Silicon Fault Diagnosis Using Sequence Interpolation with Backbones

Charlie Shucheng Zhu*, Georg Weissenbacher†, Sharad Malik*
*Princeton University, Princeton, NJ; †Vienna University of Technology, Vienna, Austria
shucheng@princeton.edu; georg.weissenbacher@tuwien.ac.at; sharad@princeton.edu

***Abstract*—Silicon fault diagnosis, the process of locating faults in a chip prototype, becomes more challenging and time-consuming with increasing design complexity. Consistency-based fault diagnosis aims at identifying fault candidates for an erroneous execution trace by symbolically checking the consistency between the golden gate-level model and the faulty behavior of the prototype chip. The scalability of this technique is limited to short executions due to the underlying decision procedure. This problem has previously been addressed by restricting the analysis to a window of fixed size and moving it along the execution trace. In this setting, limited observability results in a loss of precision and potentially missed fault candidates.**

We present a novel interpolation-based framework which formalizes the propagation of state information across sliding windows as a satisfiability problem. Our approach provides both spatial and temporal localization for general faults and is not restricted to a specific fault model. Further, our approach can be used to provide more accurate localization for a single permanent fault model. We experimentally demonstrate the efficacy and scalability of this approach by applying it to a variety of benchmarks from multiple suites (OpenCores, ITC99 and HWMCC).

I. Introduction

Silicon fault diagnosis aims at detecting and localizing a fault in a manufactured prototype chip, when the ***fault*** manifests as some observable ***error*** during a test. This diagnosis is based on a correct logic design referred to as the golden model. Silicon fault diagnosis is the counterpart of design debug or pre-silicon verification, whose goal is to find bugs in the logic design by using high level specification and simulation [20]. It also differs from manufacturing test, where the primary goal is to detect manufacturing defects on all fabricated chips without localizing the fault [15].

Silicon fault diagnosis can take advantage of executing long tests at speed. However, the limited observability in the post-silicon setting hinders accurate fault localization. Thus, it is also one of the most challenging problems in post-silicon debug [9]. With the aid of Design-for-Debug (DfD) techniques, such as scan chains [23] and trace buffers [1], we can increase observability in chips. Architecture-specific analyses such as IFRA [16] can narrow down the fault's location, but require considerable insight about the chip design. Fault diagnosis depending on fault models, e.g. [19], is limited to specific types of faults and has difficulty localizing general faults.

We focus on localizing *general* faults in *general* silicon circuits with limited observability. Our technique uses consistency based diagnosis [17], which has been successfully applied for fault diagnosis [20] as well as design debugging [18], [5]. This approach is based on a symbolic representation of a circuit's execution trace. The symbolic representation can be analyzed by formal tools, such as a SAT solver, to detect the inconsistency between the golden gate-level design and the behavior of its post-silicon prototype. Our algorithm further utilizes information gathered during execution to identify a set of ***fault candidates***, which are defined as the potential causes for the faulty behavior of the silicon chip. This technique enables our approach to handle silicon chips with either a ***single fault*** or ***multiple faults***. Further, our approach is not restricted to a particular fault model (e.g. stuck-at fault model) and is applicable to both ***permanent*** and ***transient*** faults.

One fault candidate is a particular gate/signal in a particular cycle that is possibly responsible for the observable error in a given execution trace. This gate/signal determines the ***spatial location*** of the fault candidate, while the cycle, in which this fault is excited, determines the ***temporal location***. The goal of fault diagnosis is to identify a small set of fault candidates.

Our framework comprises two phases. In the first phase (*inconsistency-detection phase or* ***ID phase***), our algorithm detects the inconsistency between a golden gate-level design and the observed behavior of the silicon prototype. In the second phase (*fault-localization phase or* ***FL phase***), we locate the fault candidates responsible for the inconsistency.

The scalability of this approach is limited by the underlying decision procedure. This problem can be addressed by partitioning the execution trace into sliding windows, which are then analyzed individually [10]. The bounded window size leads to information loss, which is remedied partially by propagating limited information across windows [25]. Two potential problems arise due to the incompleteness of the information. First, the sliding window approach may not detect any inconsistency in the ID phase when an error occurs. In this case additional debug information is needed to locate the bug (and can be obtained using techniques such as BackSpace [6], [7]). Second, the set of fault candidates returned by the FL phase may not contain the faulty gate that actually causes the erroneous behavior. This leads to *false negatives*, which is unacceptable since it misguides the debug process. In our work, we exclusively address the second problem. In other words, when an error is observed, we may not be able to detect an inconsistency in the first phase. However, once an inconsistency is detected, the fault candidates obtained by the second phase will include the actual fault causing the error.

In summary, our work makes the following contributions:

- We present an off-line approach that uses *Craig interpo-*

lation (See Section II-C) to successfully identify a small set of fault candidates in post-silicon sequential circuits with low observability.
- We prove that if an inconsistency is successfully detected in the first phase, the set of fault candidates returned by the second phase is guaranteed to contain the actual fault (if it is a single fault) or at least one of the actual faults (if there are multiple faults).
- Our methodology is not restricted to a particular fault model. An assumption for the fault's type is not necessary in our algorithm. The faulty chip can have unknown fault types, which can be either a permanent fault or a transient fault, as well as a single fault or multiple faults.
- For any type of fault, our approach can identify both spatial and temporal locations for the faults simultaneously. This is achieved by using sequence interpolation to construct a trace showing how a fault propagates and results in erroneous behavior.
- For a single permanent fault assumption, our framework can further reduce the set of fault candidates.
- Our algorithm significantly reduces the number of fault candidates. Experiments show that the spatial locations of general faults can be reduced by an average of 93.9% of the original circuit size. For single permanent faults, the reduction rate is increased to more than 99.4%.
- Our window-sliding approach is an iterative procedure, which is scalable to large circuits by making the size of the sliding window adjustable in our framework. The scalability of our approach is demonstrated by a variety of experiments. The largest sequential circuit tested has nearly one hundred thousand gates.

