SAT-based Techniques for Determining Backbones for Post-Silicon Fault Localisation

Charlie Shucheng Zhu Georg Weissenbacher Divjyot Sethi Sharad Malik
Department of Electrical Engineering, Princeton University

***Abstract*—The localisation of faults in integrated circuits is a dominating factor in the overall verification effort. The limited observability of internal signals of chips complicates the spatial and temporal localisation of bugs in post-silicon validation. We address the problem of recovering the values of unobservable signals of a chip prototype from state bits recorded in a trace-buffer of limited size using a SAT-based analysis.**

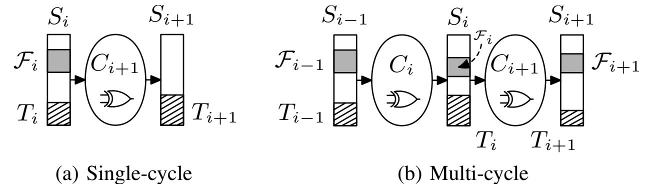
Our technique is a novel application of *backbones*. This term refers to the set of parameters of a Boolean function that need to be fixed to a constant value for that function to evaluate to true. There is a range of known SAT-based techniques targeting this problem. We discuss a number of existing techniques and gradually extend these techniques with novel ideas, leading to novel and previously unstudied algorithms.

We evaluate the performance of these algorithms using the aforementioned application in post-silicon validation. Our results show that these SAT-based techniques are suitable for large-scale applications with even millions of variables. Moreover, we evaluate the utility of backbones by quantifying the restored state bits in a number of case studies, including two processor cores.

I. Introduction

The localisation of faults in fabricated prototypes, referred to as *silicon debug* or *post-silicon validation*, is a challenging and time-consuming problem. While test cases can be run much more efficiently on a chip than by means of simulation, this advantage comes at the cost of limited observability of signals in integrated circuits. Logging techniques such as trace buffers enable us to track a relatively small number of signals over a limited amount of time (e.g., a few thousand execution cycles). This limited observability adds to the challenge of analysing erroneous behaviour of chip prototypes. We present a SAT-based analysis which enables the restoration of unobservable signals from state bits recorded in a trace buffer of limited size. Our analysis is based on computing the *backbone* [19] of a symbolic representation of the circuit. The problem is quite simply stated: given a Boolean representation $F(x_1, x_2, x_3, \ldots, x_n)$ of the circuit, determine the set $\mathcal{F}$ of *all* variables such that each $x_i \in \mathcal{F}$ takes the same constant value $c_i \in \{0, 1\}$ in all assignments under which F evaluates to 1, i.e., $F \cdot \overline{c_i}$ implies 0. Thus, for F to evaluate to 1, the assignments to the variables in $\mathcal{F}$ are *fixed*. For example, in the function $x_1 \cdot (x_2 + x_3)$, the variable x_1 is fixed to 1.

A number of SAT-based algorithms for computing backbones of formulae have been suggested (see [11], [25], [10], [17]). We discuss these algorithms in § II (and § IV) of our paper and gradually extend them with techniques such as learning, leading to novel and previously unstudied algorithms, which reduce the required number of calls to the SAT solver.



(a) Single-cycle (b) Multi-cycle

Fig. 1: Unwinding the combinational circuit C

We focus on the application of backbones for fault localisation in post-silicon validation. Given an execution of a chip prototype which exhibits unexpected behaviour, we want to localise the cycle and the gate which caused this error. We recently published a description of our methodology in [33], where we also present a case study demonstrating that backbones can aid fault localisation. In the following, we discuss the underlying algorithms and provide an empirical evaluation of the amount of information that can be recovered using backbones.

The specific methodology that we are relating to is based on storing a subset of the state bits of the chip prototype in a trace buffer and then using the stored values for localising the faulty cycle and the logic block [22], [21]. The setting is shown in Figure 1a: C_{i+1} represents the combinational part of the circuit under consideration. The state bits S_i and S_{i+1} correspond to the full system state at the end of the ith and $(i+1)^{th}$ cycles for the given error trace. The hatched sections show the subset of the system state that is stored in the trace buffers and is labelled T_i and T_{i+1}. The values of this fraction of the system state are known, whereas $S_i \setminus T_i$ and $S_{i+1} \setminus T_{i+1}$ is unknown. The error can be localised to cycle $i + 1$ if the "golden model" of C_{i+1} contradicts the observations T_i and T_{i+1}; various consistency-based analysis techniques have been proposed (see, for instance, [27]). These are not the focus of this paper. If, however, the error cannot be localised to cycle $i+1$, then the analysis proceeds backwards to cycle i. In doing so, it is helpful to determine whether, given the values of T_{i+1} and T_i, any of the assignments to the variables in $S_i \setminus T_i$ can be determined to be fixed: intuitively, the more state bits are known the higher is the chance to detect an inconsistency of the observed behaviour and the golden model. Let the set of the fixed variables thus determined be $\mathcal{F}_i$. This is shown as the shaded part of S_i in the figure. Thus, in the analysis of cycle i, we can use the values of $\mathcal{F}_i$ in addition to T_i and T_{i-1}. Furthermore, $\mathcal{F}_i$ can then be used in conjunction with T_i, T_{i-1}, and C_i to derive $\mathcal{F}_{i-1}$. We refer to the problem of

determining $\mathcal{F}_i$ for a given C_{i+1}, T_i, and T_{i+1} as the trace-buffer fixed-assignment problem.

The circuit C and the set of states S can be quite large in practice. Therefore, we need techniques that can handle functions with hundreds of thousands to millions of variables. Furthermore, as shown in Figure 1b, a multi-cycle version of the problem (in this case two cycles) can be used to determine the fixed assignments $\mathcal{F}_{i-1}$ and $\mathcal{F}_i$ using T_{i-1}, T_i, T_{i+1}, C_i, C_{i+1}, and $\mathcal{F}_{i+1}$. The set of fixed assignments over multiple cycles can be larger than in the one cycle case, as we have additional information from cycle i. The size of the functions grows quickly with unrolling over multiple cycles, further increasing the need for scalable algorithms.

