Automated Silicon Debug Data Analysis Techniques for a Hardware Data Acquisition Environment

Yu-Shen Yang[1], Brian Keng[1], Nicola Nicolici[2], Andreas Veneris[1,3,4], Sean Safarpour[5]
[1]Dept. of ECE, University of Toronto, Toronto, Canada.
[2]Dept. of ECE, McMaster University, Hamilton, Canada.
[3]Dept. of CS, Univeristy of Toronto, Toronto, Canada.
[4]Dept. of CS, Athens University of Economics and Business, Athens, Greece.
[5]Vennsa Technologies Inc., Toronto, Canada.
E-mail: [1]{yangy, briank, veneris}@eecg.utoronto.ca. [2]nicola@ece.mcmaster.ca, [5]sean@vennsa.com

***Abstract*—Silicon debug poses a unique challenge to the engineer because of the limited access to internal signals of the chip. Embedded hardware such as trace buffers helps overcome this challenge by acquiring data in real time. However, trace buffers only provide access to a limited subset of pre-selected signals. In order to effectively debug, it is essential to configure the trace-buffer to trace the relevant signals selected from the pre-defined set. This can be a labor-intensive and time-consuming process. This paper introduces a set of techniques to automate the configuring process for trace buffer-based hardware. First, the proposed approach utilizes UNSAT cores to identify signals that can provide valuable information for localizing the error. Next, it finds alternatives for signals not part of the traceable set so that it can imply the corresponding values. Integrating the proposed techniques with a debugging methodology, experiments show that the methodology can reduce 30% of potential suspects with as low as 8% of registers traced, demonstrating the effectiveness of the proposed procedures.**

***Index Terms*—Silicon debug, post-silicon diagnosis, data acquisition setup**

I. Introduction

Modern integrated circuit development cycles require several different synthesis and verification stages before a silicon prototype is manufactured. Each verification stage ensures that its corresponding synthesis step did not introduce any errors (e.g. timing, functional, power). At the Register Transfer Level (RTL), formal methods [1], [2] and simulation [3] approaches are used to verify that the RTL model complies with its functional specification. However with the growing size of modern designs along with the prevalent use of intellectual property (IP), it is infeasible to achieve 100% functional verification coverage. In addition, time-to-market constraints only allow a finite amount of engineering resources to be dedicated towards functional verification. This limits the verification engineer's ability to ensure functional correctness. As a result, functional bugs are introduced into silicon prototypes. In fact, more than 60% of design tape-outs require a re-spin with more than half of these containing a bug due to logic or functional errors [4]. Each re-spin dramatically increases the cost of a project and eats away the time-to-market.

The main challenge of silicon debug is the limited observability of the internal signals in the chip. Unlike pre-silicon verification, only nets that are connected directly to output pins can be observed which is generally insufficient for debugging. To overcome this issue, two types of *Design-for-Debug* (DfD) techniques, *scan-based* and *trace buffer-based* techniques, have been introduced. Scan chains provide the value of all scanned states for one cycle, while trace buffers allow the engineer to trace a small number of states in consecutive cycles. Although these two techniques greatly enhance the observability of the chip, the proportion of nets that can be observed is still small compared to pre-silicon verification.

Once a silicon chip fails during test, an iterative post-silicon validation process is launched to locate the root-cause as shown in Figure 1. Engineers start with setting up the environment to capture appropriate data from the chip under test while it is run in real-time. Then, this sparse amount of acquired data is analyzed by the engineer to prune the error candidates and also setup the next debug session. This time-consuming and labor-intensive cycle continues until the root cause of the failure is determined.

Several techniques have been proposed to automate the data analysis process [5]–[7]. They analyze the acquired data to determine the root cause of the error. Clearly, the quality of those analysis is affected by the acquired data and it can be more effective if the data contains useful information regarding the error location. Hence, one question software solutions to silicon debug need to address is which set of signals is important for tracing. This includes which signals and cycles during the execution run to trace. Note that acquiring the data at run time can be time-consuming to setup and the data is read out at slower speed. Therefore, it is desired to minimize the iterations of the data acquisition and have a concise set of signals for tracing. Furthermore, since only a small proportion of the entire design can be accessed by DfD hardware, it is important for any software solutions to be able to handle this constraint.