Below, we highlight our contributions relative to previous work in the field of fault diagnosis and fault localization:
- Interpolation is successfully used in model checking [14], and recently to propagate error information across sliding windows in the context of design debugging [10], where full observability is guaranteed. Our work, in contrast, targets silicon fault diagnosis with limited observability.
- In [20] and [21], Boolean satisfiability and unsatisfiable cores are applied to perform debugging. However, they do not partition the formula representing the execution, leading to scalability problems when the techniques are applied to fault diagnosis with long error traces.
- The approach presented in [25] identifies inconsistencies between a golden netlist design and the silicon behavior. It uses a MAX-SAT solver to identify a subset of the RTL design as the fault candidates. Although this subset is one possible explanation for the inconsistency, there is no guarantee that this subset includes the actual fault. In contrast, our approach guarantees to locate the actual fault (if it is a single fault) or at least one of the actual faults (if there are multiple faults).
- The BACKSPACE approach [7], [6] extracts error traces from repeated executions, but does not address fault localisation at the gate level. The approach in [19] based on BACKSPACE localizes faults with high accuracy, but depends on a given fault model. Without a fault model, e.g. stuck-at fault model in [19], the pre-image of a state cannot be computed. In contrast, our approach does not require a fault model. Also, our approach provides better scalability and is able to locate faults in 100 times larger sequential circuits.
- IFRA [16] is a processor specific technique focusing on electrical bugs, requiring detailed knowledge of the circuit under debug. Our approach, on the other hand, works for arbitrary circuits.

II. PRELIMINARIES

A. Notation and Basic Definitions

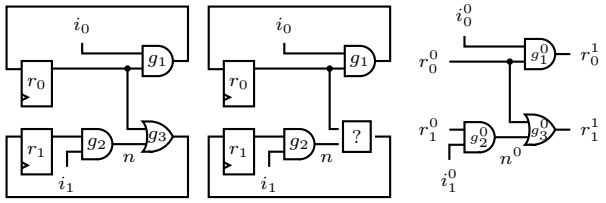
A *combinational Boolean circuit* C is a directed acyclic graph over the nodes N, with n input nodes $\{i_1, \ldots, i_n\} \subseteq N$ of in-degree zero, m output nodes $\{o_1, \ldots, o_m\} \subseteq N$ of out-degree zero, and a set of gates $\{g_1, \ldots, g_j\} \subseteq N$, each of which corresponds to one of the Boolean functions $+$ (OR), $\cdot$ (AND), and negation. C represents a Boolean function of type $\mathbb{B}^n \rightarrow \mathbb{B}^m$ mapping n input signals to m output signals. Our technique can handle arbitrary gate types. Simple gates are used here for ease of exposition.

In a *sequential circuit*, the outputs depend on both the input signals and the state represented by the latches $\{r_1, \ldots, r_l\} \subseteq N$ of in-degree one, whose outgoing edges and incoming edges are connected via a combinational circuit C. The function $C : \mathbb{B}^{n+l} \rightarrow \mathbb{B}^{m+l}$ maps the input signals and the state of the execution cycle t to outputs and a successor state of cycle t.

By replicating the combinational part C of a sequential circuit k times (where each instance C_t represents a cycle t) and connecting the outgoing edges of the latches in C_t with the respective incoming edges of the latches in C_{t+1} (for $1 \leq t < k$), we obtain an *iterative logic array* (ILA) of length k [2]. In the resulting combinational circuit, we use a superscript to indicate the cycle of a node, and define a ***surjective function*** ν which maps each i_i^t, o_i^t, r_i^t and g_i^t to the corresponding $i_i, o_i, r_i, g_i \in N$ in the original circuit C.

An (unquantified) *Boolean formula* is built from variables, operators ($+$, $\cdot$, $\Rightarrow$, $=$, negation), and parentheses. A *literal* is either a variable x or its negation $\overline{x}$. A *clause* is an OR of literals, and a *product term* is an AND of literals. A formula F is *satisfiable* if there exists an assignment of Boolean values to its variables such that F evaluates to 1, and *unsatisfiable* otherwise. Let F be a product of clauses. Any unsatisfiable subset of F's clauses forms an ***unsatisfiable core*** (or ***UNSAT core***), and a ***minimal*** unsatisfiable core if the set is minimal [12], [21]. Each assignment that satisfies a formula can be represented as a product term containing x if x maps to 1 and $\overline{x}$ if x maps to 0. The ***backbone*** of a satisfiable formula is the set of literals which evaluate to 1 in all satisfying assignments [25].

Every sequential circuit C corresponds to a Boolean relation $R : \mathbb{B}^{n+l} \times \mathbb{B}^{m+l}$ encoding the graph of the function $C : \mathbb{B}^{n+l} \rightarrow \mathbb{B}^{m+l}$. Each input, output, and latch in C corresponds to a Boolean variable, and each gate corresponds to a Boolean operation in R. By introducing fresh variables $n_1, \ldots, n_h$ that represent the output signals of gates, each gate (and circuit) can be represented as a product of clauses [22], which is called



(a) Golden model (b) Faulty circuit (c) ILA with $k = 1$

Fig. 1: Single-cycle diagnosis of a sequential circuit

conjunctive normal form or *CNF*. Let R be the encoding of the combinational part C of a sequential circuit. The following Boolean relation encodes the corresponding ILA of length k:

$$\prod_{t=0}^{k-1} R\left(\begin{array}{c} i_1^t,\ldots,i_n^t, r_1^t,\ldots,r_l^t, o_1^t,\ldots,o_m^t, \\ n_1^t,\ldots,n_h^t, r_1^{t+1},\ldots,r_l^{t+1} \end{array}\right) \quad (1)$$

We define a ***surjective function*** γ which maps a clause in Formula 1 to a particular gate in a particular cycle of the ILA. This function γ, when combined with function ν, maps clauses in a formula to gates in the original circuit.

A ***crash state*** is a state in the execution trace in which an error is observed (e.g. a system hang or property violation occurs). We assume that when a crash state is encountered, the chip's execution stops and signals in the ***scan chain*** (full state information from crash state) and ***trace buffers*** (partial state information for a number of cycles) are available [1].

