Lazy Suspect-Set Computation: Fault Diagnosis for Deep Electrical Bugs

Dipanjan Sengupta
Dept. Elec. & Comp. Eng.
University of Toronto
Toronto, Canada
dipanjan@eecg.toronto.edu

Flavio M. de Paula
Dept. of Computer Science
University of British Columbia
Vancouver, Canada
depaulfm@cs.ubc.ca

Alan J. Hu
Dept. of Computer Science
University of British Columbia
Vancouver, Canada
ajh@cs.ubc.ca

Andreas Veneris
Dept. Elec. & Comp. Eng.
University of Toronto
Toronto, Canada
veneris@eecg.toronto.edu

André Ivanov
Dept. Elec. & Comp. Eng.
University of British Columbia
Vancouver, Canada
ivanov@ece.ubc.ca

ABSTRACT

Current silicon test methods are highly effective at sensitizing and propagating most electrical faults. Unfortunately, with ever increasing chip complexity and shorter time-to-market windows, an increasing number of faults escape undetected. To address this problem, we propose a novel technique to help identify hard-to-find electrical faults that are not detected using conventional test methods, but manifest themselves as observable functional errors during functional test, system test, or during actual use in the field. These faults are too sequentially deep to be diagnosed using simulation, ATPG, or formal tools. Our technique relies on repeated full-speed chip runs that witness the functional bug, combined with some additional on-chip functional debug support and off-line analysis, to compute a possible set of suspected faults. The technique quickly prunes the suspect set, and for each suspect, it can provide a short test vector for further analysis. Experiments on the ITC'99 benchmarks demonstrate the effectiveness of our approach.

Categories and Subject Descriptors

B.7.2 [**Design Aids**]: Verification; B.7.3 [**Reliability and Testing**]: Testability

General Terms

Algorithms, Performance, Verification

Keywords

Electrical Fault, Post-Silicon Debug, Satisfiability

1. INTRODUCTION

One of the most challenging problems in post-silicon debug is the diagnosis of a fault that has eluded conventional testing but manifests only later as an observable functional error when the chip is running full-speed during bring-up, system test, or actual use. We dub these bugs "deep electrical bugs" because they have been first sighted only at extreme sequential depth (e.g., many billions of cycles after only seconds of silicon run time). In this paper, we propose a novel technique to help diagnose these bugs.

Existing methods from pre-silicon verification or manufacturing test do not solve this problem. Pre-silicon verification does not consider electrical faults. One could imagine mutating the RTL with a postulated fault and then applying simulation or formal verification — and indeed, this is the primary debugging technique once the set of possible faults is greatly narrowed, and a very short trace demonstrating the bug has been captured — but the vast set of possible faults, the slow speed of simulation, and the capacity limits of formal verification prevent employing these techniques initially for fault diagnosis. ATPG and other methods from manufacturing test explicitly consider faults, but are also inadequate for initial diagnosis of deep electrical bugs. In particular, ATPG algorithms face two fundamental complexity limits. First is the need to cover a fault model. Because the goal is 100% coverage, every possible fault must be analyzed, even if they are irrelevant to a specific, sighted bug. The second limit is even worse: sequential ATPG algorithms [13] blow-up exponentially in the sequential depth (number of time frames). As noted earlier, if we are trying to debug a crash that occurs after a few seconds of the actual silicon running, then it corresponds to billions of cycles of sequential depth — well beyond the capabilities of sequential ATPG methods.

The post-silicon debug team has the luxury of sequential depth, as they are the ones running applications full-speed on real silicon. In our problem scenario, they are the ones who sight the bug. But the almost complete lack of controllability and observability forces them into an extremely challenging, ad hoc debug flow [6]. Some techniques can enhance observability, but these are slow and very limited. For example, physical probing can measure the voltages on a handful of on-chip signals (as long as accessible probe points are available) [5]. However, decreasing feature-size, flip-chip technologies, and growing complexity of chips make such a method cumbersome [7]. Scan chains [11] can aid observability, but typically provide only one-cycle snapshot of the chip's scan chain. Trace buffers [1] provide a recording of

GLSVLSI '12 May 3–4, 2012, Salt Lake City, Utah, USA.

hundreds or more cycles, but only of a few key signals. All of these methods help, but it is exceptionally hard to get that scan or trace buffer dump that captures the exact moment that root-causes a bug.