This work proposes two novel approaches and a ranking system to enhance the data analysis while considering the data acquisition hardware constraints. It first utilizes UNSAT cores to identify registers that may contain useful information to help the diagnosis process. Since the UNSAT core relates directly to the error, it can provide greater precision. Secondly, a searching procedure is proposed to find alternatives for untraceable states. The algorithm takes the hardware constraints into account and finds alternative states among the pre-selected registers such that values of those registers can

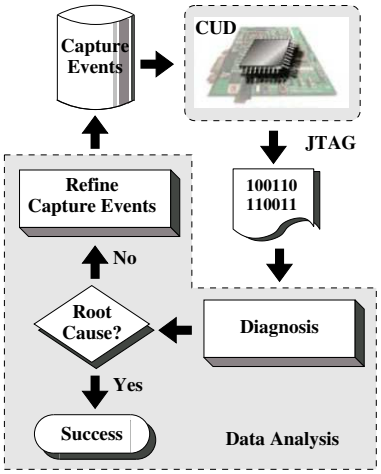


Fig. 1. A typical silicon debug flow

imply the untraceable ones. The proposed searching algorithm is memory efficient because only a small window of the complete silicon trace is analyzed. Finally, since in practice only one register group can be traced at a time, a simple ranking system is suggested to prioritize the traceable register groups according to the result from the proposed analysis.

Experimental results show that when the available hardware is complemented with the new techniques, the data analysis methodology can reduce, on average, 30% of the number of suspects that the engineer needs to investigate. This reduction is achieved with only 8% to 20% of registers traced.

The remaining paper is organized as follows. Section II summarizes prior work on hardware and software solutions for silicon debug. Section III and Section IV illustrate the new approaches for selecting signals to be traced in runtime. Then, a simple ranking system to determine the pre-selected group for tracing is presented in Section V. Experimental results are presented in Section VI, followed by the conclusion in Section VII.

II. Background

Silicon debug involves hardware and software components. The hardware components consist of DfD hardware that improves signal observability. The software components include the automated debugging tools and the test environment setup to collect and analyze the data from the tester. In the following subsections, we briefly introduce notation and review some of this background material.

A. Notation

In this work, we consider a sequential circuit with primary input set $\mathcal{X} = (x_1, \cdots, x_m)$, register set $\mathcal{S} = (s_1, \cdots, s_p)$, and primary output set $O = (o_1, \cdots, o_n)$. Sequential circuits are modelled in *Iterative Logic Array* (ILA) representation. The design is unfolded over time to maintain the state transition information. Throughout this paper, the superscript of a symbol refers to the cycle of the unfolded circuit. For example, $\mathcal{X}^2$ represents the set of the primary input in the second cycle. x_1^2 refers to the primary input x_1 in the second cycle. We also let symbol T_i denote the i-th simulated cycle.

B. Design for Debug Hardware Solutions

To increase observability of internal signals in the silicon prototype, there are two main DfD techniques adopted in practice: scan chains and trace buffers.

Scan chains allow engineers to capture and off-load the value of scanned registers at a specific cycle. However, scan-out operation interrupts the execution of the chip because the values stored in the registers are destroyed. In order to resume the execution from the same point, the environment must be reset and restarted from the beginning of the test vector [8].

Trace buffers [9], [10] are another DfD technique that uses an on-chip memory to record internal signals. As shown in Figure 2, a trace buffer contains control logic, called trigger logic (e.g., embedded hardware assertions), employed for on-line monitoring of circuit behavior. Once the trigger condition is asserted the on-chip memory can start/stop recording the logic values of the selected signals into buffers, which typically range from 8K to 256K. Subsequently, the recorded data can be read via a low-bandwidth interface, such as boundary scan. Due to the limited size of this on-chip memory, only a 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 selected and 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 [11]–[13]. In those works, authors try to select a small set of signals such that their values can restore a significant amount of untraceable states. For example, *State signal selection* proposed in [11] conducts structural analysis on the circuit. Then, it calculates the restorability of signals to determine the signals to be traced.

C. Automated Data Analysis

Although DfD hardware enhancement increases the observability of internal signals, there is a lack of techniques to automate the data analysis process on the acquired data, e.g. the *data analysis* step in Figure 1. Recently, there has been an effort to develop methodologies to aid the engineer in other parts of the silicon debug process besides data acquisition.

Caty et. al. [5] and Yen et. al. [14] both perform silicon debug by analyzing the data collected from scan chains. The former uses the data to determine the failing registers at each timeframe and then conducts back-tracing and forward-tracing from those registers to narrow down the potential root cause candidates. The latter proposes a similar approach. They use the scan chain data to identify the portion of the trace where the error is sensitized. Then, the suspect list is pruned using both the path-tracing method [15] and simulating the faulty value of the suspect in the golden model.