B. Consistency Based Diagnosis

For a k-cycle execution on a sequential circuit, the observed outcome can be represented as a product term P over the variables of Formula 1. If the product of Formula 1 and P is unsatisfiable, there are two possible explanations: (a) P is the *expected* outcome according to the specification of the circuit, and Formula 1 is derived from a faulty implementation, and (b) Formula 1 encodes the circuit's golden model, and P represents a execution of a faulty silicon prototype. In fault diagnosis, we consider the second scenario [20].

Consistency based diagnosis [17] is an approach that identifies fault candidates which cause the unsatisfiability. Recent realizations of this technique are based on partial *maximum satisfiability* (MAX-SAT) solvers and *minimal correction sets* (MCS). Each MCS is an irreducible hitting set of the set of minimal unsatisfiable cores [12]. In other words, each clause of each minimal unsatisfiable core is part of some MCS. To avoid missing a fault, it is necessary to consider *all* MCSs [13], which is computationally expensive. By applying γ (see Section II-A) to these MCSs, we obtain the temporal and spatial location of the fault candidates in the ILA. This set of fault candidates covers *all* gates that contribute to a minimal unsatisfiable core. Therefore, our approach aims at finding small unsatisfiable cores rather than MCSs.

***Example** 1: Figure 1a shows a simple sequential circuit with two latches r_0 and r_1. Figure 1c depicts the corresponding single-cycle ILA, with the following Boolean encoding:*

$$\left(r_0^1 = i_0^0 \cdot r_0^0\right) \cdot \left(r_1^1 = n^0 + r_0^0\right) \cdot \left(n^0 = i_1^0 \cdot r_1^0\right) \quad (2)$$

To avoid clutter, we omit the encoding as CNF. In Figure 1b, gate g_3 has been replaced with a blackbox, which can represent any type of fault, including permanent and transient faults. As a consequence, the logic value in r_1^1 may be corrupted. Assume the corrupted value of r_1^1 becomes 1. This fact, as well as the initial state and the input values are encoded in the product term $\overline{r_0^0} \cdot r_1^0 \cdot \overline{i_0^0} \cdot \overline{i_1^0} \cdot \overline{r_0^1} \cdot r_1^1$. The product of (2) and this term is unsatisfiable, and

$$(r_1^1 = n^0 + r_0^0) \cdot (n^0 = i_1^0 \cdot r_1^0) \cdot (\overline{r_0^0}) \cdot (r_1^1) \cdot (\overline{i_1^0}) \quad (3)$$

the unique minimal unsatisfiable core. The product becomes satisfiable if either $(r_1^1 = n^0 + r_0^0)$ *or* $(n^0 = i_1^0 \cdot r_1^0)$ *is dropped, suggesting that either g_2 or g_3 is at fault in this single cycle execution. Note that this consistency based diagnosis is independent of a fault model and gives all possible fault candidates that may be responsible for the inconsistency between chip's design and behavior.*

C. Sequence Interpolation

Let A and B be a pair of Boolean formulas whose product is unsatisfiable. By *Craig's interpolation theorem*, there exists a Boolean formula I which is implied by A and inconsistent with B (i.e. $1 \cdot A \Rightarrow I$ and $I \cdot B \Rightarrow 0$), such that all variables that occur in I also occur in A as well as in B. Then I is an ***interpolant*** of (A, B). Interpolants are not unique. Let I_1 and I_2 be interpolants of (A, B). Then $(I_1 + I_2)$ as well as $(I_1 \cdot I_2)$ are interpolants of (A, B).

Given k formulas $A_{k-1}, \ldots, A_0$ ***whose product is unsatisfiable***, a ***sequence interpolant*** comprises formulas $I_k, \ldots, I_0$ such that $I_k = 1$, $I_{t+1} \cdot A_t \Rightarrow I_t$ for $0 \leq t < k$, and $I_0 = 0$. Moreover, all variables that occur in I_t occur in $A_{k-1}, \ldots, A_t$ as well as in $A_{t-1}, \ldots, A_0$ (for $1 \leq t < k$).

III. Sliding Windows and Fault Localization

A. Sliding Windows for Limited Observability

In Example 1, we assumed that the inputs, outputs, and latches are fully observable. In the post-silicon setting, however, we need to rely on trace buffers and scan chains, which are only able to record a fraction of the execution history, resulting in weaker constraints. Assume that the literal $\overline{r_0^0}$ is not observable in the silicon chip and thus dropped from the product term in Example 1. Since this literal is part of the minimal unsatisfiable core in Formula 3, the diagnosis fails.

This problem can be addressed by considering an ILA of greater length, as demonstrated in the following example.

***Example** 2: Consider an ILA of length four for the circuit in Figure 1a. The corresponding instance of Formula 1 is*

$$\prod_{t=0}^{3} \left((r_0^{t+1} = i_0^t \cdot r_0^t) \cdot (r_1^{t+1} = n^t + r_0^t) \cdot (n^t = i_1^t \cdot r_1^t)\right) \quad (4)$$

We assume that the final state $(\overline{r_0^{t+4}}) \cdot (r_1^{t+4})$ of the execution of the faulty circuit from Example 1 has been recorded using a

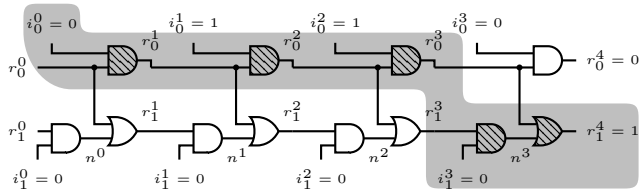


Fig. 2: Minimal unsatisfiable core of the four-cyle ILA

scan chain, and that the inputs $(\overline{i_0^0}) \cdot (\overline{i_1^0}) \cdot (i_0^1) \cdot (\overline{i_1^1}) \cdot (i_0^2) \cdot (\overline{i_1^2}) \cdot (\overline{i_0^3}) \cdot (\overline{i_1^3})$ *are known. The nodes derived from the corresponding minimal unsatisfiable core*

$$(r_0^1 = i_0^0 \cdot r_0^0) \cdot (r_0^2 = i_0^1 \cdot r_0^1) \cdot (r_0^3 = i_0^2 \cdot r_0^2) \cdot (r_1^4 = n^3 + r_0^3) \cdot (n^3 = i_1^3 \cdot r_1^3) \cdot (\overline{i_0^0}) \cdot (\overline{i_1^3}) \cdot (r_1^4)$$