We explore SAT-based techniques with the goal of leveraging the capabilities of modern SAT solvers. We show a range of techniques which use existing SAT solvers in different ways, and provide a theoretical analysis of the number of calls to the SAT solver for the different techniques. We also report our experimental results with using these techniques for the single and multi-cycle trace-buffer fixed assignment problem for processor cores used in the Backspace work [6] and for a set of circuits from the hardware model checking competition[1] (HMCC) benchmarks. These results show the practical value of the techniques and provide insights into their characteristics.

The paper is organised as follows. § II describes the various SAT-based techniques for the fixed-assignment problem and provides a theoretical analysis of the number of calls to a SAT solver for each of them. Our presentation covers existing techniques [11], [25], [10], [17] as well as novel algorithms. This is followed by § III, which covers the experimental evaluation of these techniques over a range of benchmark circuits. We address the performance of the algorithms in § II and address the utility of the approach in post-silicon validation by quantifying the information gained using backbones. § IV discusses related work for this problem, and finally § V provides some concluding remarks.

II. Determining Fixed Variables

Preliminaries and Problem Definition

Let $\mathcal{V}$ be a set of n propositional logic variables and let $\texttt{0}$ and $\texttt{1}$ denote the elements of the Boolean domain $\mathbb{B}$. Every Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ can be expressed as a propositional logic formula F in n variables $x_1, \ldots, x_n \in \mathcal{V}$. The logical connectives $\{-, +, \cdot, \rightarrow, \oplus\}$ are defined as usual. For brevity, we may omit $\cdot$ in conjunctions (e.g., $x_1\overline{x}_3$). An assignment $\mathcal{A}$ is a total mapping from $\mathcal{V}$ to $\mathbb{B}$, and $\mathcal{A}(x)$ refers to the value $\mathcal{A}$ assigns to x. $\mathcal{A}$ *satisfies* a formula $F(x_1, \ldots x_n)$ iff $F(\mathcal{A}(x_1), \ldots, \mathcal{A}(x_n))$ evaluates to $\texttt{1}$ (denoted by $\mathcal{A} \models F$). A formula F is satisfiable iff $\exists \mathcal{A} \,.\, \mathcal{A} \models F$, and unsatisfiable otherwise. We use $\#\mathcal{A}_F$ to denote the number of satisfying assignments of a formula F and drop the subscript if F is clear from the context. A formula F *holds* iff $\forall \mathcal{A} \,.\, \mathcal{A} \models F$.

We use $\texttt{Lit}_\mathcal{V} = \{x, \overline{x} \mid x \in \mathcal{V}\}$ to denote the set of literals over $\mathcal{V}$, where $\overline{x}$ is the negation of x. Given a literal $l \in \texttt{Lit}_\mathcal{V}$, we write $\mathrm{var}(l)$ to denote the variable occuring in l. A *cube* over $\mathcal{V}$ is a product of literals $l_1 \ldots l_m$ such that $l_i \in \texttt{Lit}_\mathcal{V}$ and $\mathrm{var}(l_i) \neq \mathrm{var}(l_j)$ for all $i, j \in \{1..m\}$ with $i \neq j$. We write $l \in C$ to indicate that the literal l occurs in a cube C. Given an assignment $\mathcal{A}$, we use $C_\mathcal{A}$ to denote the cube $\prod_{i=1}^{n} l_i$ where $l_i = x_i$ if $\mathcal{A}(x_i) = \texttt{1}$ and $l_i = \overline{x}_i$ otherwise. For a cube C over $\mathcal{V}$ and a set of variables $\mathcal{W} \subseteq \mathcal{V}$, let $C|_\mathcal{W}$ be the restriction of C to $\mathcal{W}$. That is, $C|_\mathcal{W} = \prod\{l \mid l \in C, \mathrm{var}(l) \in \mathcal{W}\}$.

Given a satisfiable formula F over $\mathcal{V}$, a variable $x_i \in \mathcal{V}$ is *fixed* in F if either $\mathcal{A}(x_i) = \texttt{1}$ for all $\mathcal{A} \models F$ or $\mathcal{A}(x_i) = \texttt{0}$ for all $\mathcal{A} \models F$, i.e., if either $(F \rightarrow x_i)$ or $(F \rightarrow \overline{x}_i)$ holds. Our goal is to determine the set $\mathcal{F} \subseteq \mathcal{V}$ of *all* fixed variables for F. We assume that F is a satisfiable formula over $\mathcal{V}$, else $\mathcal{F} = \emptyset$ holds trivially. In the formula

$$F = (x_1 \oplus x_2 \oplus x_3) \wedge \overline{(x_1 \oplus x_2)}\,, \qquad (1)$$

for instance, x_3 is fixed to $\texttt{1}$, while x_1 and x_2 are not fixed. We explore a number of increasingly sophisticated techniques to determine the set $\mathcal{F}$, culminating in the algorithm in § II.F which, to the best of our knowledge, is novel.

A. All-SAT

A first naïve attempt to determine the set $\mathcal{F} = \{x_3\}$ for Formula (1) is to enumerate *all* satisfying assignments $\{(x_1 \mapsto \texttt{0}, x_2 \mapsto \texttt{0}, x_3 \mapsto \texttt{1}), (x_1 \mapsto \texttt{1}, x_2 \mapsto \texttt{1}, x_3 \mapsto \texttt{1})\}$ of Formula (1). This technique, known as All-SAT [18], requires $\#\mathcal{A}$ iterations, which may be as many as 2^n. This approach has been evaluated in [17] and found unfit for large problem instances. The following sections describe our quest for a more practical solution leveraging efficient satisfiability solving techniques (c.f. [20] and [7]).