Yang et. al. [7] propose a software solution to silicon debug that utilizes trace buffers. It analyzes the acquired data to discovering the root-cause of chip failure in both spatial and temporal domains. At the end of each debug session, if the root-cause is inconclusive, the methodology provides suggestions on the data acquisition setup of the sub-sequent debug session. However, it assumes that all registers can be traced by the trace buffer which is impractical. In practice, the number

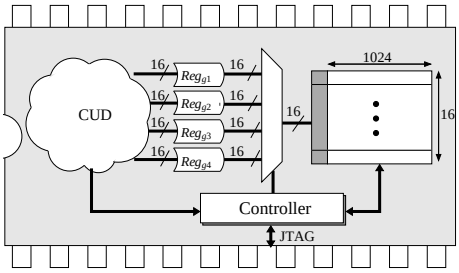
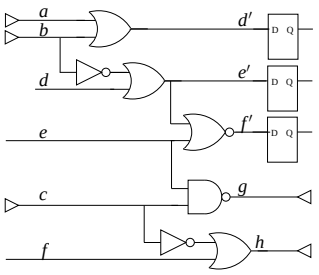


Fig. 2. Trace buffer configuration



(a) Erroneous Circuit

Vector $\{abc\}$ $(\mathcal{V})$	Response $\{g, h\}$	
	Correct $(\mathcal{O}_{corr})$	Erroneous $(\mathcal{O}_{err})$
100	11	11
011	01	01
110	11	11
111	11	00

(b) Test vector sequence and response. The initial value of {d, e, f} is 000

Fig. 3. Example erroneous circuit. The correct implementation of gate d′=OR(a,b) is d′=AND(a,b)

of registers that are traceable are limited. Consequently, this constraint can greatly affect the performance of [7].

Formal methods are used in [6] to restore the state information when a chip fails the test. It starts from the crash state and computes the states backward in time. Signatures are captured during the chip execution and used to determine a unique or a small set of possible predecessor states that leads to the crash state. It requires additional hardware structures to compute signatures before they are stored in the trace buffer.

D. UNSAT Cores

This work utilizes the use of UNSAT cores to find suggestions for the data acquisition setup. A brief overview of UNSAT cores is give in this section.

Given a set of Boolean variables, a *literal* is an instance of the variable or its negation. An SAT instance in *Conjunctive Normal Form* (CNF) is a conjunction of *clauses* where each clause is a disjunction of literals. An SAT instance is *satisfied* if there exists an assignment over variables such that all clauses are evaluated to be true. That is, at least one literal in each clause is evaluated to be 1.

If such an assignment cannot be found, the SAT instance is *unsatisfied*. In this case, any subset of clauses in the instance that is also unsatisfiable is referred to as an *UNSAT core*. Modern SAT solvers [16]–[18] can produce UNSAT cores as a result of finding an instance to be unsatisfiable. The following is an example of an unsatisfied CNF formula Φ and its UNSAT core.

$$\Phi = (a+b)\cdot(a+c)\cdot(\overline{b}+\overline{c})\cdot(\overline{a})\cdot(\overline{c})$$
$$\textit{UNSAT core} = \{(a+c),(\overline{a}),(\overline{c})\}$$

An unsatisfied SAT instance can have multiple UNSAT cores. Each represents a situation where the CNF formula is unsatisfied. Additional UNSAT cores can be obtained by performing relaxation on the current UNSAT core clauses [19]. In summary, each relaxible clause in the UNSAT core is augmented with a distinct relaxation variable. Additional clauses are added to the CNF to ensure assignments to relaxation variables complied with one-hot property, i.e. one and only one relaxation variable is assigned to 1.

III. Data Acquisition Setup

Due to the insufficient observability of internal signals, selecting which set of signals to observe is a key step in the silicon debug process. Trace buffer-based DfD hardware provides the engineer great flexibility in the choice of traced signals. However, they can only trace a limited subset of signals. To make the most efficient use of this hardware, the engineer uses two major criteria for selecting signals to acquire using trace buffers:

1) Signals that are related to the error source or provide valuable information to aid in pruning suspects.
2) Signal selection needs to comply with the hardware constraints. As discussed in Section II-B, in most real-word designs, only a small set of hard-wired signals can be traced during the execution.