Recently, researchers have started proposing methodologies to harness the on-chip test and debug hardware to provide greater assistance to the post-silicon debug team. For example, scan dumps can be compared automatically between good and bad runs to help root-cause a fault [2], or propagated forward and backward to help improve visibility and diagnosis [10]. 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. These methods assume the system behavior is deterministic, which is rarely true in practice (due to test case randomness, incomplete control of the operating environment, clock domain crossings, arbitration, etc.). The BackSpace [3] approach can handle non-determinism, but does not consider electrical faults.

Building on those previous works, this paper presents FD-BackSpace (Fault Diagnosis BackSpace) to provide assistance for diagnosing sequentially deep electrical bugs. In particular, the contributions are as follows:

- We propose a novel algorithm that identifies a small set of possible faults that could be responsible for the observed buggy behavior running the actual system in silicon.
- The key characteristic of our method is its laziness. Rather than trying to analyze whether a deep bug could occur for every possible fault, our method starts from the observed bug sighting (e.g., a system crash) and goes backwards, lazily considering only the faults that are relevant to a possible execution leading to the actual bug.
- Experimental results show that our method can reduce the number of suspect faults by an average of 94%. Furthermore, we can eliminate the majority of possible faults within only few clock cycles (going backwards from the bug observation). Even if we cannot run the algorithm to completion, we still greatly reduce the set of possibilities for the debug team to consider.
- Not only does our technique produce a small suspect set of possible faults, it simultaneously reconstructs a trace showing for each fault how that fault can cause the observed buggy behavior.
- Unlike [3,4], our method handles deep electrical faults, being specifically used for fault diagnosis. In contrast to [2, 10, 12], our method handles non-determinism in the system execution.[1] Unlike [8], our method is not processor-specific and can be applied to any design. Unlike [9], which assumes the existence of test vectors demonstrating faulty behavior before fault diagnosis can be performed, our method simultaneously constructs plausible test vectors along with diagnosing possible faults.

2. PRELIMINARIES

2.1 Notation and Basic Definitions

Let $\mathcal{C}$ be a fault-free circuit. We model $\mathcal{C}$ as a finite state machine, $M = (Q, I, O, Q_0, \delta, \omega)$, where:

- $Q = 2^{\mathcal{L}}$ is the set of states, where $\mathcal{L}$ is the set of latches in C;
- I is the input alphabet;
- O is the output alphabet;
- $Q_0 \subseteq Q$ is the set of initial states;
- $\delta \subseteq Q \times I \times Q$ is the transition relation;
- $\omega \in Q \times I \mapsto O$ is the output function.

Notice that we model M as a non-deterministic finite state machine, so the formalism can handle randomness in the bring-up tests as well as transient errors, race conditions, etc.

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.

2.2 Fault Modeling

We use the SAT-based fault-modeling technique introduced by [9]: 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 is a very flexible fault-modeling framework, as any gates can be disconnected, with arbitrary faulty values forced into the circuit 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").

Consider the sequential circuit in Fig. 1(a), for example. 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 the 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 $w_2 = 1$.

With the (potentially faulty) circuit in this form, many useful properties are easily phrased as SAT queries. For example, an important basic computation is pre-image: what states/inputs are possible predecessors of a given state? To answer this question, we simply constrain that next-state signals to the given state and ask the SAT solver for solutions for the present-state and input signals. Upon receiving

[1] Our experimental results are done with deterministic simulation, but we show how to relax this assumption to handle non-determinism.