*are marked in the ILA in Figure 2; each gate in the marked area constitutes a viable fault candidate, which refers to a particular gate (**spatial location**) in a particular cycle (**temporal location**).*

We use $B_t(i_1^t, \ldots, i_n^t, o_1^t, \ldots, o_m^t, r_1^{t+1}, \ldots, r_l^{t+1})$ to encode the partially observed signals of cycle t. The formula encoding the corresponding constrained ILA for a ***window*** starting from cycle t_0 and with length k is: $W_{t_0}^k =$

$$\prod_{t=t_0}^{t_0+k-1} \begin{array}{l} R\left(\begin{array}{c} i_1^t, \ldots, i_n^t, r_1^t, \ldots, r_l^t, o_1^t, \ldots, o_m^t, \\ n_1^t, \ldots, n_h^t, r_1^{t+1}, \ldots, r_l^{t+1} \end{array}\right) \\ \cdot \\ B_t(i_1^t, \ldots, i_n^t, o_1^t, \ldots, o_m^t, r_1^{t+1}, \ldots, r_l^{t+1}) \end{array} \quad (5)$$

assuming that the initial state $r_1^{t_0}, \ldots, r_l^{t_0}$ is unconstrained.

In the setting of Example 2, the corresponding instantiation is W_0^4. In that particular example, we need to consider an ILA of length at least four to locate the fault, since the instances W_{4-k}^k with $1 \leq k \leq 3$ are satisfiable.

In practice, the length k required to obtain an unsatisfiable core can be large, especially if the available state information is sparse. The maximum k we can consider is limited by the scalability of contemporary SAT solvers (e.g. [4]). This problem motivated the idea of moving a ***sliding window*** with fixed length backwards from a crash state along the execution trace. This is effectively splitting the ILA [10], [25]. Formally, for an execution trace of length c and a window of fixed length k, this amounts to analysing the instances W_{i-k}^k for $k \leq i \leq c$.

In general, this approach fails unless information is propagated between the windows. In Example 2, for instance, all formulas W_{i-2}^2, $i \in \{2, 3, 4\}$ are satisfiable (see Section IV). While the missing information may potentially be obtained by observing additional executions of the circuit [6], [7], we restrict ourselves to the constraints that can be derived *statically* from the windows W_{i-k}^k by symbolic reasoning. In the following, we use sequence interpolants to characterise the information required to find an unsatisfiable core.

B. Extracting the Unsatisfiable Core from Overlapping Windows Using Sequence Interpolant

Given an unsatisfiable formula W_0^c, which represents the entire ILA, let us consider a sequence of $c-k+1$ overlapping windows W_i^k for $0 \leq i \leq c-k$. Consecutive overlapping windows W_{i-1}^k and W_i^k share all the variables $i^t, o^t, n^t, r^t, r^{t+1}$ for $i \leq t \leq i+k-1$. Since the entire formula W_0^c is unsatisfiable, we can find a sequence interpolant, $I_{c-k+1}, \ldots, I_0$, for this window sequence $(W_{c-k}^k, \ldots, W_0^k)$. By the definition of sequence interpolant, for all $0 \leq i \leq c-k$, we have $I_{i+1} \cdot W_i^k \Rightarrow I_i$. This is equivalent to the expression that $I_{i+1} \cdot W_i^k \cdot \overline{I}_i$ is unsatisfiable for all $0 \leq i \leq c-k$. The following holds:

***Theorem** 1:* Let $\widehat{W}_i^k$ be a product of a subset of the clauses of W_i^k such that $I_{i+1} \cdot \widehat{W}_i^k \cdot \overline{I}_i$ is still unsatisfiable. Then $\prod_{i=0}^{c-k} \widehat{W}_i^k$ is an unsatisfiable core of W_0^c.

Proof: The unsatisfiable formulas give us $I_{i+1} \cdot \widehat{W}_i^k \Rightarrow I_i$ for $0 \leq i \leq c-k$. Recursively, we can obtain $I_{c-k+1} \cdot \prod_{i=0}^{c-k} \widehat{W}_i^k \Rightarrow I_0$. Since $I_{c-k+1} = 1$ and $I_0 = 0$ by definition, $\prod_{i=0}^{c-k} \widehat{W}_i^k$ is unsatisfiable. ■

Note that even if each $\widehat{W}_i^k$ is a minimal subset of clauses satisfying this property, the resulting core $\prod_{i=0}^{c-k} \widehat{W}_i^k$ is not necessarily a minimal unsatisfiable core of W_0^c. This is only guaranteed if $\overline{I}_i$ and I_{i+1} are the ***strongest*** formulas implied by the prefix and the suffix of the trace, respectively. Lemma 1 enables us to find ***smaller*** cores by strengthening individual $\overline{I}_i$:

***Lemma** 1:* Let $I_{c-k+1}, \ldots, I_i, \ldots I_0$ be a sequence interpolant as in Theorem 1. The unsatisfiable core of $I_i \cdot W_{i-1}^k \cdot \overline{I}_{i-1}$ must have the form $\widehat{I}_i \cdot \widehat{W}_{i-1}^k \cdot \widehat{\overline{I}}_{i-1}$, where $\widehat{I}_i$ is a product of a subset of the clauses of I_i. Then $I_{c-k+1}, \ldots, \widehat{I}_i, \ldots I_0$ is also a sequence interpolant.