In the next subsection, an algorithm that utilizes the proof trace generated by SAT solvers to identify registers that may contain useful information to aid debugging is presented.

A. Registers Identification with UNSAT Cores

As discussed in Section II-D, an UNSAT core of an unsatisfiable SAT problem is a subset of clauses that is also unsatisfiable. Assume that there is a golden model, such as a high-level behavioral model, available during the debug to provide correct responses of the design. Note, although in this behavioral model we do not have access to the data on every single net in the implementation, the important information on the data/address buses, as well as the essential control signals that steer the data through the data-path, can be monitored. Then, given an erroneous circuit $\mathcal{C}$, the input trace $\mathcal{X}$ and the correct output response O, the CNF formula $\mathcal{C}\cdot\mathcal{X}\cdot O$ 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 have to be one of the followings:

- Nodes that excite the error
- Nodes along the error propagation paths
- Side inputs to the error propagation paths

Clearly, those signals can be potential locations for tracing and provide information about the erroneous behavior of the design.

Example 1 *Consider the circuit shown in Figure 3(a). Assume the error is at* d′*, which should be* d′=AND(a,b)*. The test vector and the correct/erroneous response are shown in Figure 3(b). Since the circuit is erroneous, the CNF formula,* $\Phi = \mathcal{C}\cdot\mathcal{X}\cdot O_{corr}$*, is unsatisfiable. An UNSAT core of the problem is*

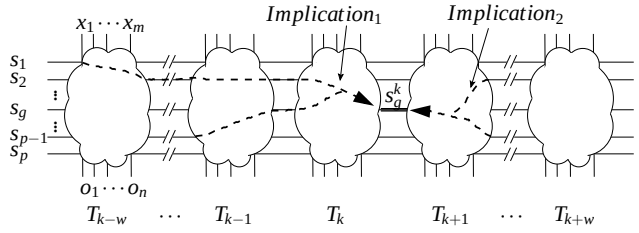


Fig. 4. An unfolded circuit from T_{k-w} to T_{k+w}

$$\{(\overline{b^2}+d'^2)\cdot(\overline{d'^2}+d^3)\cdot(\overline{d^3}+e'^3)\cdot$$
$$(\overline{c^4}+\overline{e^4}+\overline{g^4})\cdot(\overline{e'^3}+e^4)\cdot(b^2)\cdot(c^4)\cdot(g^4)\} \quad (1)$$

Therefore, signals that should be traced are d *at frame 3 (from the clause* $(\overline{d'^2}+\overline{d^3})$ *) and* e *at frame 4 (from the clause* $(\overline{e'^3}+e^4)$ *).*

Algorithm 1 shows the overall algorithm. It starts with obtaining the initial UNSAT core ($\mathcal{U}_{init}$ in line 6). Then, the algorithm tries to obtain more UNSAT cores by relaxation as summarized in Section II-D. First, it relaxes unit clauses in $\mathcal{U}_{init}$ that represent input vectors (line 9) until the problem is SAT. Next, it repeats for unit clauses in $\mathcal{U}_{init}$ that represent output responses (line 14). Since each UNSAT core can represent different error propagation paths, different signals can be included. To ensure all paths are considered, the union of all UNSAT cores is taken as shown in line 11 and line 15 in the algorithm. Finally, registers associated with variables in the UNSAT cores are the potential locations for tracing.

Example 2 *Continue from Example 1, another UNSAT core can be obtained by relaxing* o_1^4*. Let* r_1 *be the relaxation variable, the relaxed clause of* (g^4) *is* $(g^4 + r_1)$ *and an additional clause* (r_1) *is added to the original* Φ*. The new UNSAT core is*

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

The new list of register-to-be-traced contains d *at frame 2, 3,* e *at frame 3, 4 and* f *at frame 4.*

IV. Alternative Signal Searching

The algorithm IdentifyTracedSignals from the previous section selects a list of registers that may contain useful information about the behavior of the faulty chip. However, as mentioned in Section II-B 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.

Consider a circuit modelled in the ILA representation shown in Figure 4. The goal is to find registers that can imply the non-traceable register, or *target register*, denoted s_g^k (s_g at timeframe T_k). Since s_g^k cannot be traced directly, we want to restore its value with traceable registers within a certain window of timeframes $\{T_{k-w}\cdots T_{k+w}\}$, where w is the *window_size*. Those traceable registers are referred to as *candidate registers*. As shown in Figure 4, the value of s_g^k can be restored in three ways: (1) forward implication, (2) backward justification or (3) combination of (1) and (2).