B. Probing

Given an initial satisfying assignment $\mathcal{A} \models F$, the set $\mathcal{F}$ can be determined by solving n independent SAT instances. In each of these instances, we constrain one variable x in F to take the value opposing $\mathcal{A}(x)$. The variable x is fixed iff $F \cdot (\mathcal{A}(x) \oplus x)$ is unsatisfiable. The following algorithm computes the set $\mathcal{F}$ by *probing* each variable independently:

1: $\mathcal{F} := \emptyset$
2: **let** $\mathcal{A}$ be such that $\mathcal{A} \models F$
3: **for all** $l \in C_\mathcal{A}$ **do**
4: **if** $\nexists \mathcal{A}' \,.\, \mathcal{A}' \models (F \cdot \bar{l})$ **then**
5: $\mathcal{F} := \mathcal{F} \cup \{\mathrm{var}(l)\}$
6: **end if**
7: **end for**

Contemporary SAT solvers support repeated incremental calls with differing assumptions (e.g., $\bar{l}$) about $\mathcal{V}$ in the form of cubes [7], which performs significantly better than restarts. The solver only discards the information inferred from $\bar{l}$ and retains the learnt clauses derived from F. *Our experiments (see § III) show that this is crucial to the feasibility of probing.*

Note that each iteration of line 4 potentially provides us with a new satisfying assignment $\mathcal{A}' \models F \cdot \bar{l}$. By construction, $\mathcal{A}'$ disagrees with $\mathcal{A}$ on the value of at least one variable.

[1] http://fmv.jku.at/hwmcc10/benchmarks.html

The assignments $\mathcal{A}$ and $\mathcal{A}'$, however, may differ in more than just one variable. The discrepancy between $\mathcal{A}$ and $\mathcal{A}'$ can be used to derive an (initially empty) set $\mathcal{N}$ of variables that are definitely *not* fixed. We call this technique *recording* and provide the following pseudo-code:

```
1: procedure RECORD(A, A', F, N)
2:     for all x ∈ (V \ (N ∪ F)) do
3:         if (A(x) ≠ A'(x)) then
4:             N := N ∪ {x}
5:         end if
6:     end for
7:     return N
8: end procedure
```

Recording can rapidly reduce the number of potentially fixed variables $\mathcal{P} = \mathcal{V} \setminus (\mathcal{N} \cup \mathcal{F})$. Iteratively computing the set $\mathcal{P}$ and skipping all literals $l \notin \mathcal{P}$ accordingly in line 3 of the probing algorithm results in a significant performance leap. For Formula (1), for instance, recording enables us to skip x_2 if we probe the variables in order of increasing index.

Probing requires $n = |\mathcal{V}|$ iterations (each of which involves a call to the SAT solver) in the worst case, but may terminate after $|\mathcal{F}|$ (or $n - |\mathcal{N}|$, respectively) iterations in the best case if paired with recording (where $\mathcal{F}$ and $\mathcal{N}$ represent to the final results of the algorithm).

Probing as well as recording have been introduced in [11].

C. All-SAT and Recording

The idea of recording discrepancies between two subsequent assignments also makes it worthwhile to revisit the enumeration-based approach previously dismissed as naïve.[2] The enumeration of assignments is typically implemented by successively blocking an increasing set of satisfying assignments of F (c.f. [18]). This approach is illustrated by the following algorithm:

```
1: N := ∅, F_0 := F, i := 0
2: let A be such that A ⊨ F_0
3: while ( ∃A' . A' ⊨ F_i · ¬C_A  and N ≠ V ) do
4:     F_{i+1} := F_i · ¬C_A
5:     N := RECORD(A, A', F, N)
6:     A := A', i := i + 1
7: end while
8: F := V \ N
```

In this case, modern SAT solvers enable the incremental construction of F_{i+1} without discarding *any* information previously learnt about F_i. The enumeration of the satisfying assignments terminates prematurely iff $\mathcal{N}$ becomes equal to $\mathcal{V}$: in the worst case, the algorithm enumerates $2^{n-1} + 1$ assignments before $\mathcal{N} = \mathcal{V}$. In all other cases, i.e., whenever $\mathcal{F} \neq \emptyset$, the algorithm still enumerates all satisfying assignments. Thus, the algorithm may require as few as two calls to the SAT solver if there are no fixed variables, but $\#\mathcal{A}$ iterations otherwise. Moreover, the size of F_i increases by n literals in each iteration, potentially resulting in a final F_i that is exponentially larger than F_0. The next section addresses the latter issue.

D. Decision-based All-SAT

Contemporary SAT solvers make a decision about the values of a subset $\mathcal{D}$ of $\mathcal{V}$ and propagate the implications of these decisions. In the terminology of [29], the set $\mathcal{D}$ is a *(weak) backdoor* for unit propagation. Setting x_1 to `0` in Formula (1), for instance, requires x_2 to be `0` and x_3 to be `1` to keep the output signal `1`. The SAT solver may be able to derive that $\overline{x}_1$ implies $\overline{x}_2 \wedge x_3$ and accordingly, the domain of a satisfying assignment $\mathcal{A} : \mathcal{V} \rightarrow \mathbb{B}$ can be partitioned into decision variables $\mathcal{D}$ and implied variables $\mathcal{V} \setminus \mathcal{D}$. We use $\mathcal{D}_\mathcal{A}$ to denote the decision variables associated with a specific assignment $\mathcal{A}$ and drop the subscript if $\mathcal{A}$ is clear from the context. The partial assignment to the decision variables is represented by the cube $C_\mathcal{A}|_\mathcal{D}$.

Note that $F \cdot (C_\mathcal{A}|_\mathcal{D}) \leftrightarrow C_\mathcal{A}$ as well as $C_\mathcal{A} \rightarrow C_\mathcal{A}|_\mathcal{D}$ must hold for all satisfying assignments $\mathcal{A} \models F$. The fact that $\overline{(C_\mathcal{A}|_\mathcal{D})} \rightarrow \overline{C}_\mathcal{A}$ and $F \cdot \overline{(C_\mathcal{A}|_\mathcal{D})} \leftrightarrow \overline{C_\mathcal{A}}$ hold immediately enables us to replace $\overline{C_\mathcal{A}}$ with $\overline{(C_\mathcal{A}|_\mathcal{D})}$ in the algorithm in § II.C. Blocking the partial assignment $\overline{x}_1$ readily eliminates the satisfying assignment $\overline{x}_1\overline{x}_2x_3$. The experiments in [29] suggest that in practical "structured" SAT instances $|\mathcal{D}|$ is much smaller than $|\mathcal{V}|$, resulting in smaller blocking clauses.

