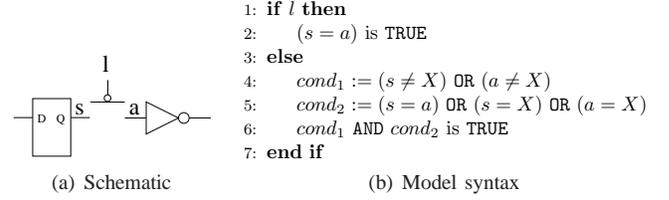


Fig. 7. The model of candidate register

The first component, $\prod_{j=p-w}^{p+w} \Phi_c^j(\mathcal{L}^j, \mathcal{V}^j, \mathcal{Y}_{obv}^j, S_{known}^j)$, models the design from cycle T_{p-w} to T_{p+w} . Each Φ_c^j represents a copy of the erroneous design at cycle j with input vector \mathcal{V}^j and observed response \mathcal{Y}_{obv}^j enforced at the primary inputs and the primary outputs, respectively. Previously traced register values (S_{known}^j) are also used to constrain the problem, since they may be helpful in generating implications. As will be explained in the next subsection, special CNF models are required for the target register and candidate registers.

Although the value w is user-defined, it also depends on the size of the trace buffer. One can set w such that $2w + 1 = \text{buffer depth}$ to fully utilize the memory space of the trace buffer. However, larger w 's can increase the computation complexity and memory consumption, since there are more candidate registers for selection and a larger portion of trace is analyzed. The flexibility of w allows the user to adjust it according to the available resources.

The second component, $E_N(\bigcup_{j=p-w}^{p+w} \mathcal{L}^j)$, constrains the number of selected candidate registers. It is an adder that sums up the value of select variables. The details of the construction can be found in [20]. To find the minimum number of candidate registers required for implications, the output of the adder is constrained to allow one active select variable, and the value is incremented until a solution is found or the total number of the select variables is reached.

The search algorithm is carried out to find alternatives for each untraceable register selected by the UNSAT core selection algorithm. Since each untraceable register may have different required sets of traceable registers and because only one group of traceable registers can be traced in a single debug session, a simple ranking system is discussed later to prioritize each group. Then, the group with the highest priority is traced.

In the next subsection, the models for target registers and candidate registers are described.

B. Register Modelling

Target registers and *candidate registers* need to be encoded specially in the CNF formula in order to solve the searching problem described above. In this section, models applied to these two types of registers are discussed.

Target Register: The goal of the target register s_g^k is to have a logic 0 or logic 1. The implication can come from two

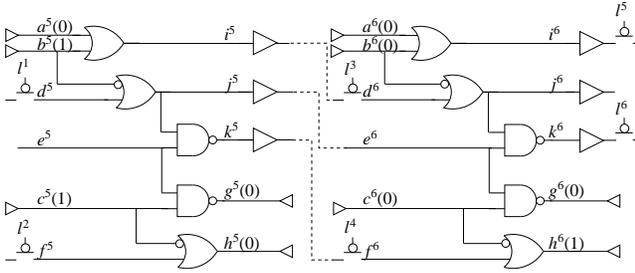


Fig. 8. ILA of the example circuit in Figure 4(a)

directions: forward propagation from assignments in the earlier time frame, or the backward justification from assignments in a later time frame. To allow the SAT solver to consider implications from both directions, the target register is modelled, as shown in Figure 6. An extra signal is introduced to disconnect s_g^k from its fanouts. If either variables have a logic 0 or a logic 1 value, there exists an implication. This is stated as $cond_1$ (line 1) in Figure 6(b). Condition $cond_2$ enforces that the implication only needs to be satisfied from one direction. Furthermore, if there are implications from both directions, the implied values have to be the same.

Candidate Register: Candidate registers are traceable registers that are available for the SAT solver to select in order to generate implications. For each candidate register, two variables are introduced as shown in Figure 7(a). The select variable, l , determines whether the register connects to its fanout. When l equals 0, the network remains the same (line 1-2 in Figure 7(b)). When l equals 1, the register is disconnected from its fanout, and the SAT-solver can assign 0 or 1 to the either end of the break. This enables the possibility to identify forward and backward implications. Similar to the model for target registers, at least one of the two variables at the disconnected ends must be either logic 0 or logic 1. If both ends are not unknown, the values must be the same.