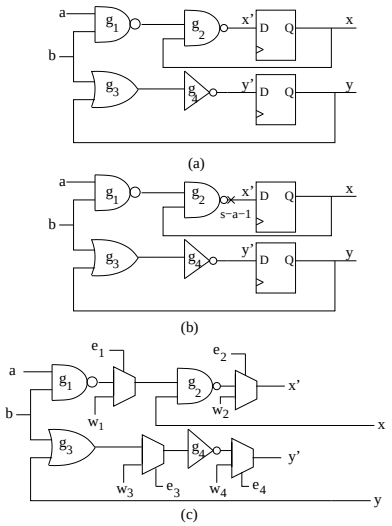

Figure 1: Sequential Circuit (a) fault-free (b) faulty (c) added hardware

a solution, we can force the solver to return additional solutions until we have them all. Returning to Fig. 1, for example, we might ask what states are the predecessors of state 11 under input 01, assuming a fault-free circuit. In that case, we would assert that $x' = y' = b = 1$ and $a = e_i = 0$, and a SAT solver would quickly tell us that this is actually impossible: no solutions for x and y exist. Similarly, we might ask which single-stuck-at faults allow the circuit to transition from state 10 to state 10 under input 01 by asserting $x' = x = b = 1$ and $y' = y = a = 0$ and leaving the fault-select and fault-signal lines partially constrained, and the SAT solver would return the fault in Fig. 1(b) (along with several other solutions). The challenge with using SAT for diagnosing deep bugs, though, is that we don't know the states leading up to the bug, so we don't know the correct constraints for the SAT solver. Our new algorithm solves this problem.

2.3 BackSpace

Since our technique relies on the same formal principles of BackSpace, we briefly review its core assumptions and debug-flow. For a complete presentation, we refer to [3].

The core assumptions of the BackSpace framework are:

- It must be possible to recover the state of the chip when an error has occurred. For example, this could be done via the scan chain with the chip in test mode;
- The silicon implements the RTL (or gate-level or layout or any other model of the design that can be analyzed via formal tools);
- The bring-up tests can be run repeatedly and the bug being targeted will be at least somewhat repeatable (one out of every n tries, for a reasonably small value of n).

The BackSpace framework consists of adding some debug support to the chip: a signature that saves some history information but otherwise has no functional effect on the chip's behavior, and a programmable breakpoint mechanism that allows the chip to "crash" when it reaches a specified state. Given these, the approach repeats the following steps

1. Run the chip until it crashes or exhibits the bug. This could be an actual crash or a programmed breakpoint.
2. Scan out the full crash state, including the signature.
3. Using formal analysis of the corresponding RTL (or other model), compute the set of predecessor-candidates of the crash state. The signature must provide enough information so that the number of predecessor-candidates is reasonably small.
4. For each predecessor-candidate s, let s be the new breakpoint; re-run the chip; if the chip reaches the breakpoint, then s is a valid predecessor.

until it has computed enough of a history trace to debug the design (or Step 3 fails). Each iteration of the loop is like hitting "backspace" on the design – going back one cycle.

Our new algorithm, FD-BackSpace, differs from the original BackSpace in two fundamental ways: 1) we do not assume that the silicon implements the RTL, instead use the techniques from Sec. 2.2 to model faults, and 2) we consider all valid paths to the crash state instead picking just one. The next section presents the FD-BackSpace algorithm.

3. FD-BackSpace ALGORITHM

We start this section by setting forth our assumptions, then we present the algorithm, and afterwards, we prove its correctness.

Similar to the original BackSpace framework, we assume that it is possible to recover the state of the chip when it crashes (or breakpoints) along with a signature of the previous cycle (using scan-chains). In addition we make the following assumptions: 1) we have a fault model that produces a finite set $\mathcal{G}$ of possible faults that might be the cause of the crash and that we can model each fault as in Sec. 2.2; 2) if a fault exists, it will be excited and observed given a reasonably small number of chip-run trials; 3) and the signature is effective at constraining the pre-image computation to yield a small number of predecessor-candidate states. The last two assumptions are in-common with the original BackSpace work, and effective techniques for signature computation are known [3].

Our new FD-BackSpace algorithm is shown in Algorithm 1. It computes a minimal set of the suspect faults, working backwards in time from the crash state towards the initial states. The three main data-structures in this algorithm are two tables, W and H, and the set of suspect faults, $\mathcal{F}$. Each table entry contains a 3-tuple: a state, a signature and a set of faults. We define W as the working-table and H as the history table. The former is a dynamic table where elements are inserted and/or deleted in each iteration. The latter stores all validated states (explained later) and their associated possible faults.

The intuition behind Algorithm 1 is that we are performing a modified graph traversal through the state space under different possible faults. The working table W is the frontier from which we must continue the search, and the history table H accumulates all states we have visited. The key to understanding the algorithm is that we are always maintaining two invariant properties of all triples in W and H: 1) for every state and associated fault, there is an execution path from that state through only states in H to the crash state, assuming that fault; and 2) every state is *validated*, meaning that we actually observed that state occurring on an actual silicon run.

