Accelerating Post Silicon Debug of Deep Electrical Faults

Bao Le, Dipanjan Sengupta, Andreas Veneris, Zissis Poulos
University of Toronto, ECE Department, Toronto, ON M5S 3G4
({lebao, dipanjan, veneris, zpoulos}@eecg.toronto.edu)

***Abstract*—With the growing complexity of current designs and shrinking time-to-market, traditional ATPG methods fail to detect all electrical faults in the design. Debug teams have to spend considerable amount of time and effort to identify these faults during post silicon debug. This work proposes off-chip analysis to speed-up the effort of identifying hard-to-find electrical faults that are not detected using conventional test methods, but cause the chip to crash during functional testing or silicon-bring-up. With the goal of reducing the search space for reconstructing the failure trace path formal methodology is used to analyze the reachable states along the path. Isolating the root cause of failure is also accelerated. Moreover, we propose a forward traversal technique on a selected few possible faults to generate a complete failure trace starting from the initial state to the crash state. Experimental results show that the proposed approach can significantly reduce the actual silicon run thereby reducing the overall debug time.**

I. Introduction

Advancement of technology allows millions of logics gates to be integrated in a single chip. ATPG algorithms generate test vectors for each fault at the gate-level model of the circuit. Although this methodology has high fault coverage, current designs have large number of faults to be tested. Moreover, the algorithm complexity makes this process time consuming and expensive[]. The shortcomings are even more acute when sequential ATPG algorithms are used to identify the electrical bugs that require multiple cycles to be detected. Thus deep electrical bugs - triggered only at extreme sequential depth - often escape the manufacturing test step but cause the chip to crash after few seconds of execution. Once a failure is observed, additional validation step, post silicon debug, is used to locate the design defects. The process of identifying these bugs is painstakingly slow and takes more than half of the chip development cycle[][][][]. In this paper we propose a novel technique to reduce the debug time for diagnosing these bugs by augmenting off-chip analysis for reconstructing the path from initial state to the crash state.

In [], a post silicon debug technique is proposed to identify such bugs. Starting from the crash state, [] traverses the failing trace backwards towards the initial state. The possible set of faults are pruned during this process. Fig. 1 shows the overall methodology. Although the actual time to run the chip can be in the order of a few seconds, performing reachability analysis for each state requires the verification engineer to run the chip multiple times. This is a very slow process negating the effectiveness of the approach. Building on the previous work on FD BackSpace, this paper reduces the time required to identify a small set of electrical faults that can cause the chip to fail.

In general deep electrical bugs cause the chip to malfunction after executing millions of cycles. Generating test cases for such bugs require months of simulation on server farms. However, actual silicon run is several orders of magnitude faster requiring few seconds to trigger the error in the design thereby leading to the crash state. Unlike during pre-silicon verification, the accessibility and visibility of internal signals are very limited in post-silicon debug and hence this is the major challenge in the validation and debug of first silicon. In [] *trace buffers* is proposed to store the values of selected signals for multiple clock cycles. A binary-search-based debug method [12] iteratively divides the search space in half until the method identifies the first cycle in which the error is activated and observed. All of these techniques address the observability problem and can enhance off-chip analysis presented in this paper.

This paper describes a technique that combines formal methods with on-chip support logic to identify the root cause of the failure. The off-chip analysis extracts information from the crashing state that results in reduction of the number of actual silicon runs. In particular, the contributions are as follows:

- We propose a formal technique to analyze the possible states in the failure trace path. Rather than running the chip to determine the reachability of each possible state, we perform off-chip analysis to determine if a state can exist on the failure trace path. This reduces the number of silicon runs as well as constraints the state space search.
- Assuming deterministic system behavior[], the algo-

rithm quickly reduces the number of possible faults by identifying failure paths that do not match the actual execution of the chip.

- In addition to backward traversal along the failure trace path, as proposed in [], this work proposes forward traversal technique that further reduces the actual silicon runs. This additional analysis also helps in reducing the overall debug time.

Experiments on the ITC'99 benchmarks demonstrate the effectiveness of our approach. On average the number of silicon runs were reduced by X%. This can be translated to shorter debug time. The rest of the paper is organized as follows. In Section II provides the background information and the introduces the notations used in the rest of the paper. Sections III, IV and V presents the three off-chip analysis techniques that reduce the debug time. Experimental results are presented in Section VI followed by conclusions in Section VII.