Example 4: Figure 8 shows a portion of the ILA of the example circuit in Figure 4(a). Assume that traceable registers are d and f , and the target register is e^6 . Let the value of the input/output trace as shown in the brackets next to the variables. The candidate registers are $\{d^5, f^5, d^6, f^6, d^7, f^7\}$, which are modeled as shown in Figure 7 with six additional select variables, $\{l^1 \dots l^6\}$. One can verify that the value of e^6 can be restored if the value of d^5 is known.

C. Formulation Improvements

As discussed in Section II-A, traceable registers are typically divided into groups. When configuring the trace buffer, one group of the traceable registers is selected and traced for several time frames. With this observation, the number of select variables for the candidate registers can be reduced. Instead of introducing one distinct select variable for each candidate register, all registers in the same group can share the same select variable. Furthermore, the same register in different time frames can share one select variable as well. In Example 4, assuming d and f are in different groups,

the number of select variables can be reduced to two, e.g. d^5, d^6, d^7 share one, while f^5, f^6, f^7 share another one.

The second optimization is to find implications for a group of target registers. As mentioned in Section III-D, target registers identified by the proposed method are correlated to each other. Hence, if there exists an implication for one of the target registers, the same implication may as well imply the value of other target registers. By grouping several target registers together, the number of executions of the searching algorithm can be reduced. As a result, the overall runtime is reduced. However, it is a trade-off between the runtime and the precision of solutions, because more traceable registers may need to be selected when multiple registers are targeted.

D. Group Ranking

The algorithms described in previous sections identify registers that should be traced to provide more information on the error. Since registers are selected by groups at the end when configuring the trace buffer, a simple ranking system is described to prioritize the traceable register groups according to the results from the proposed algorithms.

- Rule 1: The group that contains the most registers returned by the algorithm IDENTIFYTRACEDSIGNALS has the highest priority. This is because those registers are directly related to the error source. Their values may contain most useful information.
- Rule 2: When searching alternatives for non-traceable registers, different target registers may require different traceable groups. If a group is being selected at higher frequency than other groups, it gets a higher rank. Intuitively, this group contains registers that have a higher chance to provide implications to non-traceable registers.
- Rule 3: A higher rank is assigned to the group that needs to be traced for more time frames. This is simply done to efficiently utilize the memory space of the trace buffer.

V. APPLICATION TO PHYSICAL DEFECTS

Although the presented methodology assumes that errors in the silicon prototype are functional errors, it can also apply to debug physical defects with minor modifications. This is possible because the underlying debug algorithm is a model-free one that works with both errors and faults [20]. It is also because most physical defects can be modeled in terms of design errors as extensively discussed in [27], [28].

When debugging functional errors as shown here, the input to the methodology is an erroneous RTL model that is implemented in an erroneous silicon prototype. In this scenario, the methodology tries to identify these error locations in the RTL model such that when corrected, the model complies with the golden RTL reference available. In contrast, when debugging physical defects, the RTL model is assumed to be correct. In this case, the algorithm identifies the source of the error by inserting *incorrect* faulty values at locations in the correct RTL model so that its behavior matches this of the failing silicon [20]. This is achieved by constraining the SAT instance of the correct RTL with the observed failed responses from the silicon. Once candidate fault locations are identified, the test

TABLE I
PERFORMANCE WITH LIMITED HARDWARE CONSTRAINTS

Circ.	Gate count	# of reg.	# of modules	# of sessions	Total time (sec)	Total groups traced	# of final susp.	init. trace length (# time frames)	% of critical interval
divider	6419	510	31	4	123.1	7	11	38	12%
spi	2832	162	79	4	351.5	6	12	213	11%
wb	5283	110	94	3	101.4	3	6	187	14%
rsdecoder	11353	521	481	4	162.2	5	15	136	10%

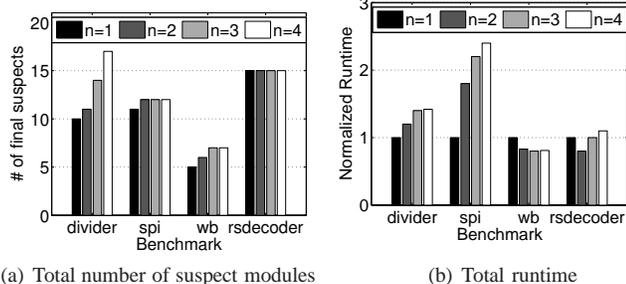


Fig. 9. Impact of parameter n in performance

engineer can use the available mapping information with the silicon prototype to probe further and analyze their source.