This procedure has 2 main loops: lines (15 – 29) and lines (30 – 57). The first loop (lines 15 – 29) builds the initial working-table. This loop iterates over all possible faults $\mathcal{G}$. First, it computes a pre-image of the crash state under some fault $g \in \mathcal{G}$ (line 17). If the pre-image is non-empty, Algorithm 1 checks whether each state s in the pre-image is already in W, adding g to the set of possible faults

of s, if that is the case. Otherwise, Algorithm 1 validates and inserts s into W by running the actual chip (this step is explained in detail later). At the end of this loop, W contains all valid predecessor-states of the crash state, each of which is associated with its set of possible faults.

The second loop (30 – 57) is essentially just an inductive repetition of the first loop, expanding the graph traversal backward through the state space and building up paths that lead to the crash state under the presence of some fault. First, Algorithm 1 reads the first entry of W (deleting the entry from the table) containing a state s, a signature ξ, and a set of possible faults P_s associated with s. It either copies this entry to the history-table or updates (adding) the set of possible faults for the existing state s. Next, Algorithm 1 iterates over P_s. Different from the previous loop, here we have a few more cases to consider. If a state in the pre-image of s (denoted as s'), under the assumption of a fault p, is in the initial set of states Q_0, then we can safely add it to the final suspect set $\mathcal{F}$. Otherwise, we have three options: a) if s' is already in the history-table, then we can proceed to the next state in the pre-image (thus, we handle paths containing cycles); b) if s' is already in the working-table but p is not in its set, we add p to P'_s (i.e., many faults can explain the same state); c) if s' is in neither table, then it needs to be validated and inserted into W (if valid). Once Algorithm 1 iterates over all faults in P_s, it removes the next entry in W, (i.e., it iterates at line 31). This loop terminates only when the working-table is empty. Thus, $\mathcal{F}$ contains only the faults than can lead the chip from an initial state to the crash state.

Note that Algorithm 1 interleaves off-line software computation with hardware runs exactly as in BackSpace. The hardware interaction is encapsulated by the routine *insertIfValid()*. This routine loads the breakpoint circuitry with the state to be validated, sets a timeout value for each run, runs the chip, and then dumps the signature of the validated state. If the breakpoint is hit, the state is validated; otherwise, the timeout will occur. This routine also handles non-determinism as in the original BackSpace framework: the parameter *ntrials* specifies the number of times the chip must be run before giving up on validating a state. If the chip does not reach the state in any of the *ntrials* run, only then do we conclude that the state cannot be reached by the chip. Setting *ntrials* large enough makes the probability of erroneously not validating a state arbitrarily small.

The termination described in Algorithm 1 is simplified by assuming we can continue until we reach the initial states, but we can easily generalize it. As described, the only termination condition is when W becomes empty, which can only happen if the algorithm reaches the initial set of states (from which point the preImage() returns the empty set). State traversal to any state in Q_0 may take too much time or memory, e.g., if the crash state is too deep from the initial set of states. But it is not difficult to see that if computing a small prefix of the crash state is sufficient to eliminate a large number of faults, then we can simply augment the condition at line 30 to limit to some bounded number of loop iterations. A similar argument could limit to computing a pre-determined number of faults. The last condition not explored in this simplified presentation of the algorithm is the case when the pre-image computation of some state s (lines 17 and 39) grows too big. In this case, we can terminate the computation for s and (conservatively) add its fault to the suspect-set.

To complete the presentation of Algorithm 1, we need to prove its correctness. We start by introducing some definitions. Then, we formally state and prove the correctness properties of the algorithm (under the assumption that