II. Preliminaries

A. Notation and Basic Definitions

We model the fault-free circuit C as a finite state machine M, with S latches, I inputs, O outputs, initial states $I \subseteq 2^S$, and transition relation $\delta \subseteq 2^S \times 2^I \times 2^S$. An execution path (run) on M is a finite sequence of states $\pi = s_0 s_1 s_2 \dots s_n$, where $n \in \mathbb{N}$. A *crash state* is a state of the chip where a bug is observable (e.g., a system hang). A path $s_i s_{i+1} \dots s_n$ is said to be a valid trace leading to the crash state if s_n is the crash state and for each s_j, $i \leq j \leq n-1$, s_j is a predecessor of s_{j+1} and reachable from the initial state.

A *signature* of a state $s \in Q$ is a projection of s onto a set of latches $\mathcal{S}ig \subseteq \mathcal{L}$, i.e., in this paper, we consider a signature to be just a subset of the bits of a state.

For any node g, we denote by $fanout(g)$ and $fanin(g)$ the set of fan-out and fan-in nodes of g, respectively.

For any fault-free node g, the variable, $\bar{g}$, denotes its faulty counterpart. We denote by $\bar{\mathcal{C}}$ a faulty circuit. To simplify our exposition, we will assume a single-stuck-at fault model: the notation $\bar{g}(0)$ ($\bar{g}(1)$) refers to a stuck-at-0 (stuck-at-1) fault at node g. However, our method works for any fault model that can be modeled using the SAT-based technique described next.

B. Fault Modeling

Similar to [], we use the SAT-based fault-modeling technique introduced by we augment the model of the fault-free circuit $\mathcal{C}$ (henceforth, $\mathcal{C}$') by adding a *mux* at the output of each gate g. Each *mux* has one *fault-select* and one *signal line*. We denote the set of *fault-select* lines by $E = \{e_1, e_2, \dots, e_n\}$ and the set of corresponding *fault-signal* lines by $W = \{w_1, w_2, \dots, w_n\}$, where $n = |\mathcal{G}|$. By setting $e_i = 1$, the node g_i is disconnected from $fanout(g_i)$, and w_i is connected to every node $g_j \in fanout(g_i)$.

This modeling framework allows us to disconnect any input of a gate from the circuit C and force and arbitrary faulty value instead. For example, for the single-stuck-at fault model, we would constrain that exactly one $e_i = 1$ (which enforces "single"), we would assign the corresponding w_i to be 0 (1) for stack-at-0 (stuck-at-1), and we would not allow e_i or w_i to change in different time-frames (which enforces "stuck-at").

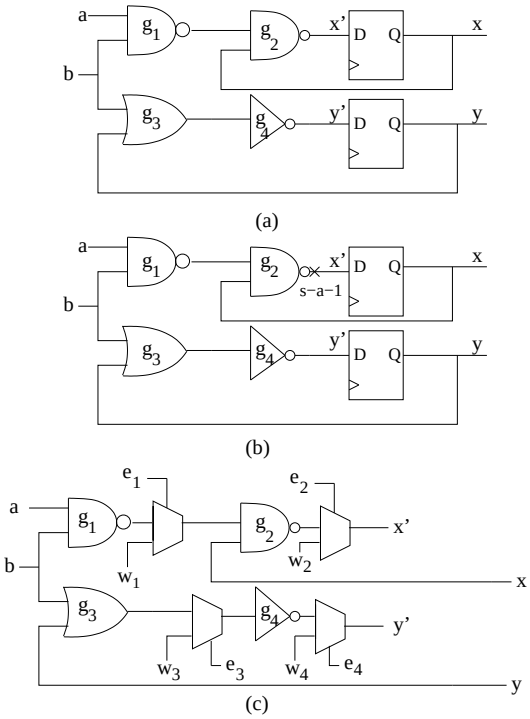

Fig. 1. Sequential Circuit (a) fault-free (b) faulty (c) added hardware

For example, consider the sequential circuit in Fig. 1(a). A possible faulty version is in Fig. 1(b), where a *s-a-1* fault is present in g_2. Fig. 1(c) shows the augmented circuit with a set of *muxes* used to model each fault. (For space reasons, the latches have been removed in Fig. 1(c).) The fault is modeled by setting fault-select line $e_2 = 1$ with all other $e_i = 0$, and fault-signal line $w_2 = 1$.