VI. EXPERIMENTS

In this section, experiments on OpenCores.org designs and ISCAS'89 benchmarks are presented. Minisat [18] is used as the underlying SAT-solver. Experiments are conducted on Core 2 Duo 2.4GHz process with 4 GB of memory. All runtimes are reported in seconds. In each testcase, a single random functional error (e.g., wrong assignment, incorrect case state, etc) is inserted into the RTL code. For designs from OpenCores.org, test vectors are extracted from the test-bench provided by OpenCores.org. Test vectors for ISCAS'89 are generated randomly. In both cases, the trace length is between 100 to 300 time frames. Finally, to fully take advantage of hierarchical diagnosis, building blocks of HDL code, such as a case statement or an if-statement, are parsed as a module.

A. Performance of the Methodology

This set of experiments first shows the performance of the methodology. Here the algorithm is configured such that during hierarchy diagnosis, it analyzes two levels in the hierarchy structure ($n = 2$) in each debug session. During timeframe diagnosis, the trace is divided into four timeframe modules of an equal number of cycles each. X-simulation is used to determine registers that should be traced in the next debug session. The size of the trace buffer is assumed to be 16×128 bits. It is assumed that 80% of registers in each design are traceable and they are divided into groups of at most 16. In each debug session, the buffer can store values of one group for at most 128 cycles or two groups for at most 64 cycles.

Table I outlines performance metrics for the methodology. Each experiment contains the average of five runs. The test-bench used is listed in the first column. The size of each test-bench in terms of the number of primitive gates is reported in

the second column. The next two columns record the number of registers in the design and the number of the modules at the lowest level of hierarchy. This is also the total number of suspects one needs to examine in a brute-force manual silicon approach. The number of debug sessions and the total runtime for all sessions are shown in the fifth and columns, respectively. The total number of groups that are traced is shown in the seventh column. The eighth column has the number of final suspects in the lowest level of hierarchy that the engineer needs to investigate. The final two columns show the initial trace length in terms of the number of time frames and the ratio of the final critical interval compared to the initial trace length, respectively.

Overall, comparing the number of final suspects to the number of modules shown in the third column, on average, an 85% improvement in resolution is observed. The experimental results also show that the critical interval can be narrowed down to only 10% to 15% of its initial length after the last debug session. Furthermore, as mentioned earlier, one or two groups of registers can be traced in each session. Taking divider as an example, seven groups are traced during debugging: one group is traced during the first session and two groups are traced in each of the remaining three sessions. Because timeframe diagnosis often reduces the critical intervals to more than half in the first one to two sessions, two register groups can be traced in one hardware run in many sessions.

Next, the impact of two parameters of the diagnosis methodology is examined, namely the level of hierarchy that hierarchical diagnosis examines at each session (n), and the timeframe module interval sizes used in timeframe diagnosis (k). Figure 9(a) shows the total numbers of modules returned by each hierarchical diagnosis round when various numbers of hierarchy levels are examined in one debug session. In general, the numbers are increased as hierarchical diagnosis runs more rounds in one debug session. This is because fewer state values are available and the diagnosis algorithm cannot distinguish some of the suspects. The runtime is plotted in Figure 9(b) and is normalized by comparing it to the runtime of $n = 1$ for each benchmark. As shown, the runtime increases as n increases. This is because more suspects need to be analyzed when more hierarchical diagnosis runs are executed in one debug session. Recall that timeframe diagnosis is carried out after the completion of n -level hierarchical diagnosis. Hence, with smaller values of n , although there are fewer numbers of suspects, a greater overhead due to timeframe diagnosis is required. As the result, in some cases the best runtime happens when $n = 2$.