Proof: Since $\widehat{I}_i \cdot \widehat{W}_{i-1}^k \cdot \overline{I}_{i-1}$ is unsatisfiable, we have $\widehat{I}_i \cdot W_{i-1}^k \Rightarrow I_{i-1}$. We can also derive $I_{i+1} \cdot W_i^k \Rightarrow I_i \wedge I_i \Rightarrow \widehat{I}_i$. So Lemma 1 holds because $\widehat{I}_i$ is weaker than I_i but still strong enough to entail the subsequent interpolant. ■

Assume we apply Lemma 1 iteratively starting from I_0 to I_{c-k+1}. Eventually, the sequence interpolant will be updated to $\widehat{I}_{c-k+1}, \ldots, \widehat{I}_i, \ldots \widehat{I}_0$. As suggested above, the benefit of using this updated interpolant is that it enables our algorithm to find a ***smaller*** unsatisfiable core. We might reach an $\widehat{I}_i$ $(i \leq c-k)$ which can be weakened to 1, and all subsequent cores $\widehat{W}_j^k$ $(i \leq j \leq c-k)$ are empty. Thus, the unsatisfiable core can be pruned to $\widehat{W}_{i-1}^k \cdot \widehat{W}_{i-2}^k \cdot \ldots \cdot \widehat{W}_0^k$. In general, given a sequence interpolant $I_{c-k}, \ldots, I_i, \ldots, I_j, \ldots, I_0$ with $I_i = 1$ and $I_j = 0$, we can prune the windows before I_i and after I_j.

Sliding windows are motivated by the fact that W_0^c is prohibitively large. Therefore, we cannot expect an analysis of the unsatisfiable core of W_0^c to be feasible. Thus, we propose to derive the large core $\prod_{i=0}^{c-k} \widehat{W}_i^k$ from individual formulas by using sequence interpolant.

The remaining problem is whether the unsatisfiable core $\prod_{i=0}^{c-k} \widehat{W}_i^k$ gives us a valid set of fault candidates, which should contain the actual fault in a circuit.

C. Identifying Fault Candidates from Unsatisfiability Cores

To prove that the unsatisfiable core $\prod_{i=0}^{c-k} \widehat{W}_i^k$ contains the actual fault, we first consider a single transient fault. This can be derived from the following theorem:

Theorem** 2:* Given a ***single transient fault, which is only stimulated in one cycle, there is a $\widehat{W}_i^k$ which contains the fault in that cycle.

Proof: $\prod_{i=0}^{c-k} \widehat{W}_i^k$ is a superset of a minimal unsatisfiable core U. The single fault in the particular cycle must be in U [21]. Thus, the fault is also in $\prod_{i=0}^{c-k} \widehat{W}_i^k$. ■

If there are ***multiple transient faults***, we might only identify a subset of these faults. The unsatisfiable core contains at least one of the transient faults at the cycle when it is stimulated. The reason is that $\prod_{i=0}^{c-k} \widehat{W}_i^k$ might not reflect all conflicts present in W_0^c [17], [21], [11]. For the permanent fault model, we only need to consider spatial localization. Given a ***single permanent fault*** (e.g. stuck-at fault), it can be regarded as multiple transient faults across the execution trace. Thus the faulty gate is included in the unsatisfiable core. For ***multiple permanent faults***, our approach can guarantee that at least one faulty gate is included in the core.

IV. Interpolants from Backbones

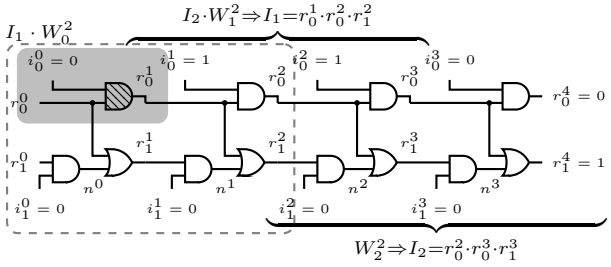
Section III proves that a sequence interpolant for sliding windows can be used to identify fault candidates. In this section, we show how the sequence interpolant can be constructed using backbones. Our algorithm is first illustrated in examples and then by the pseudo code in Algorithm 1.

Theorem 1 in Section III-B is based on a sequence interpolant for a partition of W_0^c. The construction of sequence interpolants is typically based on a refutation proof of W_0^c. Bayless *et al.* [3] recently presented an approach which enables processing the partitions individually by separate SAT solvers, but requires the windows to be repeatedly revisited to propagate conflict clauses. Both techniques are at odds with our assumption that W_0^c is prohibitively large.

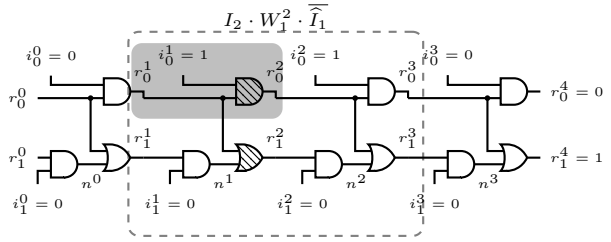
Zhu *et al.* [25] present preliminary work which uses propositional backbones to propagate information from the suffix to earlier cycles of W_0^c. Their work focuses on detecting unsatisfiability and extracting fault candidates from a single window. In the following, we show that backbones can be used to derive interpolants. Consequently, the key concepts presented in Section III can be applied to compute fault candidates for the entire execution trace.

Our algorithm is composed of two phases. In the first ***inconsistency-detection phase (ID phase)***, the sliding window moves from the crash state towards the beginning state (illustrated in Example 3). In the second ***fault-localization phase (FL phase)***, the sliding window moves in the reverse direction towards the crash state (illustrated in Example 4).

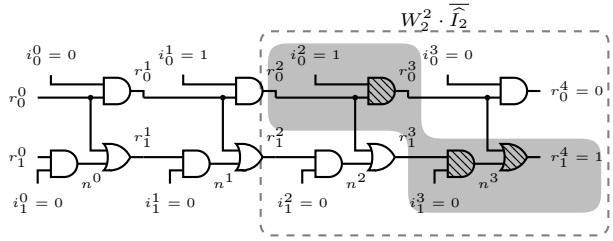
***Example** 3: We continue working in the setting of Example 2 in Figure 3a. As we are in the ID phase, the sliding window moves from right to left. Given the final state* $(\overline{r_0^{t+4}}) \cdot (r_1^{t+4})$ *and the input signals* $(i_0^2) \cdot (\overline{i_1^2}) \cdot (\overline{i_0^3}) \cdot (\overline{i_1^3})$ *for the window* W_2^2*, a SAT-based algorithm [8] yields the backbone* $I_2 \stackrel{\text{def}}{=} r_0^2 \cdot r_0^3 \cdot r_1^3$*. Similarly, from* $I_2 \cdot W_1^2$ *with inputs* $i_0^1 \cdot \overline{i_1^1} \cdot i_0^2 \cdot \overline{i_1^2}$ *we derive* $I_1 \stackrel{\text{def}}{=} r_0^1 \cdot r_0^2 \cdot r_1^2$*. Finally, we obtain the unsatisfiable instance* $I_1 \cdot W_0^2$*. This unsatisfiable instance indicates the inconsistency between the golden gate-level design and the faulty circuit's behavior.*