Observe, however, that $F \cdot (C_\mathcal{A}|_\mathcal{D})$ necessarily implies exactly one assignment $C_\mathcal{A}$ (since $C_\mathcal{A} \cdot C_{\mathcal{A}'}$ is unsatisfiable if $\mathcal{A} \neq \mathcal{A}'$). Accordingly, the decision variable-based algorithm, while generating smaller blocking clauses, still potentially eliminates satisfying assignments one by one and does not necessarily perform better than All-SAT (solving up to $\#\mathcal{A}$ SAT instances, unless there are no fixed variables). This observation is consistent with the result in [17], where the blocking clauses are minimised using variable lifting (see § IV). This worst case behavior can only be prevented by blocking more than one satisfying assignment per iteration. Such an attempt is presented in the following section.

E. All-SAT and Learning

The benefit of recording (see § II.B) is contingent on the values of $\mathcal{A}$ and $\mathcal{A}'$ on the variables in $\mathcal{P}$. Assignments $\mathcal{A}$ and $\mathcal{A}'$ that disagree exclusively on the values of $\mathcal{N}$ do not provide any additional information and can be safely ignored. *Accordingly, it is sufficient to block only the combination of values that $\mathcal{A}$ imposes on $\mathcal{P}$.* The following algorithm uses this idea to *learn* sets of assignments that can be safely eliminated:

```
1: N := ∅
2: let A be such that A ⊨ F
3: while ∃A' . A' ⊨ F · ¬(C_A|_P) do ▷ (note: P = V \ N)
4:     N := RECORD(A, A', F, N)
5:     A := A'
6: end while
7: F := P
```

[2] This combination of recording and All-SAT has not been reported in [11], [25], [10], [17].

Another benefit of this implementation is that it, unlike the algorithm in § II.C, discards the cubes blocking previous assignments, preventing the formula from growing: given two subsequently computed sets of potentially forced variables $\mathcal{P}$ and $\mathcal{P}'$ such that $\mathcal{P}' \subseteq \mathcal{P}$ holds, $\overline{(C_{\mathcal{A}}|_{\mathcal{P}'})}$ subsumes $\overline{(C_{\mathcal{A}}|_{\mathcal{P}})}$ since $(C_{\mathcal{A}}|_{\mathcal{P}}) \rightarrow (C_{\mathcal{A}}|_{\mathcal{P}'})$. Thus $\overline{(C_{\mathcal{A}}|_{\mathcal{P}})}$ can be dropped.

Note that each iteration of line 4 decreases the size of $\mathcal{P}$ (and increases the size of $\mathcal{N}$, respectively) by at least one, thus guaranteeing the progress of the algorithm in each iteration. Moreover, $F \cdot \overline{(C_{\mathcal{A}}|_{\mathcal{P}})}$ becomes unsatisfiable once $\mathcal{P} = \mathcal{F}$. Thus, the algorithm terminates after at most $|\mathcal{V} \setminus \mathcal{F}|$ iterations.

The following section presents an algorithm which draws together the ideas presented in § II.D and § II.E.

F. Decision-based All-SAT and Learning

When combining learning and blocking decision variables, we have to carefully navigate around the case in which $\mathcal{D} \subseteq \mathcal{P}$ does not hold. In this case, namely, blocking $C_{\mathcal{A}}|_{(\mathcal{D}\cap\mathcal{P})}$ potentially eliminates assignments still required to derive a correct solution, effectively "cornering" the algorithm in a dead end in which $F \cdot \overline{(C_{\mathcal{A}}|_{(\mathcal{D}\cap\mathcal{P})})}$ is unsatisfiable but only a fraction of the satisfying assignments has been taken into account.

What we can conclude from the unsatisfiability of $F \cdot \overline{(C_{\mathcal{A}}|_{(\mathcal{D}\cap\mathcal{P})})}$, however, is that all variables in $\mathcal{D} \cap \mathcal{P}$ are fixed. Unless $(\mathcal{D} \cap \mathcal{P}) = \emptyset$, this enables us to enlarge the set $\mathcal{F}$ of definitely fixed variables, as realised in the following implementation:

```
 1: F := ∅, N := ∅
 2: let A be such that A ⊨ F
 3: while (N ∪ F) ≠ V do
 4:     while true do
 5:         if (D_A ∩ P) ≠ ∅ then      ▷ (P = V \ (N ∪ F))
 6:             G := D ∩ P
 7:         else
 8:             G := P
 9:         end if
10:         if ∃A′ . A′ ⊨ F · ¬(C_A|_G) then
11:             N := RECORD(A, A′, F, N)
12:             A := A′
13:         else
14:             break                  ▷ (exit inner loop)
15:         end if
16:     end while
17:     F := F ∪ {var(l) | l ∈ (C_A|_G)}
18: end while
```

Note that $\mathcal{N} \cup \mathcal{F} = \mathcal{V}$ if $\mathcal{P} = \emptyset$, in which case the algorithm terminates.

The case in which $F \cdot \overline{(C_{\mathcal{A}}|_{(\mathcal{D}\cap\mathcal{P})})}$ becomes trivially unsatisfiable due to $\mathcal{D} \cap \mathcal{P}$ being empty is avoided by falling back on the solution presented in § II.E. Thus, termination is guaranteed since the algorithm expands either $\mathcal{F}$ or $\mathcal{N}$ in each iteration. However, unless the SAT solver selects $\mathcal{D}$ in a clairvoyant manner such that $\mathcal{F} \not\subseteq \mathcal{D}$ holds, the algorithm may potentially end up *probing* individual variables in $\mathcal{F}$ (by means of an unfavourable choice of $\mathcal{G}$ in line 6). Notably, the probing technique we started out with in § II.B can be derived from the algorithm above by replacing $\mathcal{G}$ with $\{x_i\}$ ($x_i \in \mathcal{P}$) in each iteration. Accordingly, the algorithm above also terminates after at most $|\mathcal{V}|$ iterations of the inner loop. Moreover, similarly to probing, the algorithm relies on incremental SAT to "retract" constraints $\overline{(C_{\mathcal{A}}|_{(\mathcal{D}\cap\mathcal{P})})}$ which made the instance unsatisfiable,[3] thus avoiding computationally expensive restarts of the SAT solver.

		worst case
A	All-SAT	$\#\mathcal{A}$
B	Probing	$\lvert\mathcal{V}\rvert$
C	All-SAT with Recording	$\#\mathcal{A}$
D	Decision-based All-SAT	$\#\mathcal{A}$
E	All-SAT and Learning	$\lvert\mathcal{V} \setminus \mathcal{F}\rvert$
F	Decision-based All-SAT + Learning	$\lvert\mathcal{V}\rvert$

TABLE I: Number of SAT calls for our techniques

We conclude the presentation of our techniques with an overview of their respective worst-case behavior. Table I lists the number of iterations (with respect to $\#\mathcal{A}$, $\mathcal{V}$, and the final value of $\mathcal{F}$) excluding the first call to the SAT solver to obtain an initial assignment. Note that the algorithms in § II.B, § II.E, and § II.F, which incorporate recording, can "get lucky" and terminate after only one iteration if there are no fixed variables. This case, however, is untypical. Further, recording does not help in § II.C and § II.D except in this very special case.