Figure 10(a) shows the ratio of the size of the critical interval after the last debug session compared to the original

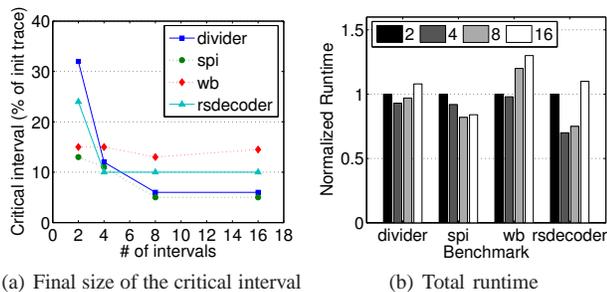


Fig. 10. Impact of number of intervals used in timeframe diagnosis

trace length when various numbers of intervals are used in timeframe diagnosis. Four cases are considered: 2, 4, 8 and 16 intervals. As expected, greater reductions are achieved with finer-grain intervals. The only exception is *wb* in the case where the interval size is 16. In this case, the error happens to be excited across two intervals, which results in a wider range. In all cases, over 50% of reduction is achieved. The normalized runtime is depicted in Figure 10(b). In general, as discussed in Section 3, it requires more computation if smaller intervals are used, since timeframe diagnosis has more candidates to screen. However, using *spi* as example, its runtime is reduced as the number of intervals increases. This is because approximately 90% of the trace interval is truncated after the first few sessions when the number of intervals is over eight. As a result, timeframe diagnosis in the latter debug sessions has a much smaller trace window to analyze and it requires less computation.

B. Performance with Hardware Constraints

This part of the experiment section demonstrates the effectiveness of the UNSAT-core register selection, as well as the searching algorithm. To emulate the real trace buffer hardware structure, a subset of registers of each design is selected randomly, or by *State signal selection* [8], [9], as traceable by the trace buffer. These registers are divided into groups. The grouping configuration is summarized in Table II. The first column lists the circuits used in the experiments; the size of each circuit in terms of the number of primitive gates is reported in the second column. The third column of the table shows the total number of registers in each design. The fourth and fifth columns have the number of the register groups and the number of registers in each group, respectively. The sixth column shows the percentage of total registers that can be traced.

Similar to the experiments in the previous subsection, a single random functional error is inserted into the RTL code. The algorithm is configured to perform one-level hierarchical diagnosis ($n = 1$) and timeframe diagnosis divides the time interval into two timeframe modules. For the searching algorithm, the window size (w) is set to be six time frames and, as mentioned in Section IV-C, the target registers in every four time frames are targeted together.

Table III summarizes the performance of debug analysis under two situations: debug without values of registers (columns 2 – 4) and debug with values of registers selected

TABLE II
TRACEABLE REGISTER GROUP INFORMATION

Circ.	Gate count	Total reg.	# of groups	# of reg./group	Perc.
<i>spi</i>	2832	162	8	8	40%
<i>hpdmc</i>	20536	453	16	8	28%
<i>usb</i>	39179	2054	32	16	25%
<i>s1423</i>	753	74	6	6	49%
<i>s5378</i>	3042	179	7	8	31%
<i>s9234</i>	5883	211	8	8	30%

by the UNSAT core-based selection procedure (columns 5 – 11). Experiments in the former situation are cases where the values of internal states are not used in debugging analysis. The debugging problem is solved with constraints on the primary inputs and primary outputs. As in the latter situation, debugging analysis utilizes the values of internal states that were selected by the proposed state selection procedure as well. Each row is one individual case that contains a different bug in the design. The final row is the geometric mean of the data in the columns. The sum of the number of modules returned at the end of each debug session is shown in the second and fifth columns. This is the total number of modules that the engineer needs to investigate. As shown in the table, with state values, the debugging tool can effectively eliminate more false candidates in all cases. The percentage reduction in the number of suspects, the ratio of the fifth column to the second column, is listed in the sixth column. The reduction can be as high as 78% (i.e., case 1 of *s1423*).

The third and seventh columns show the number of debug sessions performed. About one third of cases require fewer debug sessions to find the root cause of the failure, for example, the second case of *spi*, *hpdmc* and both cases of *usb*. The number of registers traced by the trace buffer is shown in the eighth column. Those numbers are small compared to the total number of registers shown in Table II. The benefit of the UNSAT-core-based technique is shown when one considers the reductions in both the number of suspects and the number of debug sessions. Furthermore, the results indicate that the proposed register selection technique is capable to support data acquisition, although the technique is not a complete solution.