(a) Fault localisation with backbones



(b) Locating faults W_1^2 using I_1 and I_2



(c) Locating faults W_2^2 using I_2

Fig. 3: Fault localisation with sliding windows

It is easy to see that the backbone sequence $(1, I_2, I_1, 0)$ is exactly a sequence interpolant for (W_2^2, W_1^2, W_0^2) in Example 3. The backbones contain only literals that are shared between windows. Non-shared literals can be safely dropped, since they do not propagate information. Recall that the most exact information that can be propagated from W_i^k to W_{i-1}^k is the pre-image of $I_{i+1} \cdot W_i^k$. Backbones can be understood as an abstract domain of limited expressiveness.

Zhu *et al.*'s technique [25] stops after this ID phase and only derives a single faulty gate g_1 from $W_0^2 \cdot I_1$ and misses the actual faulty gate g_3. Thus, it fails to locate the general type fault introduced in Figure 3a, since their propagation technique ignores the fact that the fault may occur in window W_1^2 or W_2^2. The FL phase in our algorithm can rectify this by analyzing the remaining windows.

In our algorithm, we continue searching for fault candidates in W_1^2 and W_2^2, by moving the sliding window from left to right towards the crash state. Example 4 shows this FL phase:

***Example** 4: In the setting of Example 3, an unsatisfiable core is obtained in* W_0^2 *as shown in Figure 3a as the grey box. Since only* r_0^1 *of* I_1 *exists in this core, we can update* $I_1 = r_0^1 \cdot r_0^2 \cdot r_1^2$ *to* $\widehat{I_1} = r_0^1$ *as suggested in Lemma 1. We*

proceed to analyse $I_2 \cdot W_1^2 \cdot \overline{\widehat{I}_1}$ *, where* $I_2 = (r_0^2 \cdot r_0^3 \cdot r_1^3)$*. Intuitively, this query determines the gates in* W_1^2 *through which the corrupted value* $\overline{r_0^1}$ *can propagate. The resulting minimal core is indicated in Figure 3b as a grey box with one AND gate. Again, by applying Lemma 1,* I_2 *is updated to* $\widehat{I}_2 = r_0^2$*, since only* r_0^2 *exists in the core. The final window* W_2^2 *is inconsistent with* $\overline{\widehat{I}_2}$ *and further yields the core from* $\overline{\widehat{I}_2} \cdot W_2^2$ *in Figure 3c. Finally, the sequence interpolant is updated to* $(1, \widehat{I}_2, \widehat{I}_1, 0)$*. We can see that the three cores in Figure 3 can be linked together to form the same core illustrated in Figure 2. This is consistent with Theorem 1.*

The sequence interpolant algorithm is shown as the pseudo code in Algorithm 1. The entire flow contains both the ID phase (lines 7-21) and the FL phase (lines 22-35). When a faulty chip reaches a crash state, we are able to record a partial observable history leading to this crash state. The length of this observable trace is limited by the on-chip trace buffers' size and is defined as $traceLength$ in the algorithm. The input W^k is a vector of size-k windows, each of which contains observable information from the trace history and is defined as Formula 1. The global variables I and C are two vector structures for the sequence interpolants and unsatisfiable cores respectively. They are initially empty and their indices are indicated as subscripts.

In the ID phase, t decreases in each iteration. The $isSAT()$ subroutine checks whether window W_t^k, when constrained by I_{t+1}, is a satisfiable instance. Given that the instance is satisfiable, the subroutine $ComputeBackbone()$ returns the backbone I_t for the formula $I_{t+1} \cdot W_t^k$. The ID phase ends when an inconsistent window is encountered. Consequently, a 0 interpolant value is derived at the end. If the IL phase fails to detect any inconsistency within the trace of $traceLength$ cycles, it stops. The completeness of inconsistency detection is not guaranteed in this work, as clarified in Section I.

In the FL phase, t increases in each iteration. The subroutine $ComputeUnsatCore()$ computes the unsatisfiable core C_t for window W_t^k, under the constraint given by I_{t+1} and clause $\overline{I_t}$. The clause $\overline{I_t}$ is crucial for this routine because without it the instance will not be unsatisfiable. The $UpdateItps()$ subroutine updates the interpolant I_{t+1} by intersecting it with C_t. Note that if there is no intersection between them, I_{t+1} will be updated to 1, which indicates an end to the FL phase.

Finally, the $Merge$ subroutine is used to obtain the final unsatisfiable core (fault candidates) by linking unsatisfiable cores between window at cycles $flPhaseStart$ and $flPhaseEnd$. It also removes all the interpolants from the cores. When the algorithm finishes, both 0 and 1 are derived in the sequence interpolant. Thus, the length of the merged core is pruned to be only a fraction of the length $traceLength$, as discussed in Section III-B.