Nominally, the learning-based All-SAT technique is at an advantage. This, however, refers to the number calls to the SAT solver. In practice, the corresponding SAT instances may be of varying complexity across different examples. The experimental evaluation in the following section exposes this phenomenon further.

III. EXPERIMENTAL EVALUATION

This section evaluates the scalability as well as the utility of the techniques presented in § II. We implemented these techniques in our tool called JEDISAT (which determines variables with *force*d values). The implementation is based on version 2.0 of MINISAT [7]. JEDISAT uses the search algorithm of MINISAT 2.0, but does not exploit its simplification capabilities.

We consider the trace-buffer fixed assignment problem as outlined in § I (c.f. Fig. 1a). Our experiments (run on an Intel 4Core i7 @ 2.67GHz CPU with 3GB memory) for this comprise benchmarks from two different sources. We selected *(i)* the five largest circuits (in terms of file size) from the HMCC 2010 as well as *(ii)* the two processor cores from opencores.org (OC) that were used in the Backspace paper [6]. We converted the latter two Verilog designs into the DIMACS CNF format using the following flow of translation steps:

$$\text{Verilog} \xrightarrow{\text{Altera Quartus}^4} \texttt{blif} \xrightarrow{\text{ABC}^5} \texttt{aig} \xrightarrow{\text{AIGER}^6} \texttt{cnf}$$

[3] This is achieved by introducing a relaxation literal r_i in each iteration of the inner loop such that $(r_i \Rightarrow \overline{(C_{\mathcal{A}}|_{(\mathcal{G})})})$. Flipping r_i from 0 to 1 eliminates the constraint.

	Benchmark	CNF vars	Clauses	$\lvert I \rvert$	$\lvert S \rvert$
HMCC	bjrb07amba10andenv	98091	294015	23	63
	bjrb07amba9andenv	72952	218616	21	59
	mentorbm1and	36299	95182	224	4377
	neclaftp1001	71296	190305	32	7880
	neclaftp1002	71296	190305	32	7880
OC	68hc05	567	1610	11	127
	oc8051	26468	70857	83	2784

TABLE II: Size statistics

Table II lists the size of the CNF formulas resulting from the benchmarks, where $|S|$ denotes the number of latches and $|I|$ denotes the number of inputs of the design.

For each of these seven benchmarks, we create four different instances of the problem as follows. For each cycle of the hardware design (c.f. Fig. 1a), we constrain a certain percentage (1%, 5%, 10%, or 20%, respectively) of the latches and IO signals (chosen at random) to values determined by a prior simulation-run. The intention is to emulate the trace-buffers recording exactly this percentage of state variables. These correspond to the T_i for cycle i in § I. In practice, the state variables recorded in a trace-buffer will most certainly be chosen with more care, thus providing more useful information than a randomly chosen subset of variables. We compare the *performance* of the algorithms and evaluate the *utility* of our technique by quantifying the additional information gained with the help of backbones.

A. Performance

Fig. 2 shows the relative performance of the algorithms in § II on this set of benchmarks for the single cycle version of this problem (c.f. Fig. 1a) with a 100 second timeout for each instance. We omit the results for All-SAT without recording, since the results are identical to All-SAT with recording (as $\mathcal{F} \neq \emptyset$). In addition, to emphasize the benefits of incremental solving, we evaluated an implementation of probing which restarts the SAT solver in each iteration and drops all learnt clauses. The run-times are presented in a "cactus plot", indicating the number of instances that can be solved (x-axis) within a given *per-instance* time limit (y-axis). For each respective algorithm, the instances are ordered by increasing difficulty. As we can see, only algorithms B, E and F are competitive, while the others are only able to solve a few instances within 100 seconds. Further, algorithm B (probing) seems to be significantly better in overall runtime. This is somewhat surprising, since Table I suggests that algorithm E (All SAT and Learning) should perform better, as it has fewer iterations in the worst case.

Fig. 3 provides some additional insight into this. It provides the number of iterations, i.e., calls to the SAT solver (y-axis) for each of the seven benchmarks. For each benchmark, we evaluated the three most competitive algorithms (B, E, F) for 4 different sizes (1%, 5%, 10%, and 20% of $|I \cup S_i \cup S_{i+1}|$) of the trace buffers. Missing bars indicate that the technique did not finish within 100 seconds on the respective benchmark.

[4]altera.com [5]www.eecs.berkeley.edu/~alanmi/abc/ [6]fmv.jku.at/aiger/

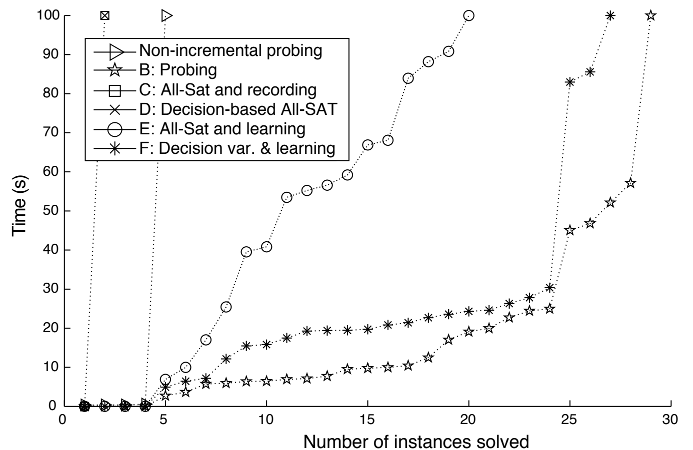


Fig. 2: Run-time of different techniques on single-cycle hardware competition and processor benchmarks