Bao: write some text about modeling the circuit using SAT model To communicate with the SAT solver, C is encoded in Conjunctive Normal Form (CNF) using Tseitin transformation with the use of auxiliary variables

C. FD-BackSpace

Since the proposed off-chip analysis is based on the FD-BackSpace framework, we briefly review the debug-flow here. For complete presentation we refer to []. FD-Backspace framework starts from the observed bug sighting (e.g. crash state)(s_n) and goes backwards towards the initial states, considering only the faults that are relevant to possible execution to the actual bug. Initially the possible fault set consists of all the possible faults in the circuit. Using formal analysis of the corresponding RTL (or other model), we compute the set of predecessor-candidates (P) of the crash state. If the fault cannot exist under the crash state, then the then P is empty and we remove the corresponding fault from the possible fault set. If P is non-empty, we load each predecessor state into a breakpoint circuit on the chip. Starting from the initial state (s_0) the chip is run full-speed and stops at the breakpoint state if the state exists in the failure trace path. The predecessor state is considered to be invalid if the chip does not reach the breakpoint state before reaching s_n. If a valid predecessor state is observed then it is considered as possible trace path from the s_0 to s_n under a given fault condition. As the

algorithm traverses through the state space, the possible fault list is pruned. This framework was able to reduce the possible fault set to a handful of faults that can be further analyzed to diagnose the root cause of failure.

III. Previous State Analysis

We start this section by formally defining unreachable states. A light-weight technique to detect unreachable states is then presented.

Definition 1: Given a circuit C, a trace π is a finite sequence of states $\pi = s_0 s_1 \cdots s_n$ where $n \in \mathbb{N}$ such that s_0 is an initial state and each pair $\{s_i, s_{i+1}\}$ in π is a legal transition in C.

Definition 2: Given a circuit C, a state s is said to be unreachable if and only if there does not exist a trace π such that $\pi = s_0 s_1 \cdots s$.

Lemma 1: Given a circuit C, if a state s is unreachable then for all $\pi = s_0 s_1 \cdots s$, there exists a pair $\{s_i, s_{i+1}\}$ that is not a legal transition in C

Proof: From Definition 2, it is trivial that if s is an unreachable state then all sequence $\pi = s_0 s_1 \cdots s$ is not a trace. From Definition 1, a sequence is not a trace when there exists a pair $\{s_i, s_{i+1}\}$ that is not a legal transition in C. ■

Theorem 1: Given a circuit C, a state s is unreachable if and only if for all reachable state s_r, the pair $\{s_r, s\}$ is an illegal transition in C.

Proof:
$\rightarrow$ direction:
Assume there exists a reachable state s_r that can reach s , a trace π is constructed such that it starts with an initial state s_0, gets to s_r and finally reaches s. Because s_r is reachable, all pairs from s_0 to s_r are legal transitions in C. Hence, $\pi_r = s_0, s_1, \cdots s_r, s$ is a trace and s is reachable. This is a contradiction and thus, there does not exist such reachable state s_r.
$\leftarrow$ direction:
Consider an arbitrary sequence $\pi = s_0 s_1 \cdots s_s$. If the last state before s is reachable then the last pair is an illegal transition from the assumption and thus π is not a trace. If the last state before s is unreachable, then π is obviously not a valid trace. Therefore, s is unreachable. ■

In order to prove a state s is unreachable, one can prove that all sequence $\pi = s_0 s_1 \cdots s$ contains at least one pair $\{s_i, s_{i+1}\}$ that is a illegal transition (Lemma 1) or prove that s cannot be reached from any reachable state (Lemma 1). These methods however require tremendous amount of work and are not feasible in practice. In this work, a light-weight method to verify whether a state s is unreachable is proposed. The next section describes the method and proves its correctness.

A. Unreachable State Identification

Theorem 1 provides an approach to verify whether a state s is unreachable. However, finding the complete set of reachable states R is a difficult task []. The following corollary of Theorem 1 shows how we abstract R and simplify the verification problem of an unreachable state.