Algorithm 1 Unsatisfiable Core from Sequence Interpolants

Input: W^k: a vector of windows with size k.
Input: $traceLength$: The length of trace history
Output: FC: Final UNSAT core to return

```
1:  /* Global Variables */
2:  I: a vector of sequence interpolants, initially empty
3:  C: a vector of UNSAT cores, initially empty
4:  crashWindow ← traceLength − k
5:  t ← crashWindow
6:  /* ID phase starts here, with decreasing t. */
7:  I_t ← 1
8:  while true do
9:      if t < 0 then
10:         /* Inconsistency detection fails */
11:         return fail
12:     else if isSAT(W_t^k, I_{t+1}) then
13:         /* The window is SAT. Save its backbone */
14:         I_t ← ComputeBackbone(W_t^k, I_{t+1})
15:     else
16:         /* The window is UNSAT. Inconsistency found */
17:         I_t ← 0
18:         break
19:     end if
20:     t ← t − 1
21: end while
22: /* FL phase starts here, with increasing t */
23: flPhaseStart ← t
24: while t ≤ crashWindow do
25:     if I_t = 1 then
26:         break
27:     end if
28:     /* Obtain UNSAT core from interpolants */
29:     C_t ← ComputeCore(W_t^k, I_{t+1}, ¬I_t)
30:     /* update the interpolant */
31:     UpdateItps(I_{t+1}, C_t)
32:     t = t + 1
33: end while
34: flPhaseEnd ← t
35: /* Obtain fault candidates by linking the UNSAT cores */
36: FC = Merge(C, I, flPhaseStart, flPhaseEnd)
37: return FC
```

V. Experimental Evaluation

We selected 20 circuit benchmarks in AIG netlist format[1]:

- 2 circuits from *Opencores.org*[2]: *68HC05* and *8051* microcontroller.
- 5 circuits from *ITC'99* benchmarks[3]: *b12*, *b14*, *b15*, *b17*, *b22*. They are chosen among the 22 benchmarks due to their larger circuit size and AIG format availability.
- 13 other circuits from *HWMCC'10*[4] benchmarks. They include the largest circuits from *HWMCC'10* benchmarks, e.g. *bjrb07amba10andenv* and *neclaftp*, as well as a variety of other randomly chosen circuits to diversify our benchmark set.

The designs listed above constitute the golden models in our experiments. We evaluate our methodology by injecting stuck-at-faults into these designs to create 20 faulty netlists.

[1] http://fmv.jku.at/aiger/FORMAT
[2] http://opencores.org/projects
[3] http://www.cad.polito.it/downloads/tools/itc99.html
[4] http://fmv.jku.at/hwmcc10/benchmarks.html

We chose stuck-at faults because they are well understood and commonly used to evaluate fault diagnosis algorithms [19][20]. However, our methodology, which analyzes the consistency in a chip's execution trace, is not limited to permanent faults or any other specific fault model, as described in Section I.

We used *PicoSAT-v951* [4] as the underlying SAT solver for computing backbones, interpolants and unsatisfiable cores as explained in Section IV. When the resulting cores are merged, the resulting core provides both ***temporal*** as well as ***spatial*** locations for the fault candidates. The time-frames spanned by the core indicate possible temporal locations of the fault, while all gates covered by the core indicate the spatial locations.

The experiment is composed of two parts:

- In the first part, we make no assumption that we know the type of the fault. They maybe either transient or permanent, single or multiple. For such a general fault, our algorithm is able to report both the spatial and temporal location of the fault (for single faults) or at least one of the faults (for multiple faults). This diagnosis is used when the type of the fault is not available.
- In the second experiment, we presume knowledge of the fact that there exists a single permanent fault in the circuit. By this assumption, we are able to increase the accuracy of the spatial localization. The temporal information is irrelevant for permanent faults.

For the entire experiment, we fixed the following parameters: 1. Besides the full execution information obtained from inputs and outputs, 5% of the intermediate latches (randomly selected) as well as the full final state are observable (by means of trace buffers of 1000 cycles and a scan chain). 2. We fixed the sliding window size to 4 cycles. Optimizing the sliding window size [25] and trace buffer selection [24] can improve the efficacy of our work, but is orthogonal to our approach.

A. Localizing Fault Candidates without Assuming Fault Model

For each of the 20 faulty netlists, we did the following: 1. The circuit design is simulated with random inputs for 10 thousand cycles or until it crashes, whichever comes later; 2. In this simulation, several crash states can occur. One crash state is randomly selected from this error trace; 3. Starting from this random crash state, we run Algorithm 1.

Table I illustrates the spatial and temporal fault localization results for the general faults. Each row in the table indicates a different faulty benchmark circuit. The second column shows the number of gates of each netlist. The third column gives the number of suspected gates (spatial locations for fault candidates) identified by our algorithm. The numbers in parentheses are not relevant at this point and will be introduced later in Section V-B. Our algorithm is able to reduce the number of suspected gates dramatically, with an average reduction rate of 93.9%, as shown in column 4. While columns 3–4 give the spatial localization result, column 5 provides the temporal locations of fault candidates. Our algorithm identifies a small portion (4–24 cycles) of the entire execution trace as temporal fault locations. This is consistent with Section III-B, where it is proved that the sequence interpolant can be used to prune the error trace. We verified that the faulty gate is excited at least once within the identified cycles, which is proved in Section III-C. The last column lists the running time for localizing fault candidates on a workstation with an Intel Xeon 3.2 GHz CPU with 16 GB RAM. The running time and space of our algorithm is dominated by the window-size. Our algorithm can be scaled for large circuits with long error traces, as long as each window can be handled by the workstation. Furthermore, one may tailor the window size for individual circuits, although it is fixed to be 4 in our experiment.

TABLE I: Spatial and Temporal Fault Localization for General Faults

Circuit Name	No. of Gates	Suspected Faulty Gates	Reduction Rate (%)	Fault Cycles	Runtime
itc99_b12	1002	240	76.0	6	19s
viscoherencep3	1610	63	96.1	4	4s
bobsmi2c	1821	34	98.1	6	6s
eijkbs3271	2229	280	87.4	7	2m 13s
bj08vsar12	4266	148	96.5	24	2m 48s
itc99_b14	4749	533	88.8	5	3m 36s
68hc05	5991	119	98.0	6	1m 59s
pdtvisvsa16a12	6261	68	98.9	4	24s
pdtvisvsa16a27	6291	116	98.2	6	1m 13s
pdtswvsam4x8p0	6376	733	88.5	12	29m 30s
itc99_b15	9068	712	92.1	9	13m 42s
bobsynthetic	12541	142	98.9	4	4m
itc99_b22	19017	211	98.9	4	6m 12s
bobaesdinvdmit	21650	2842	86.9	4	133m 49s
8051	23600	1192	94.9	7	103m 51s
itc99_b17	28101	178 (81)	99.4	8	20m 59s
bobsmmem	30920	91 (9)	99.7	4	3m 30s
bjrb07amba6andenv	35272	5538 (203)	84.3	7	911m 5s
neclaftp1001	63383	274 (265)	99.6	4	65m 5s
bjrb07amba10andenv	98004	2372 (489)	97.6	9	829m 37s
AVG			93.9		

B. Localizing Fault Candidates for a Single Permanent Fault

Consider the case where we assume a single permanent fault. Multiple error traces can be used to further advantage in this context. Since the fault is permanent, it explains the faulty behavior for every error trace. By taking the intersection of the fault candidates from every trace, we can obtain a more accurate spatial localization result. While our methodology is applicable to both transient as well as permanent faults, we can provide more accurate spatial locations for permanent faults, since this scenario allows us to combine the information obtained from multiple test runs.