Algorithm 1 Identifying registers for tracing using UNSAT cores

```
1:  C := The erroneous design
2:  X := Input vectors
3:  O := Output vectors
4:  Φ := C · X · O
5:  procedure IDENTIFYTRACEDSIGNALS(Φ)
6:      U_init := Solve Φ and extract the UNSAT core
7:      U ← U_init
8:      while Φ is unsatisfiable do
9:          relax on clauses {c|c ∈ U_init and c is an input
                vector unit clause}
10:         U_new ← solve Φ and extract the UNSAT core
11:         U ← U ∪ U_new
12:     end while
13:     while Φ is unsatisfiable do
14:         relax on clauses {c|c ∈ U_init and c is an output
                response unit clause}
15:         U_new ← solve Φ and extract the UNSAT core
16:         U ← U ∪ U_new
17:     end while
18:     R ← extract registers in U
19:     return R
20: end procedure
```

To solve this problem, we formulate a SAT instance that will search for assigned values to candidate registers that, together with the input/output trace, imply to the target register. The alternative for the target register consists of those selected candidate registers. The detail of the formulation is given in the following subsections.

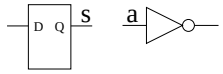
A. Problem Formulation

The basic problem formulation is presented in this section. In order to indicate whether a candidate register is selected for generating an implication, new variables are added for every candidate register. We use the notation $\mathcal{L}=\{l_1,l_2,\cdots\}$ to label those variables. The formula contains two components as expressed as:

$$\Phi=\prod_{j=k-w}^{k+w}\Phi_c^j(\mathcal{L}^j,\mathcal{X}^j,O_{obv}^j,\mathcal{S}_{known}^j)\cdot E_N(\bigcup_{j=k-w}^{k+w}\mathcal{L}^j) \quad (3)$$

The first component models the design from timeframe T_{k-w} to T_{k+w}. Intuitively, each Φ_c^j represents a copy of the erroneous design at timeframe j with the vector $\mathcal{X}^j$ and O_{obv}^j enforced. Previous traced register values ($\mathcal{S}_{known}^j$) 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.

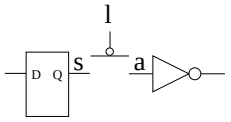
The value w is user-defined, but also depends on the size of the trace buffer. We can set w such that $2w+1 = buffer\ depth$



1: $cond_1 := (s \neq X)$ OR $(a \neq X)$
2: $cond_2 := (s = a)$ OR $(s = X)$ OR $(a = X)$
3: $cond_1$ AND $cond_2$ is TRUE

(a) Schematic (b) Model Syntax

Fig. 5. The model of target registers



1: **if** $\bar{l}$ **then**
2: $(s = a)$ is TRUE
3: **else**
4: $cond_1 := (s \neq X)$ OR $(a \neq X)$
5: $cond_2 := (s = a)$ OR $(s = X)$ OR $(a = X)$
6: $cond_1$ AND $cond_2$ is TRUE
7: **end if**

(a) Schematic (b) Model Syntax

Fig. 6. The model of candidate register

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 $E_N(\bigcup_{j=k-w}^{k+w} \mathcal{L}^j)$ constrains the number of selected candidate registers. The detail of the construction can be found in [20]. To find the minimum number of candidate registers required for implications, it starts the constraint from one active select variable and increments the value up to the total number of the select variables, until a solution is found.

At the end, each solution of the problem is one implication for the target register. Candidate registers of which the select variable (l) is active are the necessary registers to generate the implication. Note that the algorithm identifies not only the registers, but also the timeframe where those registers are at to generate the implication.

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

B. Register Modelling

Target register and *candidate registers* need to be encoded specially in the CNF formula in order to solve the problem. 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 non-unknown value. The implication can come from two directions: forward propagation from assignments in the earlier timeframe, or the backward justification from assignments in a later timeframe. Hence, the target register is modelled as shown in Figure 5. A extra signal is introduced to disconnect s_g^k from its fanin. The syntax of the model is shown in Figure 5(b). $cond_1$ (line 1) ensures that a non-unknown implication is generated by either forward implication or backward justification. $cond_2$ gives the flexibility that the implication only needs to be satisfied from one direction. Furthermore, if there are implications from both directions, the implied value have to be the same.