Corollary 1: Given a circuit C and the set S where S is an abstraction of the set of reachable states, if for all state $s_i \in R$ the pair $\{s_i, s\}$ is an illegal transition then s is unreachable.

Proof: As S is an abstraction of R, it must contain all reachable state s_r. Because $s_r \in R$, $\{s_r, s\}$ is an illegal transition in C. From Theorem 1, the fact that all $\{s_r, s\}$ is illegal indicates that s is unreachable. ■

It is essential to note that the opposite statement of Corollary 1, "if there exists a state s_i such that pair $\{s_i, s\}$ is a legal transition in C, s is reachable " is not true. This is because S is just an abstraction of R and hence may still contains unreachable states. The state s is reachable only if s_i is reachable, which cannot be confirmed without further investigation.

From Corollary 1, a state s can be verified as unreachable by answering the question: is there any state that can transition to s given C? This question can be encoded as a SAT query. In this SAT query, the next-state signals are constrained as s while present-state signals are left unconstrained, and the SAT solver is asked to find a solution on the present-state signals. If there is a solution (SAT), no conclusion can be drawn on s. If there is no solution ($UNSAT$), s is unreachable.

In the SAT query, we leave the present-state signals unconstrained so that the SAT solver can find a solution in the set of all states, which of course is an abstraction of R. Nevertheless, this solution space can be reduced while still maintain the property as an abstraction of R. Specifically, if a state s is already found unreachable, s can be safely removed from the solution space. This is accomplished by simply forcing the solver to return a solution that is different from s. As the number of unreachable states found grows, this optimization becomes more powerful as it does not only reduce the search space for the solver but also prevents the case where an unreachable state is not detected due to spurious solutions.

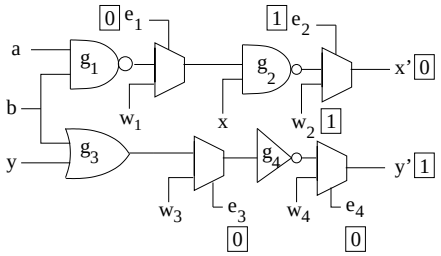

Fig. 2. Unreachable State Example

Example 1: Consider the example in Figure 2. The next-state signal x' is constrained to 0 and y' is constrained to 1. In this example, the circuit is also assumed under a stuck-at-1 fault at gate g_2. We want to find a present-state, an assignment on x and y, such that it would satisfy the next-state $< 0, 1 >$. Unfortunately, there are no values on x and y that can alternate the stuck-at-1 fault at gate g_2. This implies that there is no state that can transition to $< 0, 1 >$; hence $< 0, 1 >$ is unreachable under the stuck-at-1 fault at g_2. Using the same argument, $< 0, 1 >$ is also unreachable under the stuck-at-0 fault at g_4.

Algorithm 1 depicts the process of identifying an unreachable state. Given a circuit C, a fault p and a state s, the function returns whether s is unreachable under p. First, a CNF presentation of circuit C with the assumption of fault p is constructed

Algorithm 1: Unreachable State

input : circuit C
input : fault p
input : state s
output: bool unreachable

1 $\phi \leftarrow$ `CNF` (C, p);
2 $\phi \leftarrow$ Constrain_Next_State(s);
3 **if** `Solve` (ϕ) = *SAT* **then return** *false*;
4 **else return** *true*;

(line 1). Next, the next-state signals of the CNF instance are constrained to s (line 2). Then, the CNF instance is sent to the SAT solver. If there is a satisfying assignment, s is reachable and hence the algorithm returns false (line 3). Otherwise, the algorithm returns true indicating s is unreachable under p (line 4).

IV. Path Analysis

In this section, a technique to prune out suspect faults early in the run of FD-BackSpace is presented. This technique is based on the consistency between a fault and the chip's behavior.

Recall that in the FD-BackSpace framework, if under the assumption of a fault p, a state s has no predecessors, it is ignored and the algorithm continues to analyze fault p. In this case, the algorithm is conservative and assumes that the state s is not in the plausible path. This is sensible as proved in Section III, if a state s has no predecessor-states, it is unreachable under the fault p. Though, s is unreachable under p does not imply that it is unreachable under other faults. This scenario is illustrated in Example 2.