Algorithm 1 Suspect-Fault-Set Lazy Computation

```
 1: input C : circuit-under-debug
 2: input Q_0 : set of initial states
 3: input cs : crash state
 4: input ξ_cs : signature of predecessor state of crash state
 5: input G : set of all possible faults (as described in Sec. 2)
 6: input timeout, ntrials: parameters to isValid()
 7: output F: final suspect-fault set
 8: /* Global variables and structures */
 9: s, s' : states
10: ξ, ξ' : signatures of the predecessor-state of s and s'
11: P_s : set of possible faults associated with state s
12: W : working-table, where each entry is a 3-tuple (s, ξ, P_s)
13: H : history-table, a non-destructive version of table W
14: F := ∅; P, W, H := NULL
15: for each fault g in G do
16:     /* C' is the faulty version of C with fault g */
17:     if (I := preImage(cs, ξ_cs, C') ≠ ∅) then
18:         for each state s' in I do
19:             if isStateMember(W, s') then
20:                 /* add g to s' possible-fault set */
21:                 UpdateFaultSetOfState(W, s', g)
22:             else
23:                 /* validate s' on-chip assuming g and
24:                  insert new entry into working-table W */
25:                 insertIfValid(W, s', g, timeout, ntrials)
26:             end if
27:         end for
28:     end if
29: end for
30: while (W ≠ ∅) do
31:     (s, ξ, P_s) = Delete(W[0]) //destructive read
32:     /* Manage history-table H */
33:     if isStateMember(H, s) then
34:         UpdateFaultSetOfState(H, s, P_s)))
35:     else Insert(H, (s, ξ, P_s))
36:     end if
37:     for each fault p in P_s do
38:         /* C' is the faulty version C with fault p */
39:         for each state s' in preImage(s, ξ, C') do
40:             if s' ∈ Q_0 then
41:                 F := F ∪ {p}
42:             else if isFaultMember(H, s', p) then
43:                 /*i.e., (s', p) has already been tested*/
44:                 continue
45:             else if isStateMember(W, s') then
46:                 if !isFaultMember(W, s', p)) then
47:                     /* append p to s' possible-fault set */
48:                     UpdateFaultSetOfState(W, s', p)
49:                 end if
50:             else
51:                 /* validate s' on-chip assuming p and
52:                  insert new entry into working-table W */
53:                 insertIfValid(W, s', p, timeout, ntrials)
54:             end if
55:         end for
56:     end for
57: end while
58: return (F)
```

ntrials is large enough that *insertIfValid()* does not fail to validate any valid state).

DEFINITION 1. *Given a physical chip and a circuit model $\bar{\mathcal{C}}$ with the same state bits/flops, a "plausible path" is a finite sequence of states such that every state in the sequence is reachable on the physical chip, each pair of successive states in this sequence is a legal transition in $\bar{\mathcal{C}}$, and the sequence ends at the crash state of $\bar{\mathcal{C}}$.*

The intuition behind this definition is to capture the best approximation to what we really want, given the information we can gather. What we would really like to compute is an actual execution path of the faulty physical chip that led to the observed crash, but this is impossible, since we do not have full trace-visibility of the chip. A plausible path is an execution path of the *model of* the faulty chip, with the extra information that every state on this path was really reachable on the physical chip. It is possible for a plausible path to connect states of the physical chip together in a way that is possible in the model, but not in the silicon, so plausible paths are an imperfect approximation. But the important point is that every real execution of a faulty chip will also be a plausible path, so considering all plausible paths guarantees not missing possible faults.

DEFINITION 2. *An "initial plausible path" is a plausible path that starts from the initial state of $\bar{\mathcal{C}}$.*

This definition restricts Defn. 1 to paths that start at an initial state, giving a complete execution from reset to the crash state, assuming some fault. And every state on this execution has been validated as a real state on the silicon.

THEOREM 1. *For every fault $g \in \mathcal{F}$ in Algorithm 1, the computed paths are initial plausible paths.*

Proof: No triple is added to F until the algorithm reaches an initial state, so any path starting from that state is obviously initial. Algorithm 1 maintains invariants that 1) for every state and associated fault, there is an execution path from that state through only states in H to the crash state, assuming that fault; and 2) every state is *validated*. These two invariants are maintained because the only way for a state/fault to be added to W or H requires that they first be in the pre-image of a state in W or H, hence guaranteeing the first invariant; and that they also be validated, hence guaranteeing the second invariant. These two properties combine to show the existence of the plausible path. ∎

THEOREM 2. *For every fault $g \in \mathcal{G}$, if there exists an initial plausible path in $\bar{\mathcal{C}}$, then $g \in \mathcal{F}$.*

Proof: Since there exists an initial plausible path, as we start from the crash state in the first loop of Algorithm 1, we will find the preceding state of the initial plausible path in the pre-image, under fault g. This will validate (because of our assumption that *ntrials* is sufficiently large), so that triple will be added to W and H. Subsequently, in the second loop, whenever we encounter a state on the initial plausible path, we will find the preceding state of the path in the pre-image, under fault g, and validate it. This will proceed until we reach the initial states, whereupon g will be added to F. ∎

The two theorems state what is promised by our algorithm. Theorem 1 means that every fault returned should be considered seriously by the debug team, because under the assumption of that fault, there is a possible execution from initial state to the crash state, in which every state on that execution has been validated on the silicon. Theorem 2 means that our algorithm will never miss a fault from G that could have explained the bug (assuming *ntrials* is sufficiently large).