Candidate Register: Candidate registers are traceable registers that the SAT solver can assign 1/0 when they are selected. For each candidate register, two variables, are introduced as shown in Figure 6(a). The variable l, referred to as *select variable*, determines whether the register connects to its fanout. When l equals 0, the network remains the same (line 1-2 in Figure 6(b)). When l equals 1, the register is disconnected from its fanout and allow the SAT-solver to assign 0/1 to the either end of the break. This enables the possibility to identify forward/backward implications. Similar to the model for target registers, at least one of the two variables at the disconnected ends must have a non-unknown value. If both ends have non-unknown values, the values must be the same.

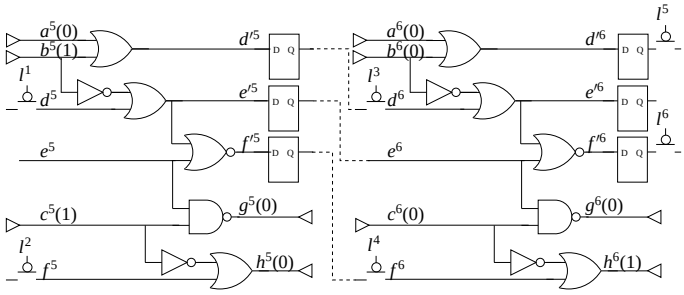


Fig. 7. ILA of the example circuit in Figure 3(a)

Example 3 *Figure 7 shows a portion of the ILA of the example circuit in Figure 3(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 6 with six additional select variables, $\{l^1 \cdots l^6\}$. One can verify that the value of e^6 can be restored if the value of d^5 is known.*

C. Formulation Improvements

As shown in Figure 2, typically, traceable registers are divided into groups. When configuring the trace buffer, one group of the traceable registers is selected and traced for several timeframes. With this observation, we can reduce the number of select variables for the candidate registers. 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 timeframes can share one select variable as well. In Example 3, assuming d and f are in different groups, the number of l's can reduce to two, e.g. d^5, d^6, d^7 share one l, 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-A, 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 register 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.

TABLE I
TRACEABLE REGISTER GROUP INFORMATION

Circ.	Total reg.	# of groups	# of reg./group	Perc.
spi	160	8	8	40%
hpdmc	453	16	8	28%
usb	2054	32	16	25%
s1423	74	6	6	49%
s5378	179	7	8	31%
s9234	211	8	8	30%

V. 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, we describe a simple ranking system to prioritize the traceable register groups according to the result from the proposed algorithms.

- Rule 1: The group that contains the most registers returned by the algorithm IDENTIFYTRACEDSIGNALS has higher priority. This is because those registers are directly related to the error source. Their values may contain most useful and direct 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 others, 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 timeframes. This is simply to efficiently utilize the memory space of the trace buffer.

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 a Core 2 Duo 2.4GHz process with 4GB Memory.

To emulate the real trace buffer hardware structure, a subset of registers of each design is selected randomly, or by *State signal selection* [11], as traceable by the trace buffer and they are divided into groups. The grouping configuration is summarized in Table I. The first column lists the benchmark used in the experiments. The second column of the table shows the total number of registers in each design. Columns three and four have the number of the register groups and the number of registers in each group, respectively. Column five shows the percentage of total registers that can be traced.

In addition, as mentioned in Section IV-C, several target registers can be combined into one search execution to reduce the runtime. In our experimental setup, the target registers in every four timeframes are targeted together. For designs from OpenCores.org, test vectors are extracted from the testbench provided by OpenCores.org. Test vectors for ISCAS'89 are generated randomly. In both cases, the trace length is between 100 to 300 timeframes. In the experiment, we set *window_size* (*w*) to be six timeframes.

Table II summaries the performance of debug analysis under two situations. Columns 2 – 4 show the performance of the analysis with no state information available, while columns 5 – 11 show the performance of the analysis with the proposed techniques. Each row is one individual case that contains a different bug in the design. A single random functional error (wrong assignment, incorrect case statement, etc) is inserted into the RTL code. The final row is the geometric mean of the data in columns. In the experiment, the analysis performs the model-free diagnosis for one hierarchical level in each debug session. The total number of modules returned at the end of the debug sessions is shown in columns two and five. This is the sum of the number of modules that the engineer needs to investigate after each debug session. As shown in the table, with state information the debugging tool can effectively eliminate more false candidates in all cases. The percentage reduction on the number of suspects, e.g. comparing column five and column two, is listed in column six. The reduction can be as high as 98% and an average of 31% reduction is achieved.