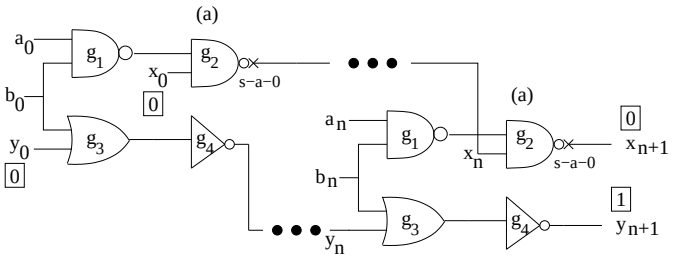

Fig. 3. Crash State and Initial State Example

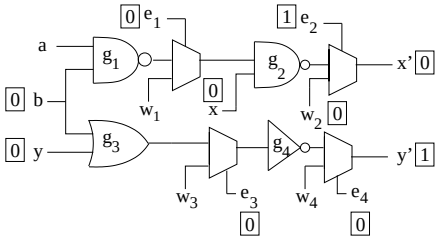

Fig. 4. Plausible Path Example

Example 2: Consider the scenario in Figure 3 where a faulty circuit is returned with a crash state and an initial state. In this case, the crash state is $< 0, 1 >$ and the initial state is $< 0, 0 >$. Now let us assume that the real fault is a stuck-at-0 at gate g_2. We have proved that the state $< 0, 1 >$ is unreachable under the faults, stuck-at-1 at gate g_2 and the stuck-at-0 at gate g_4. If the fault is stuck-at-0 at gate g_2, the state $< 0, 1 >$ is actually reachable from the initial state $< 0, 0 >$. Figure 4 shows a plausible path for the fault stuck-at-0 at gate g_2. In this case, $< 0, 1 >$ is actually observed as the crash state and that refutes stuck-at-1 at gate g_2 and the stuck-at-0 at gate g_4 as possible faults.

Now what if a state s is observed during a chip-run; this observation indicates that s should be reachable under the real fault. As a result, if s is observed and it is unreachable under a fault p, p cannot be the real fault and is invalidated as a possible fault. Algorithm 2 shows how to invalidate a fault p using this technique. Given a state s that is unreachable under fault p, a timeout and an initial state Q_0, the function returns whether p is an invalid fault. This function runs the chip to find if s can be observed (line 2). This is accomplished by using the original BackSpace framework. If s is observed, the function returns false indicating that p is an invalid fault; otherwise, true result is returned (line 3- 4). This technique although requires additional chip-runs has the ability to prune out spurious faults that can not be detected by the original FD-BackSpace framework.

Algorithm 2: Invalid Fault

input : fault p
input : set initial_state Q_0, state s
input : timeout
output: bool invalid_fault

1 // s is unreachable under p
2 Run_Chip(Q_0, s, timeout);
3 **if** s *is observed* **then return** *false*;
4 **else return** *true*;

V. Forward Traversal

This section proposes a simple heuristic to reduce the number of SAT calls in FD-BackSpace. This heuristic utilizes the fact that simulation is much faster than a formal engine.

In FD-BackSpace, a fault p is considered possible if under the assumption of p, there is a plausible path from a reachable state to the crash state. To find such path, the algorithm traverses from the crash state until it reaches a reachable state. During our analysis, we realize that the set of reachable states usually just contains one initial state. This of course hinders the traversal process as it is difficult to reach that one state. To tackle this problem, we use a fast simulation to find more reachable states.

Specifically, under the assumption of a fault p, we simulate the design from the initial state and collect states. As these states are simulated from the initial state, they are reachable and hence can be added to the set of reachable states. To reduce overhead, a timeout is set. Algorithm 3 shows the technique in detail. Given a circuit C, a fault p and a set of initial states Q_0, we expand the set of reachable states under p. First, a timeout is set to one second. Next, the circuit with the assumption of fault p is simulated until timeout is reached (line 3). During the simulation, a log file is utilized to store the state in each clock-cycle. Every state in the log file is then added to the set of reachable states (line 5).