Table 1: Suspect Set Computation Results

Circuit Name	No. of Flops	Initial No. of Faults	Final No. of Faults	Runtime	Suspect Set reduction (%)
b01	5	146	10	1m 37s	93
b02	4	70	3	32s	95
b03	30	382	3	4m 42s	99
b04	66	1406	1	54m 40s	99
b05	34	4140	4	34m 41s	99
b06	9	146	39	2m 51s	74
b07	49	886	173	75m 7s	80
b08	21	484	2	66m 17s	99
b09	28	403	3	27m 47s	99
b10	17	426	9	17m 53s	97
b12	121	2802	4	640m 25s	99
AVG					**94**

4. EXPERIMENTAL RESULTS

In this section, we present experimental results of our proposed flow for identifying single-stuck-at faults during post-silicon debug. Experiments were conducted on an Intel i5, 3.1 GHz workstation with 16GB of RAM. We report experiments on eleven ITC'99 benchmark circuits. We implemented Algorithm 1 based on the open-source BackSpace v0.3 codebase.[2] To interface with BackSpace, we had to synthesize these benchmarks using BackSpace's cell library. We used Synopsys Design Compiler Version Y-2006.06-SP2. Also, we used Synopsys VCS Version A-2008.09 as our logic simulator.

Recall that BackSpace uses a signature to reduce the size of pre-images. Choosing a good signature depends on knowledge of the circuit [3] or an advanced algorithm for signal selection [7], but this is orthogonal to our work. Thus, for simplicity, we *randomly* chose signatures. Moreover, the signature size was set for each circuit such that the number of possible predecessor states never exceeded 1024.

After synthesizing each benchmark, we randomly inserted either a *s-a-0* or a *s-a-1* in the gate-level netlist. For each experiment, we simulated the gate-level netlist with the accompanying test bench for an arbitrary number of cycles; randomly selected a crash state; and then we started Algorithm 1.

Table 1 shows the results of all our experiments. The first three columns provides details of the circuits. The next two columns show the size of the final suspect set and the runtime of our algorithm. We also report the percentage reduction in the suspect set size (column 6). Note that in the majority of the benchmark circuits the final suspect fault set is extremely small. Thus, this set can now be efficiently handled by ATPG for identifying the actual fault on the chip.

Figure 2 shows the reduction in the set of faults using Algorithm 1 for each circuit. The x-axis represents the clock cycles, starting from the crash state ($x = 0$) to the initial state. (Note that we simulated each circuit for 100 clock cycles before picking a crash state, but due to the presence of loops, some circuits have initial plausible paths of less than 100 states. For example, the initial plausible path for *b*06 has 39 states. Thus, for some of the circuits, Algorithm 1

[2] `http://www.cs.ubc.ca/~depaulfm/BackSpace`

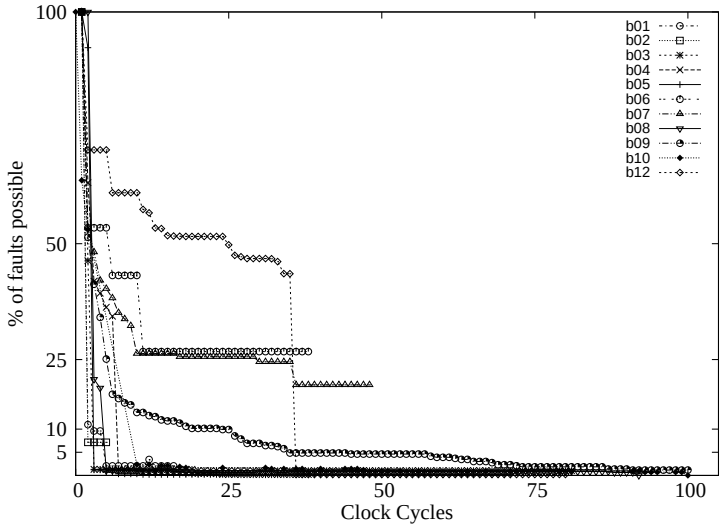

Figure 2: Reduction in possible fault list while tracing from crash state to initial State