Columns three and seven show the number of debug sessions performed. About one third of cases require less debug sessions to find the root cause of the error, for example, the second case of `spi`, `hpdmc` and both cases of `ubs`. The number of registers traced by the trace buffer is shown in column eight. Those numbers are small compared to the total number of registers shown in Table I. The benefit of the proposed technique is shown when one considers the reductions in both the number of suspects and the number of debug sessions.

Finally, the runtime of the debugging is summarized in column four and columns 9–11. Because of the reduction of suspects and debug sessions, the runtime for diagnosis is also reduced in the case of the proposed methodology. The runtime is 52% less on average (from 1426s down to 684s). The runtime on searching the registers for tracing is recorded in column 10. This is the additional computation required by the proposed methodology. As shown in the table, it can be significant in cases, such as `hpdmc`. This is because the algorithm has a higher failing rate on finding the recommendation for the non-traceable registers in those cases. The detail on the performance of the searching algorithm will be discussed later. Overall, the total runtime of the propose method is about 1.43 times longer than the runtime when no state information is used. However, since the number of the final suspects is reduced significantly, this additional runtime may be acceptable if the time saved from the manual inspection of less suspects is greater.

Next, we discuss the performance of the alternative searching algorithm. Clearly, the performance of the algorithm depends on the available traceable signals. Some signals may not be able to restore at all if the necessary registers are not traced. Hence, in addition to selecting the traceable register randomly, another approach, *State signal selection*, is also used. *State signal selection* selects registers that their values have a higher potential to restore other unknown registers. However, due to the technical implementation, *State signal selection* only handles ISCAS benchmarks. The results are summarized in Table III. The second and fourth columns of Table III show the percentage of targets that the search algorithm successfully finds alternative recommendation. The number of traceable register groups selected in order to generate application is shown in columns three and five. In the case of the random

TABLE II
PERFORMANCE OF DEBUGGING WITH PROPOSED TECHNIQUES

Circ.	With no state information			With proposed methods						
	# 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	98%	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

TABLE III
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
s5387	100%	6.3	100%	7
	100%	6.3	100%	7
s9234	50%	1	50%	1
average	49%	4.7		

selection, the algorithm is able to find an alternative for almost half of the targets on average. The performance of using *State signal selection* is similar to the random selection. This can be because that the main goal of *State signal selection* is to restore as many registers as possible over the whole design. It does not target a specific region of the design.

In the next set of experiments, we investigate the performance of debugging when various state information is available. The experimental results are summarized in Table IV. All numbers are the average of 11 buggy benchmarks discussed in Table II. The reference case for comparison is the case where no state information is used (columns 2 – 4 of Table II). The first column lists the four considered cases. The next two columns summarize the number of suspects reduced and the ratio of the number of sessions comparing to the reference case. The fourth column is the ratio of traced registers to the total number of registers, followed by the reduction on the diagnosis runtime.

To demonstrate the advantage of the proposed UNSAT core approach, we compare it with the X-simulation described in [7] as shown in the first two cases of the table. In these two cases, no hardware constraints are considered, i.e. assuming all registers can be traced. From the table, we can see 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 less state information*. In 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 carried out to find alternatives for non-traceable registers. Those registers are simply ignored. Comparing the result of the case 3 and the case 4, we can see that with the help of the searching algorithm, the debugging has better performance. For example, the reduction of suspects increases from 27% to 31%. This implies the effectiveness of the searching algorithm.

In the last set of experiments, two variations of the experiment setup are implemented to further investigate the performance of the searching algorithm, namely, the search window size (w) and the hardware group structure.

First, the algorithm is executed with four different *window_size* (*w*). The performance is summarized in Table V. The first column lists the four *window_size* considered. The second column shows the average number of targets of all testcases. One can observe that as the window size becomes larger, the number of targets is decreased. This is because that more input/output values are applied when a larger window is used, which provides extra information to imply values to some targets that are unknown under the smaller window of the trace. The third and fourth columns show the average number of targets that are successfully found an alternative and the success rate, respectively. In general, a higher success rate is achieved as the window size increases. This is expected since there are more candidates for selection and a longer trace is used, which can restore values of more signals.