Finally, the runtime of the diagnosis procedure in both situations is reported in the fourth and ninth columns. In the case of the proposed methodology, the additional runtime for searching the registers for tracing is recorded in the 10th column and the total runtime is shown in the 11th column. Because of the reduction of suspects and debug sessions, the runtime for diagnosis is reduced in the cases of the proposed methodology. However, the proposed methodology requires additional computation for the searching algorithm. As shown in the table, the additional runtime can be significant in cases such as *hpdmc*. This is because the algorithm has a higher failing rate on finding the recommendation for non-traceable registers in those cases.

Overall, an average of 31% reduction in the number of suspects and 12% reduction in the number of sessions (from 9.5 down to 8.4) are achieved. The runtime for diagnosis is

TABLE III
PERFORMANCE OF DEBUGGING WITH PROPOSED TECHNIQUES

Circ.	No state value used			With UNSAT-core-based register selection						
	# of susp.	# of sessions	Runtime (s) Diag.	# of susp.	% reduction	# of sessions	# of traced sig.	Runtime(s)		
								Diag.	Search	Total increased
spi	146	11	1990	73	50%	11	24	828	1011	0.92
	144	11	179	76	48%	9	32	101	94	1.09
hpdmc	213	17	3817	170	21%	17	40	2323	15734	4.73
	167	16	2321	131	22%	15	40	1963	14233	6.98
usb	103	15	3795	38	74%	11	64	1609	9218	2.85
	224	14	7091	138	39%	7	128	4245	18519	3.49
s1423	438	6	847	13	78%	6	6	19	28	0.06
	506	6	768	148	71%	6	18	452	36	0.64
s5378	103	6	549	92	11%	6	16	456	288	1.36
	191	6	1577	164	25%	6	32	1505	634	1.36
s9234	83	6	1042	74	11%	6	16	1011	1553	2.46
average	179	9.5	1426	83	31%	8.4	28	684	1012	1.43

52% less on average (from 1426s down to 684s). Due to the searching algorithm, the total runtime of the proposed methodology is about 1.43 times longer than the runtime when no register data is used. However, since the number of the final suspects is reduced significantly, this additional runtime may be acceptable if there is a greater amount of time saved by manually inspecting fewer suspects.

The next experiment examines the performance of the alternative searching algorithm. Clearly, the performance of the algorithm depends on the availability of traceable signals. Some signals may not be able to be restored at all if the necessary registers are not traced. Hence, in addition to selecting the traceable registers randomly, *state signal selection* is also used. *State signal selection* selects registers with values that are more likely to restore other registers of which values are unknown. The results are summarized in Table IV. Due to the technical implementation, *state signal selection* only handles ISCAS benchmarks. Hence, there is no result for all OpenCores.org designs, as indicated by “-”.

The second and fourth columns of Table IV show the percentage of non-traceable registers for which the search algorithm successfully finds alternative recommendations. The number of traceable register groups selected in order to generate implications is shown in the third and fifth columns. In the case of the random selection, the algorithm is, on average, able to find an alternative for almost half of the targets. The performance of the searching algorithm in the cases where pre-selected traceable registers are chosen by *state signal selection* and by the random selection is similar. This is possible because the main goal of *state signal selection* is to restore as many registers as possible over the whole design [8], [9]. The procedure does not target a specific region of the design.

The next set of experiments investigates the performance of debugging when various state values are available. The experimental results are summarized in Table V. All numbers are the average of the 11 erroneous benchmarks discussed in Table III. The reference case for comparison is the case wherein no state value is used (columns 2 – 4 of Table III). The first column lists the four considered cases. The next two columns summarize the reduction of the number of suspects and the number of sessions. The fourth column is the ratio of

TABLE IV
PERFORMANCE OF THE SEARCH ALGORITHM

Circ.	Random		State signal selection	
	succ. rate	# cand. sel.	succ. rate	# cand. sel.
spi	100%	6.8	-	-
	100%	4	-	-
hpdmc	25%	11	-	-
	40%	11	-	-
usb	13%	8	-	-
	6%	8	-	-
s1423	100%	1	100%	2
	100%	3.5	100%	4
s5378	100%	6.3	100%	7
	100%	6.3	100%	7
s9234	50%	1	50%	1
average	49%	4.7		

TABLE V
IMPACT OF STATE VALUES ON THE DIAGNOSIS

Cases	Susp. reduc.	Sess. reduc.	Traced signals	Diag. runtime reduc.
X-sim with no constraint	82%	21%	56%	77%
UNSAT with no constraint	85%	26%	16%	89%
UNSAT with no search	27%	10%	8%	15%
UNSAT with search	31%	11%	10%	26%

traced registers to the total number of registers, followed by the reduction in the diagnosis runtime.