Table 2: Suspect-Fault Set Reduction

Circuit Name	$\|\mathcal{F}\| = 25\%$ of $\|G\|$	% reduction in 10 clock cycles
b01	2	93
b02	2	95
b03	3	99
b04	7	99
b05	3	99
b06	>100	74
b07	11	74
b08	6	99
b09	5	86
b10	10	97
b12	38	50

computes the shorter path and terminates well below 100 clock cycles.) Notice that we can negate a significant number of faults within the first few iterations. Thus, when tracing to the initial state is impractical, one can stop Algorithm 1 after a few iterations.

Table 2 shows the results where the algorithm terminates prematurely before reaching the initial state. Column 2 shows the minimum number of cycles to be backspaced such that the suspect set size is below 25% of the total faults. In column 3 we show the percentage of fault reduction when the size of the plausible path reaches 10, i.e. maximum number of clock cycles to be iterated is 10. Results show that it is not necessary to trace back to the initial state, in case the crash happens too deep from the initial state. This would lead to considerable reduction in runtime without compromising the effectiveness of our approach.

The plausible path can be considered as the *test vector* in our problem. Unlike ATPG, we do not have any control over these vectors. However, results show that our lazy computation of the suspect set of faults is very efficient in reducing the number of suspect faults. The key insight is that we can use actual "buggy" traces, that lead the chip to a crash state, to prune out suspect faults (instead of using the very expensive, exhaustive methods of automatic test pattern generation).

5. CONCLUSION

We have presented a novel framework to aid debugging deep electrical faults in silicon. Assuming that we can model the faults, our method computes a small set of possible suspects that could explain the silicon's malfunction. In addition, the method also reconstructs, for each fault, plausible paths that lead the silicon to the actual bug. Key to our approach is the laziness of this computation. We avoid the expensive task of computing very long test-vectors for each and every fault. Our experiments show that our method is effective with much less effort. We are able to reduce the suspect set of faults by an average of 94%, and we show that we need only a few cycles from the buggy state to eliminate the majority of the possible faults. The direct line of future work is to target larger designs, experiment with more general fault models, and investigate efficient methods for reducing the runtime of the algorithm by using on-chip circuitry, such as trace buffers.

6. REFERENCES

[1] ARM. *Embedded Trace Macrocell Architecture Specification*, volume 20. July 2007. Ref: IHI0014O.

[2] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch divergency in microprocessor failure analysis. In *ITC'03*, pages 755–763, 2003.

[3] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang. Backspace: formal analysis for post-silicon debug. In *FMCAD '08*, pages 5:1–5:10. IEEE Press, 2008.

[4] F. M. de Paula, A. Nahir, Z. Nevo, A. Orni, and A. J. Hu. Tab-backspace: unlimited-length trace buffers with zero additional on-chip overhead. In *DAC '11*, pages 411–416. ACM, 2011.

[5] R. Desplats, F. Beaudoin, P. Perdu, N. Nataraj, T. Lundquist, and K. Shah. Fault localization using time resolved photon emission and stil waveforms. In *ITC'03*. IEEE Computer Society, 2003.

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

[7] H. F. Ko and N. Nicolici. Automated trace signals identification and state restoration for improving observability in post-silicon validation. In *DATE '08*, pages 1298–1303. ACM, 2008.

[8] S.-B. Park and S. Mitra. IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors. In *Design Automation Conference*. ACM, 2008.

[9] 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*, pages 1606–1621, 2005.

[10] V. C. Vimjam, E. Amyeen, R. Guo, S. Venkataraman, M. S. Hsiao, and K. Yang. Using scan-dump values to improve functional-diagnosis methodology. In *VTS'07*, pages 231–238, 2007.

[11] M. J. Y. Williams and J. B. Angell. Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic. *IEEE Transactions on Computers*, C-22(1):46–60, January 1973.

[12] 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*, pages 94–98. IEEE Computer Society, 2006.

[13] L. Zhang, I. Ghosh, and M. Hsiao. Efficient sequential atpg for functional rtl circuits. In *ITC'03*, pages 290–298, 2003.