Next, three trace buffer group structures are tried to see the performance of the searching algorithm. `Config 1` is the configuration in Table I. `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 32 groups of the size of eight registers; `Config 2` has 32 groups of the size of four registers, while `Config 3` has 32 groups of the size of two registers.

The success rate on finding an alternative is plotted in Figure 8(a). As expected, since there are less traceable registers, more non-traceable registers cannot be replaced. Hence, the success rate drops as the number of candidate becoming less. Figure 8(b) depicts the average number of selected traceable groups for generating implications. In general, more traceable groups are required when each group contains less

TABLE IV
IMPACT OF STATE INFORMATION ON THE DIAGNOSIS

Cases	% of susp. reduc.	% sess.	% of traced sig.	% of diag runtime reduc.
X-sim with no constraint	82%	79%	56%	77%
UNSAT with no constraint	85%	74%	16%	89%
UNSAT with no search	27%	90%	8%	15%
UNSAT with search	31%	89%	10%	26%

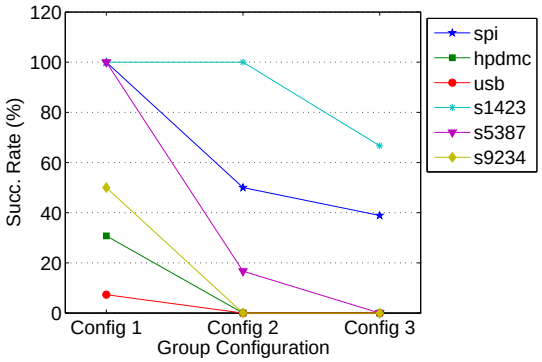
TABLE V
IMPACT OF WINDOW SIZES ON THE SEARCHING ALGORITHM

Window	# of targets	# of succ.	Succ. rate
w=2	5	2	40%
w=4	4.8	2.3	48%
w=6	4.1	2	49%
w=8	3.6	1.7	47%

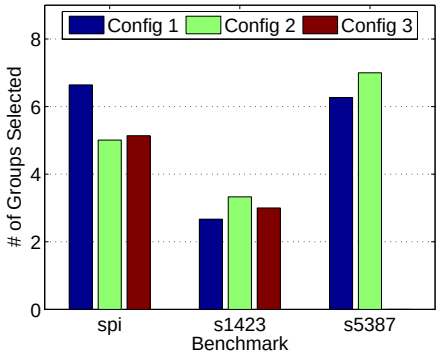
registers, such as `Config 1` and `Config 2` of `s1453`. However, there are cases where less groups are required when smaller groups are used, such as `Config 3` of `s1453`. In those cases, the algorithm is not able to find alternatives for some non-traceable registers that have the alternative previously. Hence, the average calculated is done with less target registers.

VII. CONCLUSION

Internal state information is important for silicon debug because they can aid in pruning the suspect modules. However, due to the hardware constraints, only a small proportion of the registers can be traced during the execution. This paper presents novel techniques to identify a more precise set of registers as the suggested candidate for tracing. It considers the hardware constraints and presents an approach to find alternative recommendation for non-traceable registers such that their value can be obtained through implications of other traceable registers. The experimental results show that the proposed techniques can help silicon debug diagnosis methodologies to achieve good performance under the data acquisition hardware limitation.



(a) Successful rate



(b) Group selected

Fig. 8. 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

REFERENCES

[1] 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.

[2] 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.

[3] A. Gupta, S. Malik, and P. Ashar, "Toward formalizing a validation methodology using simulation coverage," in *Design Automation Conf.*, June 1997, pp. 740–745.

[4] J. Jaeger, "Virtually every ASIC ends up an FPGA," EETimes.com, Dec. 7 2007.

[5] 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. 1–10.

[6] 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.

[7] Y. S. Yang, N. Nicolici, and A. Veneris, "Automated data analysis solutions to silicon debug," in *Proc. of Design, Automation and Test in Europe*, April 2009, pp. 982–987.

[8] 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.

[9] M. Abramovici and Y.C.Hsu, "A new approach to silicon debug," in *IEEE International Silicon Debug and Diagnosis Workshop*, Nov. 2005.

[10] E. Anis and N. Nicolici, "Low cost debug architecture using lossy compression for silicon debug," in *Proc. of Design, Automation and Test in Europe*, April 2007, pp. 225–230.

[11] 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.

[12] H. Ko and N. Nicolici, "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.

[13] 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.

[14] 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.

[15] 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.

[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. Een and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003.

[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.