To demonstrate the advantage of the proposed UNSAT core approach, we compare it with X-simulation as shown in the first two cases of the table. In these two cases, no hardware constraints are considered; that is, all registers are assumed to be traceable. The table shows that the UNSAT core approach outperforms the X-simulation approach in all columns, particularly with respect to the number of traced registers. This demonstrates that the UNSAT core approach can achieve better performance with fewer register values. For the second two cases, only debugging with the UNSAT core approach is considered, as well the trace buffer hardware constraints. However, in the third case, the searching algorithm is not executed to find alternatives for non-traceable registers. This means that none of those registers, or any other alternatives,

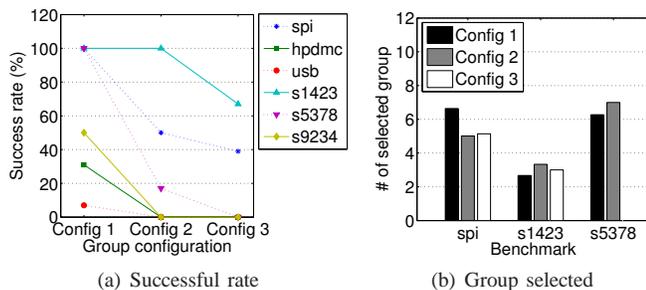


Fig. 11. Performance of the search algorithm with three trace buffer group configurations. All have the same number of groups, but the number of registers per group is 4:2:1

are traced. Comparing the results of the case 3 and the case 4, it shows that, with the help of the searching algorithm, the debugging process performs better. For example, the reduction of suspects increases from 27% to 31%. This implies the effectiveness of the searching algorithm.

In the last set of the experiments, we investigate the performance of the searching algorithm when three different hardware group structures are used. Config 1 is the configuration in Table II. Config 2 and Config 3 have the same number of traceable groups as Config 1 does, but the number of registers in each group is only half and quarter of the size in Config 1, respectively. For instance, Config 1 of hpdmc has 16 groups of the size of eight registers; Config 2 has 16 groups of the size of four registers, while Config 3 has 16 groups of the size of two registers.

The success rate on finding an alternative of non-traceable registers is plotted in Figure 11(a). As expected, since there are fewer traceable registers, more non-traceable registers cannot be replaced. Hence, the success rate drops as the number of candidates is reduced.

Figure 11(b) depicts the average number of selected traceable groups for generating implications. When the searching algorithm is executed with a trace buffer configuration where each group contains fewer traceable registers, two situations can happen: (i) more groups are required since each group contains fewer registers, (ii) the algorithm fails to find the alternatives because the crucial registers are not traceable anymore. In general, more traceable groups are required due to the situation (i), for instance, Config 3 of spi and Config 2 of s1423. However, because the reported number is the average number of selected groups for target registers that the algorithm can find an alternative recommendation, the average can decrease if the situation (ii) occurs and cases where the search algorithm fails to find an alternative require a greater number of groups previously. This is what is observed in Config 2 of spi and in Config 3 of s1423.

C. Performance on Physical Defects

The last experiment applies the methodology to single stuck-at faults. Its performance is summarized in Table VI. Columns two and three show the reduction of returned suspects and debug sessions with the proposed flow. The total runtime comparison is reported in the last column. Overall, the

TABLE VI
PERFORMANCE OF DEBUGGING STUCK-AT FAULTS

Circ.	Suspects reduced	Debug sessions reduced	Total runtime increased
spi	26%	0%	1.64
	69%	0%	0.72
hpdmc	0%	0%	1.63
	41%	57%	6.60
usb	48%	82%	0.69
	37%	44%	1.37

algorithm can successfully identify the location of the stuck-at fault in all cases. Similar results as shown in Table III are observed in Table VI. Fewer suspects are returned and fewer debug sessions are required due to the availability of values of internal states. The total runtime increases because of the overhead of the searching algorithm. However, in some cases, such as case 2 of spi and case 1 of usb, the total runtime is reduced as the result of fewer suspects or debug sessions.