As expected, probing requires significantly more iterations. Moreover, the number of iterations of probing increases with the size of the trace-buffers, since larger trace-buffers in general result in more fixed variables. However, Fig. 2 shows that B is faster than the algorithms E and F, implying that the iterations of B are significantly faster on average. The reason for this is that probing a single literal strongly constrains the search space and enables efficient unit-propagation. We observed that probing performs better in particular for the first few iterations, i.e., probing gets a head start over the other approaches. The SAT-runs in later iterations provide an answer almost instantaneously, which we attribute to the fact that the SAT solver retains the information learnt in previous iterations. *We emphasise that this approach depends crucially on the capability of the SAT-solver to solve instances incrementally even if unit-clauses are discarded along the way (c.f. §§ II.B and II.F).* Experiments in which we deactivated this feature and performed a restart of the SAT solver instead (see non-incremental probing in Fig. 2) show that probing and decision-based All-SAT with recording quickly become computationally infeasible. Between E and F the results are somewhat mixed across the benchmarks.

B. Utility

Fig. 4 illustrates the fraction of unknown latch values determined to be fixed by using these techniques (naturally, B, E, and F yield the same results if they terminate and only differ in run-time). This corresponds to $|\mathcal{F}_i \cup \mathcal{F}_{i+1}|/|(S_i \setminus T_i) \cup (S_{i+1} \setminus T_{i+1})|$ in Fig. 1a. The x-axis lists the 7 benchmarks, with the 4 bars for each corresponding to increasing sizes of the trace buffers. With increasing sizes of the trace buffer, a larger fraction of the unknown state bits are determined to be fixed. An exception is the oc8051. The backbone size decreases for the largest trace buffer size due to the fact that the signals recorded in the trace buffer are chosen at random and vary across different trace buffer sizes.

Motivated by the application described in § I, we also explored the scalability and utility of the multi-cycle version

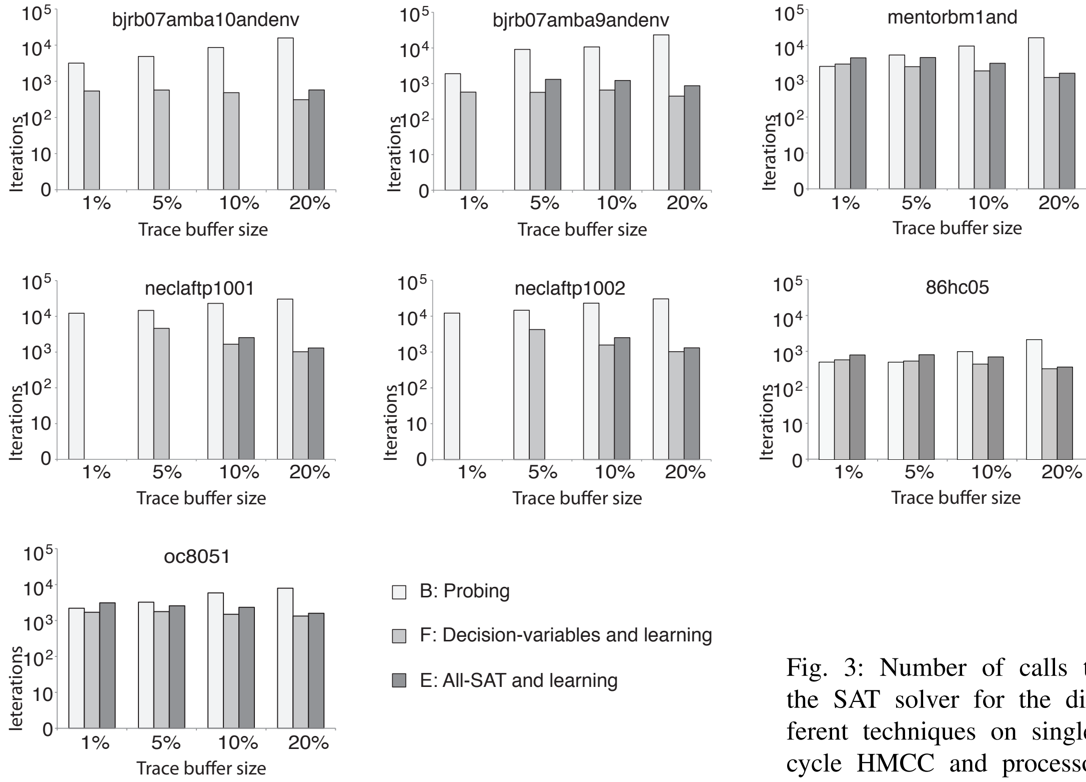


Fig. 3: Number of calls to the SAT solver for the different techniques on single-cycle HMCC and processor benchmarks

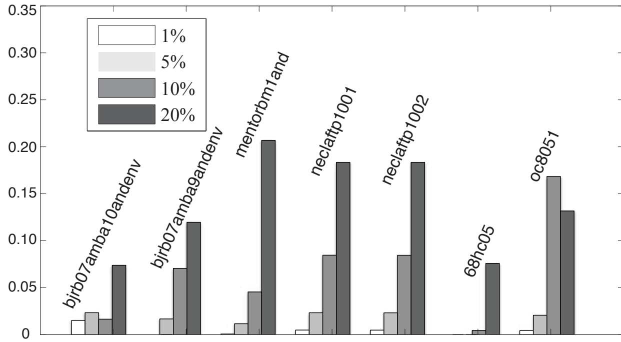
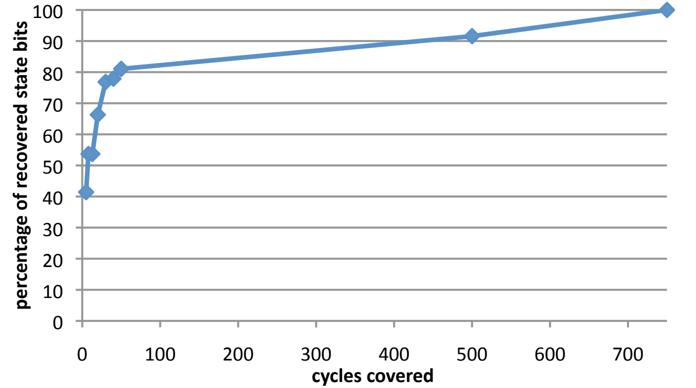
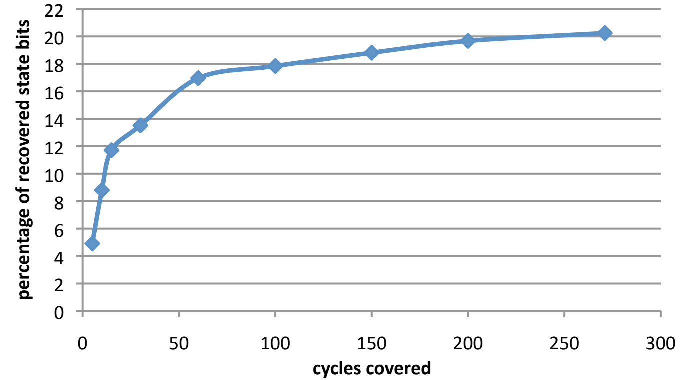