Algorithm 3: Reachable State Set Expansion

input : circuit C
input : fault p
input : initial_state Q_0
output: reachable_states

```
// assign timeout to 1 second
timeout ← 1;
log ← Simulate(Q0, timeout) ;
foreach s ∈ log do
    reachable_states ← s ;
end
```

A. The Updated FD-BackSpace Algorithm

In this section, we present the updated algorithm of FD-BackSpace after integrating techniques mentioned in previous sections.

Algorithm 4 takes in the circuit C, the set of initial states Q_0, the crash state s_c, a timeout and the set of initial faults $\mathcal{G}$. At the end of its computation, this algorithm returns the set of possible faults $\mathcal{F}$. The set $\mathcal{G}_i$ is used to store invalid faults (line 1). For each fault, S_v, S_{uv}, S_r S_{ur} are sets of visited states, unvisited states, reachable states, unreachable states, respectively. To expand reachable state set for a fault p, a function Expand_Reachable_State_Set is called (line 5). Moreover, for each fault, two Boolean variables $valid$ and $invalid$ are employed to keep track of its status (line 7- 8). The loop is terminated when the fault p is proved to be invalid or valid (line 11). If S_{uv} is empty, p is an invalid fault as all states found during traversal are proved unreachable (line 12). When p is proved invalid, it is added to $\mathcal{G}_i$ (line 13- 14). At each iteration, a state s at the top of S_{uv} is analyzed (line 17). If s is reachable from an initial state, p is a possible fault and it is added to $\mathcal{F}$ (line 19- 21). Otherwise, it is tested on unreachability (line 23). If it is unreachable but observed in a chip-run, p is marked as invalid fault (line 25). However, if it can only be unreachable, it is added to S_{ur} (line 28). When an unreachability test cannot draw any conclusion on s, the algorithm attempts to go back further. All predecessors of s are found using the SAT solver (line 33). For each predecessor state s', it is carefully tested before added to S_{uv} (line 37). First, s' must be different from all found unreachable states, $s' \notin S_{ur}$, and is not visited before, $s' \notin S_v$ (line 35). Moreover, s' must be observed during a chip-run (line 37). These tests assure that s' has not been encountered before and it is a real state appearing in a chip-run.

VI. Experimental Results

This section presents the experimental results of the proposed framework. The presented techniques are implemented on top of the original FD-BackSpace framework from.All experiments are run on an Intel Core i5 3.1 GHz quad-core workstation with 8 GB of RAM. To interface with the original BackSpace framework, all circuits are synthesized using the BackSpace's cell library. We use Synopsys Design Compiler Version Y-2006.06-SP2 and Synopsys VCS Version A-2008.09 as our compiler and logic simulator, respectively.

Ten ITC'99 benchmark circuits are used in our experiments. After a circuit is synthesized, a gate-level netlist is obtained.

Algorithm 4: The Updated FD-BackSpace Algorithm

input : circuit C
input : set of initial states Q_0, crash state s_c
input : timeout
input : set of initial faults $\mathcal{G}$
output: final set of possible faults $\mathcal{F}$

```
1  set of invalid faults G_i;
2  foreach fault p ∈ G do
3      set of visited states S_v  unvisited states S_uv;
4      set of reachable states S_r  unreachable states S_ur;
5      Expand_Reachable_State_Set(C, p, Q_0, S_r);
6      current state s;
7      bool invalid = false;
8      bool valid = false;
9      S_uv ← s_c;
10     S_r ← Q_0;
11     while valid ≠ invalid do
12         if S_uv is empty then
13             invalid ← true;
14             G_i ← p ;
15         end
16         else
17             s ← S_uv.top();
18             S_v ← s;
19             if s ∈ S_r then
20                 valid ← true ;
21                 F ← p ;
22             end
23             else if Unreachable_State(C, p, s) then
24                 if Invalid_Fault(p, Q_0, s, timeout) then
25                     invalid ← true;
26                     G_i ← p ;
27                 end
28                 S_ur ← s ;
29             end
30             else
31                 φ ← CNF (C, p);
32                 φ ← Constrain_Next_State(s);
33                 while Solve(φ) = SAT do
34                     s' = Solve(φ) ;
35                     if (s' ∉ S_v) ∧ (s' ∉ S_ur) then
36                         Run_Chip(Q_0, s', timeout);
37                         if s' is observed then
                               S_uv.push_back(s');
38                         end
39                     end
40                 end
41             end
42         end
43     end
```