VII. CONCLUSION

Automated software silicon debug solutions are the necessity today to ease the task of the test/design engineer during chip failure analysis. In this paper, we propose a novel debugging methodology that comprises of multiple iterative debug sessions. In each session, the methodology uses the circuit hierarchy to debug the failure and also narrows down the window of cycles wherein the error or fault is exercised. Since the debug analysis relies on the data acquired during the test run, two techniques are proposed to aid in selection of traceable registers to be traced in the next debug session such that the diagnosis can benefit from the new data. The experimental results confirm the effectiveness of the approach. It also demonstrates that the methodology maintains good performance under the constraints presented by the data acquisition hardware.

As future work, several techniques can be investigated to increase the scalability of the proposed methods. For example, abstraction and refinement [29] reduces the problem size by abstracting the implementation of portions of the design. Later, it refines the model to improve the resolution of the result. Another example is to use vector compression [30], [31] to shorten the erroneous traces, which results in a smaller problem for debugging. Those techniques and the generation of tests that exercise specific portions of a design can aid the silicon debug step to localize the errors in a more effective manner.

REFERENCES

- [1] A. Gupta, S. Malik, and P. Ashar, "Toward formalizing a validation methodology using simulation coverage," in *Design Automation Conf.*, June 1997, pp. 740–745.
- [2] J. Kumar, N. Strader, J. Freeman, and M. Miller, "Emulation verification of the Motorola 68060," in *Int'l Conf. on Comp. Design*, Oct. 1995, pp. 150–158.
- [3] J. Jan, A. Narayan, M. Fujita, and A. S. Vincentelli, "A survey of techniques for formal verification of combinational circuits," in *Int'l Conf. on Comp. Design*, Oct. 1997, pp. 445–454.
- [4] G. Parthasarathy, M. K. Iyer, K. T. Cheng, and L. C. Wang, "Safety property verification using sequential SAT and bounded model checking," *IEEE Design & Test of Comp.*, vol. 21, no. 2, pp. 132–143, March 2004.