Fig. 4: $|\mathcal{F}_i \cup \mathcal{F}_{i+1}|/|(S_i \setminus T_i) \cup (S_{i+1} \setminus T_{i+1})|$ (c.f. Fig. 1a) for different trace-buffer sizes.

of the problem as illustrated in Figure 1b. Fig. 5 shows the percentage of the unknown state bits $\bigcup_{i=1}^{n}(S_i \setminus T_i)$ (as a function of the number of cycles that form the problem instance) that can be recovered (i.e., $\bigcup_{i=1}^{n} \mathcal{F}_i$) given that the content of 5% of the latches (randomly chosen) of the `oc8051` and the `68hc05` design is known for each of the cycles. In this scenario, we assume that all IO values are known. The timeout for the computation was set to one hour. Fig. 5b illustrates the scalability limits of the approach: within one hour, the analysis can cover at most 271 cycles of the slightly large `oc8051` processor, and 20.24% of the unknown state bits can be restored. For the smaller `68hc05`, we are able to analyse 3292 cycles within one hour, however, in this case, full information can be restored with just 750 cycles.



(a) `68hc05`



(b) `oc8051`

Fig. 5: Recovered latches for multiple cycles $\left(\frac{\sum_{i=1}^{n}|\mathcal{F}_i|}{\sum_{i=1}^{n}|S_i \setminus T_i|}\right)$

IV. Related Work

There is extensive literature on the computation and the applications of backbones. In the following, we give a brief overview covering the complexity of the problem and different algorithms to compute backbones. Finally we cover techniques for post-silicon validation that are related to our application.

A. Complexity of Computing Backbones

The term backbone was coined by researchers investigating the hardness of instances of NP-complete problems [19], [1], [12], [3], [31], [32]. The backbone represents the strongly constrained variables of a SAT instance and its size is one parameter studied in this context. The problem of computing backbones is *NP-equivalent* [13]. It is in the class of NP-hard problems and can be solved using polynomially many calls to a SAT oracle, i.e., it is NP-easy. Kilby *et al.* [14] shows that even the existence of a polynomial approximation algorithm for computing backbones would imply that $P = NP$. The notion of backbones has also been generalised to graph colouring problems [3].

B. Techniques for Computing Backbones

Climer *et al.* [2] presents a graph-based algorithm which relies on approximate lower and upper bounds to compute backbones of instances of the travelling salesman problem. The focus of our paper is on SAT-based techniques. A recent overview and experimental evaluation of such techniques is provided by Marques-Silva *et al.* [17]. In particular, this paper covers model enumeration (corresponding to the All-SAT algorithm in § II.A), *iterative SAT-testing* (equivalent to probing in § II.B), and filtering (which we call recording). Moreover, [17] discusses two optimisations (introduced in [23]) which we have not considered in our presentation. Firstly, *variable lifting* denotes the technique of discarding variables if they are not used for satisfying any clause. Secondly, a greedy approximation algorithm for the *set covering* problem, the problem of finding a minimal set of variables that satisfy all clauses, helps to eliminate variables in the probing algorithm.

Probing as well as recording were first developed as means to determine valid configurations of products. Kaiser and Küchlin [11], [25] partitions backbones into *inadmissible* (always false) and *necessary* (always true) variables and present three algorithms dubbed *basic*, *filter*, and *directed*. The former two algorithms correspond to probing with and without recording (see II.B), and the latter uses a variable selection and assignment strategy which aims at maximising the number of variables eliminated by recording. Janota [10] uses probing and a memoization technique similar to recording to find backbones.

The conclusions of [17] are similar to ours:

1) Backbone computation for large practical instances is feasible. Enumeration-based algorithms do not scale, neither do iterative algorithms that do not use the incremental interface of the SAT solver.
2) Backbones can represent a significant percentage of the number of variables (up to 90% and never below 10% in the benchmarks presented in [17], some of which are selected instances from the SAT 2005, 2007, and 2009 competition).

Gregory *et al.* [8] confirms that backbones can be as large as 86% of the variables in structured instances, but also shows that they are as small or non-existent in problems such as graph colouring. Hsu *et al.* [9] discusses approximation techniques to compute the *bias* of a variable, i.e., the proportion of solutions that assign a variable a particular value. This concept is more general than backbones.

The reduction of the size of blocking clauses in All-SAT based on conflict clauses is discussed in [18]. To the best of our knowledge, the algorithm we present in § II.F is novel.

C. Other SAT-based Techniques and Learning

Contemporary decision procedures are modern SAT solvers based on the DLL algorithm [4] using unit-propagation [5] for deriving implications. The implications determined in this context are not intended to be complete, but rather to help prune the search space [20]. In contrast, recursive learning is a complete algorithm for learning all variable implications [16]. There are some similarities between this technique and Stålmarck's algorithm for Boolean satisfiability, which can also provide a complete list of implications by recursively learning from setting all individual variables to both possible values, all pairs of variables to all four possible values and so on [24]. While both recursive learning as well as Stålmarck's algorithm are theoretically complete in the sense of potentially learning all implications, in practice they are only used in limited ways to improve on local implications to help prune large search spaces. In contrast, in this paper, we are interested in *complete* solutions to this problem in a large-scale setting (for functions of a hundred thousand or even millions of variables).