In our experiment, we picked the 5 largest circuits from the 20 benchmarks, which are injected with single permanent faults. They are diagnosed using six random error traces. Each error trace is analyzed by our algorithm to obtain a set of fault candidates, which we project to the original circuit using ν (See Section II-A). The assumption that there is a single permanent fault ensures that each resulting set of fault candidates contains the actual fault. Therefore, intersecting the sets of fault candidates results in a smaller set which is still guaranteed to contain the actual fault.

Figure 4 indicates the reduction rate of spatial fault candidates by analysing multiple traces. The number of fault candidates decreases as the number of error traces used for

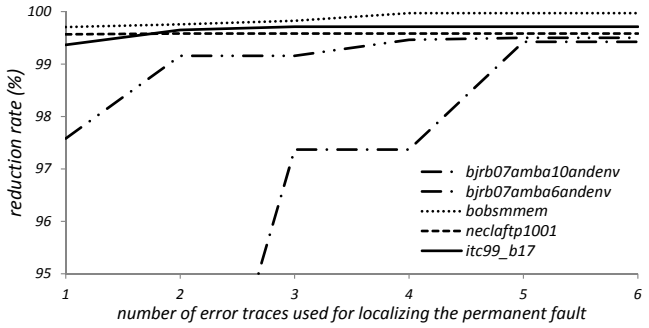


Fig. 4: Single Permanent Fault Localisation Using Multiple Test Traces

diagnosis increases. As we can expect, it gradually saturates. For the given five benchmarks, the final number of spatial fault candidates obtained by using 6 test traces are shown in the parentheses of the third column in Table I. More than 99.4% of gates are pruned from the spatial fault candidates.

In both parts of the experiment, we have verified that the injected faults are contained in the resulting set of fault candidates, as proved in Section III-C. Through the experiment, we have shown that the unsatisfiable core computed from sequence interpolants can correctly localize the actual fault.

VI. Conclusions

We propose a novel framework for fault diagnosis based on sequence interpolation. Our scalable approach iteratively computes unsatisfiable cores, which identifies both temporal and spatial locations for fault candidates in an execution trace. On the theoretical side, at least one of the actual faults is guaranteed to reside in the reported set of fault candidates. Our interpolation-based methodology finds an unsatisfiable core for a large unsatisfiable propositional formula by merging smaller cores derived from multiple windows, which significantly reduces the scalability requirements for the underlying decision procedure. On the practical side, the experimental results demonstrate that our approach can significantly narrow down the faults' potential location, which facilitates manual inspection. Our algorithm can be successfully applied to circuits with around 100 thousand gates and reduce the suspected faulty gates by an average of 93.9% without an assumption of the fault model. We instantiate our framework by using propositional backbones for computing interpolants. Since the expressiveness of backbones is limited, we intend to explore other means of obtaining interpolants as future work.

Acknowledgments: This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, and by the Austrian National Research Network S11403-N23 (RiSE) of the FWF and the Vienna Science and Technology Fund through grant VRG11-005.

References

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *DAC*, pages 7–12, 2006.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital systems testing and testable design.* Computer Science Press, 1990.

[3] S. Bayless, C. G. Val, T. Ball, H. Hoos, and A. J. Hu. Efficient modular SAT solving for IC3. In *FMCAD*. IEEE, 2013.

[4] A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[5] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris. Automated design debugging with maximum satisfiability. *TCAD*, 29(11):1804–1817, 2010.

[6] 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*, pages 1–10. IEEE, 2008.

[7] 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*, pages 411–416. ACM, 2011.

[8] M. Janota, I. Lynce, and J. Marques-Silva. Experimental analysis of backbone computation algorithms. In *Experimental Evaluation of Algorithms for solving problems with combinatorial explosion (RCRA)*, 2012.

[9] D. Josephson. The good, the bad, and the ugly of silicon debug. In *DAC*, pages 3–6. ACM, 2006.

[10] B. Keng, S. Safarpour, and A. Veneris. Bounded model debugging. *TCAD*, 29(11):1790–1803, Nov. 2010.

[11] B. Keng and A. Veneris. Managing complexity in design debugging with sequential abstraction and refinement. In *ASPDAC*, pages 479–484. IEEE, 2011.

[12] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[13] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In *IJCAI*, 2013.

[14] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[15] S. Mitra, S. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 12–17, June 2010.

[16] S.-B. Park, T. Hong, and S. Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *TCAD*, 28(10):1545–1558, Oct. 2009.

[17] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, Apr. 1987.

[18] S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. Sakallah. Improved design debugging using maximum satisfiability. In *FMCAD*, pages 13–19, 2007.

[19] 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*, pages 189–194. ACM, 2012.

[20] A. Smith, A. Veneris, M. F. Ali, and A. Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *TCAD*, 24:1606–1621, 2005.

[21] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Great Lakes symposium on VLSI*, pages 77–82. ACM, 2008.

[22] G. Tseitin. On the complexity of proofs in poropositional logics. In *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer, 1983.

[23] 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*, 22(1):46–60, Jan. 1973.

[24] J.-S. Yang and N. Touba. Automated selection of signals to observe for efficient silicon debug. In *VLSI Test Symposium, 2009. VTS '09. 27th IEEE*, pages 79–84, May 2009.

[25] C. Zhu, G. Weissenbacher, and S. Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *FMCAD*, pages 63–66, 2011.