- [5] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *IEEE Micro*, vol. 27, no. 1, pp. 12–25, Feb. 2007.
- [6] J. Jaeger. (2007, Dec.) Virtually every ASIC ends up an FPGA. EETimes. [Online]. Available: <http://www.eetimes.com/showArticle.jhtml;jsessionid=JRHNSOJ1CLD2SQSNDLPSKH0CJUNN2JVN?articleID=204702700>
- [7] P. M. Rosinger, B. M. Al-Hashimi, and N. Nicolici, "Scan architecture with mutually exclusive scan segment activation for shift- and capture-power reduction," *IEEE Trans. on CAD*, vol. 23, no. 7, pp. 1142–1153, July 2004.
- [8] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Proc. of Design, Automation and Test in Europe*, 2008, pp. 1298 – 1303.
- [9] —, "Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug," *IEEE Trans. on CAD*, vol. 28, no. 2, pp. 285 – 297, Feb. 2009.
- [10] J.-S. Yang and N. A. Touba, "Automated selection of signals to observe for efficient silicon debug," in *VLSI Test Symp.*, May 2009, pp. 79 – 84.
- [11] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," in *Proc. of Design, Automation and Test in Europe*, 2009, pp. 1338 – 1343.
- [12] O. Caty, P. Dahlgren, and I. Bayraktaroglu, "Microprocessor silicon debug based on failure propagation tracing," in *Proc. of Int'l Test Conf.*, Oct. 2005, pp. 284–293.
- [13] C. C. Yen, T. Lin, H. Lin, K. Yang, T. Liu, and Y. C. Hsu, "Diagnosing silicon failures based on functional test patterns," in *Int'l Workshop on Microprocessor Test and Verification*, Dec. 2006, pp. 94–97.
- [14] S. Venkataraman and W. K. Fuchs, "A deductive technique for diagnosis for bridging faults," in *Proc. of Int'l Conf. on CAD*, Nov. 1997, pp. 562–567.
- [15] F. M. D. Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, "BackSpace: Formal analysis for post-silicon debug," in *Int'l Conf. on Formal Methods in CAD*, 2008, pp. 1–10.
- [16] M. W. Moskewicz, C. F. Madigan, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conf.*, 2001, pp. 530–535.
- [17] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a new search algorithm for satisfiability," *IEEE Trans. on Comp.*, vol. 48, no. 5, pp. 506–521, May 1999.
- [18] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518. [Online]. Available: <http://dblp.uni-trier.de/db/conf/sat/sat2003.html#EenS03>
- [19] Z. Fu and S. Malik, "On solving the partial max-sat problem," in *SAT*, 2006, pp. 252–265.
- [20] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.
- [21] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 11, no. 1, pp. 4–15, Jan. 1992.
- [22] V. Boppana and M. Fujita, "Modeling the unknown! towards model-independent fault and error diagnosis," in *Proc. of Int'l Test Conf.*, Oct. 1998, pp. 1094–1101.
- [23] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Great Lakes VLSI Symp.*, aug 2008, pp. 77–82.
- [24] Y. Chen, S. Safarpour, A. Veneris, and J. M. Silva, "Spatial and temporal design debug using partial MaxSAT," in *Great Lakes VLSI Symp.*, aug 2009, pp. 345 – 350.
- [25] M. F. Ali, S. Safarpour, A. Veneris, M. S. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Proc. of Int'l Conf. on CAD*, Nov. 2005, pp. 871–876.
- [26] L. Huisman, "Diagnosing arbitrary defects in logic designs using single location at a time (SLAT)," *IEEE Trans. on CAD*, vol. 23, no. 1, pp. 91–101, Jan. 2004.
- [27] A. Veneris and M. S. Abadir, "Design rewiring using ATPG," *IEEE Trans. on CAD*, vol. 21, no. 12, pp. 1469–1479, Dec. 2002.
- [28] R. Blanton, K. N. Dwarakanath, and R. Desineni, "Defect modeling using fault tuples," *IEEE Trans. on CAD*, vol. 25, no. 11, pp. 2450–2464, Nov. 2006.
- [29] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Proc. of Design, Automation and Test in Europe*, April 2007, pp. 16–20.
- [30] N. A. Touba, "Survey of test vector compression techniques," *IEEE Design & Test of Comp.*, vol. 23, pp. 294–303, July 2006.
- [31] S. Safarpour, A. Veneris, and H. Mangassarian, "Trace compaction using sat-based reachability analysis," in *Proc. of ASP Design Automation Conf.*, 2007, pp. 932–937.

PLACE
PHOTO
HERE

Yu-Shen Yang received the B.A.Sc, M.A.Sc and Ph.D degrees in computer engineering from University of Toronto in 2002, 2004, and 2010, respectively. He is currently a senior software engineer at Vennsa Technologies, Toronto, Ontario, Canada, where he is in charge of research and development. His research interests include logic design debugging and correction, logic resynthesis and silicon debug.

PLACE
PHOTO
HERE

Andreas Veneris (S'96-M'99-SM'05) received a Diploma in Computer Engineering and Informatics from the University of Patras in 1991, an M.S. degree in Computer Science from the University of Southern California, Los Angeles in 1992 and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1998. In 1998 he was a visiting faculty at the University of Illinois until 1999 when he joined the Department of Electrical and Computer Engineering and the Department of Computer Science at the University

of Toronto where today he is an Associate Professor. His research interests include CAD for debugging, verification, synthesis and test of digital circuits/systems, and combinatorics. He has received several teaching awards and a best paper award. He is the author of one book and he holds three patents.

He is a member of ACM, IEEE, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society.

PLACE
PHOTO
HERE

Nicola Nicolici (S'99-M'00) is an Associate Professor in the Department of Electrical and Computer Engineering at McMaster University, Canada. He received the Dipl. Ing. degree in Computer Engineering from the University of Timisoara, Romania (1997), and a Ph.D. in Electronics and Computer Science from the University of Southampton, U.K. (2000). His research interests are in the area of computer-aided design and test. He has authored a number of papers in this area and received the IEEE TTTC Beausang Award for the Best Student Paper

at the International Test Conference (ITC 2000) and the Best Paper Award at the IEEE/ACM Design Automation and Test in Europe Conference (DATE 2004).