A s-a-0 or a s-a-1 is then randomly inserted to each gate-level netlist. For each benchmark, we simulate the gate-level netlist with the accompanying test-bench. We then collect the crash state. Experiments are conducted with five different version of Algorithm 4. In the first version, we turn off all new techniques to represent the the original FD-BackSpace framework **FD-BackSpace**. Then we turn on each technique individually, unreachable state identification **UR**, reachable states expansion **RS**, invalid fault removal **IF + UR**. Note that in order to apply

IF, we need to detect unreachable states first and hence UR must be turned on for IF to work. As a result, we encode the invalid fault removal optimization as **IF + UR**. Finally, we apply all new optimizations **UR + RS + IF**.

Table I displays the information of each benchmark. The first column gives instance names. The next two columns respectively show the numbers of flops and the initial numbers of faults.

TABLE I. INSTANCE INFORMATION

Instance Name	No. of Flops	Initial No. of Faults
b01	5	146
b02	4	70
b03	30	382
b04	66	1406
b05	34	4140
b06	9	146
b07	49	886
b08	21	484
b09	28	403
b10	17	426

Table II shows the results of all our experiments. The first column gives the instance name. The next three columns respectively shows the final numbers of faults, the run-times and the numbers of chip-runs under the original FD-BackSpace framework. Columns five, and six shows the run-times and numbers of chip-runs under **UR**. Column seven and eight give the run-times and the numbers of chip-runs under **RS**. Note that these optimizations cannot detect more invalid faults than the original FD-BackSpace framework. Hence, the final numbers of faults in these experiments stay the same as in the original FD-BackSpace. These numbers are hence omitted for the sake of the length of the paper. Column nine, ten and eleven respectively show the final numbers of faults, the run-times and the numbers of chip-runs under **IF + UR** In this case, the final numbers of fault are different from the original FD-BackSpace and hence are reported. The last two columns give the run-times and the numbers of chip-runs when all optimizations are turned on (**UR + RS + IF**). We do not report the final numbers of faults in this case because they are the same as when **IF + UR** is applied.

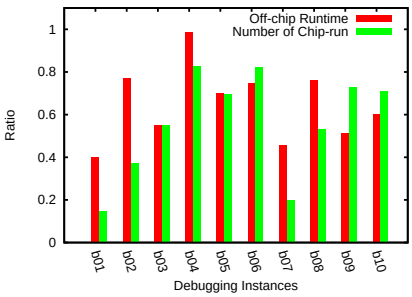

Fig. 5. Run-time and Numbers of Chip-runs Ratios

Figure 5 plots the ratios of run-times and the numbers of chip-runs between the **UR + RS + IF** and the original FD-BackSpace. Our algorithm outperforms the original FD-BackSpace framework in all cases both in run-times and the numbers of chip-runs. Specifically, in b01, we are able to obtain 60% reduction in run-times and 85% reduction in the numbers of chip-runs. Note that for b01, our algorithm also finds six invalid faults more than the original FD-BackSpace algorithm.

It is essential to note that the unreachable state identification technique poses quite an overhead in run-times when applied individually. This technique requires the system to make a lot more SAT queries and hence increases the run-times significantly. However, this extra run-times come with the benefit of less chip-runs. Given that each chip-run can take hours to set-up and run, the extra time spent in the analysis software is justified. Moreover, identifying unreachable states is crucial for the **IF + UR** optimization later. The **IF + UR** optimization gives us the ability to find invalid faults early in the run. This means a lot less states to analyze compared to **UR** and FD-BackSpace. As a result, while **IF + UR** still makes extra SAT queries on unreachable analysis, it is a lot less compared to when only **UR** is applied. Furthermore, in some cases while FD-BackSpace and **UR** go further back and makes extra SAT queries, **IF + UR** marks a fault invalid and stops analyzing that fault. In fact, as seen from the table, **IF + UR** reduces the numbers of chip-runs, finds more invalid faults and has comparable runtimes compared to FD-BackSpace. The **RS** optimization of course reduces the run-times and chip-runs as it allows us to stop much earlier during the path traversal compared to FD-BackSpace. Overall, when applying all optimizations we are able to obtain 36% reduction in run-times and 44% reduction in the numbers of chip-runs compared to the original FD-BackSpace. This shows the effectiveness of our methods.