In his work on learning Boolean formulae, Valiant used the notion of queries to an oracle [28]. There is some similarity between that and the iterative use of a SAT-solver in our methods. However, Valiant's learning is limited to Boolean monomials (a conjunction of literals, or a Boolean cube), or Disjunctive Normal Forms (a set of cubes) and does not extend to learning fixed assignments.

D. Related Applications

The work most related to the application of backbones for post-silicon validation is [15]. This paper addresses the restoration of states by constructing a set of "restorability equations" that enable the propagation of known trace signals. Our approach does not require the construction of such equations. Moreover, our technique is complete in the sense that it will restore all signals covered by the backbone, whereas the recovery rate in [15] is limited by the fact that, for reasons of efficiency, no branching and backtracking is performed. Finally, our approach is SAT-based, allowing us to take advantage of the advances in satisfiability solving.

Yang *et al.* [30] proposes a SAT-based technique that, given a test scenario that results in a failure, identifies signals

that are relevant to the analysis of the failure and should therefore be recorded in (configurable) trace buffers. Our technique is orthogonal and would benefit significantly from systematically gathered information, which increases the size of the backbones.

De Paula *et al.* [6], have also used SAT for “backspacing” in post-silicon validation. However, their end-goal is different; they enumerate all possible previous states using a SAT solver and do not consider deriving the fixed assignments, which is the focus of our paper. Smith *et al.* [27] discusses a SAT-based technique to localise faults in post-silicon validation which does not take limited observability into account.

An overview of applications of backbones in optimisation problems and approximation algorithms, covering graph colouring, the travelling salesman problem, number partitioning, and planning, can be found in [26]. These applications exceed the scope of our paper.

V. Conclusion

This paper studies SAT-based techniques for determining the backbones of a Boolean formula with applications in post-silicon validation. We consider a set of known techniques as well as develop new techniques for this purpose. Overall these techniques scale up to realistic problem instances. We provide a number of case studies (including two processor cores) that demonstrate the utility of backbones in post-silicon validation.

Acknowledgements. We are indebted to our anonymous reviewers for their extremely helpful comments.

References

[1] D. Achlioptas, C. P. Gomes, H. A. Kautz, and B. Selman. Generating satisfiable problem instances. In *National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence*, pages 256–261. American Association for Artificial Intelligence, 2000.

[2] S. Climer and W. Zhang. Searching for backbones and fat: a limit-crossing approach with applications. In *Artificial Intelligence*, pages 707–712. American Association for Artificial Intelligence, 2002.

[3] J. Culberson and I. Gent. Frozen development in graph coloring. *Theoretical Computer Science*, 265:227–264, August 2001.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM (CACM)*, 5:394–397, July 1962.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7:201–215, July 1960.

[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 *Formal Methods in Computer-Aided Design (FMCAD)*, pages 5:1–5:10. IEEE, 2008.

[7] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[8] P. Gregory, M. Fox, and D. Long. A new empirical study of weak backdoors. In *Principles and Practice of Constraint Programming (CP)*, volume 5202 of *LNCS*, pages 618–623. Springer, 2008.

[9] E. I. Hsu, C. J. Muise, J. C. Beck, and S. A. Mcilraith. Probabilistically estimating backbones and variable bias: Experimental overview. In *Principles and Practice of Constraint Programming (CP)*, volume 5202 of *LNCS*, pages 613–617. Springer, 2008.

[10] M. Janota. Do SAT solvers make good configurators? In *Software Product Lines (SPLC)*, pages 191–195. Lero International Science Centre, University of Limerick, Ireland, 2008.

[11] A. Kaiser and W. Küchlin. Detecting inadmissible and necessary variables in large propositional formulae. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 96–102, 2001. (short paper).

[12] H. Kautz, Y. Ruan, D. Achlioptas, C. Gomes, B. Selman, and M. Stickel. Balance and filtering in structured satisfiable problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 351–358. Morgan Kaufmann, 2001.

[13] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *AAAI Conference on Artificial intelligence*, volume 3, pages 1368–1373. AAAI Press, 2005.

[14] P. Kilby, J. Slaney, and T. Walsh. The backbone of the travelling salesperson. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 175–180. Morgan Kaufmann, 2005.

[15] H. F. Ko and N. Nicolici. Automated trace signals identification and state restoration for improving observability in post-silicon validation. In *Design, Automation and Test in Europe (DATE)*, pages 1298–1303. ACM, 2008.

[16] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems-test, verification, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(9):1143–1158, Sept. 1994.

[17] J. Marques-Silva, M. Janota, and I. Lynce. On computing backbones of propositional theories. In *European Conference on Artificial Intelligence (ECAI)*, pages 15–20. IOS Press, 2010.

[18] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 250–264. Springer, 2002.

[19] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansk. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, July 1999.

[20] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[21] S.-B. Park, A. Bracy, H. Wang, and S. Mitra. BLoG: post-silicon bug localization in processors using bug localization graphs. In *Design Automation Conference (DAC)*, pages 368–373. ACM, 2010.

[22] S.-B. Park, T. Hong, and S. Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(10):1545–1558, 2009.

[23] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 31–45. Springer, 2004.

[24] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design (FMSD)*, 16:23–58, 2000.

[25] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17:75–97, January 2003.

[26] J. Slaney and T. Walsh. Backbones in optimization and approximation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 254–259. Morgan Kaufmann, 2001.

[27] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(10):1606–1621, 2005.

[28] L. G. Valiant. A theory of the learnable. *Communications of the ACM (CACM)*, 27:1134–1142, November 1984.

[29] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1173–1178. Morgan Kaufmann, 2003.

[30] Y.-S. Yang, B. Keng, N. Nicolici, A. G. Veneris, and S. Safarpour. Automated silicon debug data analysis techniques for a hardware data acquisition environment. In *International Symposium on Quality of Electronic Design*. IEEE, 2010.

[31] W. Zhang. Phase transitions and backbones of 3-SAT and maximum 3-SAT. In *International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2239 of *LNCS*, pages 153–167. Springer, 2001.

[32] W. Zhang. Phase transitions and backbones of the asymmetric traveling salesman problem. *Artificial Intelligence Research*, 21:471–497, April 2004.

[33] C. S. Zhu, G. Weissenbacher, and S. Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011.