VII. CONCLUSION

This work improves the original FD-BackSpace framework by proposing three optimizations. First, a formal mechanism to detect unreachable states without chip-run trials is introduced. These unreachable states are used to find invalid faults early in the run. Finally, we propose a light-weight technique to expand the set of known reachable states from the set of initial states. The end result is significant reduction both in run-times and the numbers of chip-runs, demonstrating the practicality of our techniques.

REFERENCES

[1] M. Prabhu and J. A. Abraham, "Functional test generation for hard to detect stuck-at faults using rtl model checking." in *European Test Symposium*. IEEE Computer Society, 2012, pp. 1–6.

[2] L. Zhang, I. Ghosh, and M. Hsiao, "Efficient sequential atpg for functional rtl circuits," in *ITC'03*, 2003, pp. 290–298.

[3] L. Lingappan, V. Gangaram, N. K. Jha, and S. Chakravarty, "Fast enhancement of validation test sets for improving the stuck-at fault coverage of rtl circuits," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 5, pp. 697–708, May 2009.

[4] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for socs," in *Proceedings of the 43rd annual Design Automation Conference*, ser. DAC '06. ACM, 2006, pp. 7–12.

[5] X. Liu and Q. Xu, "On multiplexed signal tracing for post-silicon debug," in *DATE*, 2011, pp. 685–690.

[6] J.-S. Yang and N. A. Touba, "Efficient trace signal selection for silicon debug by error transmission analysis," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 442–446, 2012.

[7] D. Sengupta, F. M. De Paula, A. J. Hu, A. Veneris, and A. Ivanov, "Lazy suspect-set computation: Fault diagnosis for deep electrical bugs," in *GLSVLSI'12*, 2012, pp. 189–194.

TABLE II. UPDATED FD-BACKSPACE RESULTS

Instance Info	**Original FD-BackSpace**			**Unreachable State(UR)**		**Reachable State(RS)**		**Invalid Fault(IF + UR)**			**UR + RS + IF**	
Instance Name	Final No. of Faults	Runtime (s)	No. of Chip-run	Runtime (s)	No. of Chip-run	Runtime (s)	No. of Chip-run	Final No. of Faults	Runtime (s)	No. of Chip-run	Run-time (s)	No. of Chip-run
b01	62	458	2068	689	1763	250	348	56	321	1334	183	302
b02	5	57	86	169	43	50	45	4	52	43	44	32
b03	3	382	96530	1706	95991	249	62921	3	364	91981	210	53066
b04	1	1166	109870	1382	109256	1114	104970	1	1166	92179	1153	91151
b05	4	8200	1994094	14931	1981779	8576	1987938	4	6269	1515094	5756	1391112
b06	2	181	1088	347	955	163	896	2	295	942	135	896
b07	219	5423	958083	7072	906767	2938	194314	205	4328	841090	2469	192482
b08	2	3954	217483	5505	211969	3803	212424	2	4212	162182	3015	116092
b09	3	1164	139910	1402	137336	719	128599	3	1364	133613	597	101798
b10	9	481	677	775	625	300	482	8	589	601	289	482

[8] Y.-C. Hsu, F. Tsai, W. Jong, and Y.-T. Chang, "Visibility enhancement for silicon debug," in *DAC '06*. ACM, 2006, pp. 13–18.

[9] ARM, *Embedded Trace Macrocell Architecture Specification*, July 2007, vol. 20, ref: IHI0014O.

[10] C.-C. Yen, T. Lin, H. Lin, K. Yang, T. Liu, and Y.-C. Hsu, "Diagnosing silicon failures based on functional test patterns," in *MTV '06*. IEEE Computer Society, 2006, pp. 94–98.

[11] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, pp. 1606–1621, 2005.

[12] G. S. Tseitin, "On the complexity of derivations in the propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic*. New York - London: Part 2. Consultants Bureau, 1968, pp. 115–125.

[13] M. Case, A. Mishchenko, and R. Brayton, "Inductively finding a reachable state space over-approximation," *Proc. IWLS06*, pp. 172–179, 2006.