Scaling VLSI Design Debugging with Interpolation

Brian Keng[1], Andreas Veneris[1,2]
[1]Department of Electrical and Computer Engineering, University of Toronto
[2]Department of Computer Science, University of Toronto
{briank, veneris}@eecg.toronto.edu

***Abstract*—Given an erroneous design, functional verification returns an error trace exhibiting a mismatch between the specification and the implementation of a design. Automated design debugging uses these error traces to identify potentially erroneous modules causing the error. With the increasing size and complexity of modern VLSI designs, error traces have become longer and harder to analyze. At the same time, design debugging has become one of the most resource-intensive steps in the chip design cycle. This work proposes a scalable SAT-based design debugging algorithm that uses interpolants to over-approximate sets of constraints that model the erroneous behavior. The algorithm partitions the original problem into a sequence of smaller subproblems by using subsections of the error trace that are examined iteratively. This is made possible by using interpolants to properly constrain the erroneous behavior for each subproblem, significantly reducing the number of simultaneous time-frames examined in the error trace. The described method is shown to be complete and an additional technique is presented to improve the quality of the debugging results using multiple interpolants. Experiments on real designs show a 57% reduction in memory and 23% decrease in run-time compared to previous work.**

I. Introduction

The aim of functional verification is to determine whether the implementation of a design conforms to its specification. If the design is found to be buggy, an *error trace* is returned which exposes the erroneous behavior. Due to the increasing size and complexity of modern designs, error traces generated from functional verification tools [1]–[5] can be thousands of clock cycles long [6]. This places a large burden on the engineer to examine the error trace and identify potential design bugs causing the erroneous behavior.

The task of identifying these potential error suspects in the design is called *design debugging*. This process begins after verification fails and an error trace is produced. An engineer then has to analyze this error trace, typically through a waveform viewer or some other graphical representation, a process today that is predominantly manual. This task can consume months of the verification effort and as much as 30% of the total time to design a Very Large Scale Integration (VLSI) chip [7]. Due to this fact, automated debugging methodologies remain of great interest to both the research and industrial communities.

Many automated debugging techniques have been developed to aid the engineer in this task. Simulation-based techniques [8], [9] have been extensively studied in the past and can be effective in certain situations. More recently, formal frameworks [10]–[15], such as those based on Boolean Satisfiability (SAT), have achieved significant advancements in design debugging.

Given a sequential design, SAT-based debugging techniques require a time-frame expansion of the circuit. This involves replicating the combinational component of the circuit such that the next-state variables of time-frame i are connected to the current-state variables of time-frame $i+1$ for the length of the error trace. For large designs and long error traces, this approach produces SAT instances which may not be practically viable because of the large memory footprint. Reducing the requirements for these techniques without sacrificing performance becomes an urgent necessity towards the development of scalable automated debugging tools.

In this work, we propose a novel scalable SAT-based design debugging algorithm which leverages interpolants to over-approximate sets of constraints that model the erroneous behavior, significantly reducing the memory-intensive circuit replication at any given time. This is accomplished by dividing the error trace into several parts, or *windows*, and analyzing each window of time-frames separately. To allow for each window to be properly constrained with the erroneous behavior, interpolants are used to over-approximate sets of constraints that model time-frames within close proximity to the observed error. The analysis begins with a window at the end of the error trace. If the analysis does not yield complete results, it proceeds by moving the window backwards iteratively. The interpolant is calculated from the unsatisfiable (UNSAT) core resulting from previously analyzed windows. The net result of this iterative methodology is a significant reduction in memory requirements and improvements in run-time.

The described method is shown to find all error locations whose functions can be modified to correct the erroneous behavior for a given error trace and number of errors. Additionally, a technique to generate multiple interpolants is introduced to reduce the number of error locations returned, thus improving the quality of the debugging results.

An extensive set of experiments on large hardware designs and long error traces illustrates the benefits of this work. It is shown that a conservative partitioning of the error trace yields an average 34% reduction in memory and 24% reduction in run-time compared to traditional SAT-based debugging, while the number of returned error locations is only increased on average by 1% of the total number of suspects. For a more aggressive partitioning scheme, averages of 57% reduction in memory and 23% reduction in run-time are achieved at the cost of increasing the relative number of error locations returned by 2%. This favorable trade-off between resolution and performance allows for scaling of existing SAT-based debugging methodologies to handle modern VLSI designs.

The remaining sections of the paper are organized as follows. Section II defines notation as well as background on debugging, UNSAT cores and interpolants. Section III illustrates the use of interpolants in partitioning the debugging problem. Section IV presents experimental results and

Section V concludes this work.

II. Preliminaries

A. Notation and Design Debugging

This section provides notation used throughout this paper and background information on design debugging.

The letters x, y and s refer to the primary inputs, primary outputs and state elements of a sequential circuit. x^i, y^i and s^i denote Boolean vectors in the i^{th} clock-cycle, or *time-frame*, of a sequential operation of a circuit. Similarly, x^i_j, y^i_j and s^i_j refer to the j^{th} indexed bit in the i^{th} Boolean vector. Finally, X^i, Y^i and S^i denote a predicate for the i^{th} clock cycle.

The behavior of a sequential circuit C can be described formally by a transition relation, $T(s^i, s^{i+1}, x^i, y^i)$, which is `true` if and only if given the current-state s^i, applying primary inputs x^i to C will generate primary outputs y^i and the next-state s^{i+1}.

Design debugging aims to find all error locations, or *suspects*, which could potentially explain the erroneous behavior demonstrated in a given error trace [9]. In this work, we define a design debugging method to be *complete* for a given error trace and number of errors, if and only if it returns all suspects whose functions can be modified separately to fix the erroneous behavior in the error trace. The *resolution* of a debugging method refers to the total number of suspects returned, where fewer suspects correspond to better resolution.

Formally, we define $\mathcal{V}^k_0$ of length k+1, as an error trace for clock-cycles 0 to k to consist of an initial state predicate, a vector of primary input predicates and a vector of *correct* or *expected* primary output predicates from 0 to k, which can be written as follows:

$$\mathcal{V}^k_0 = \langle S^0, \langle X^0, \ldots, X^k\rangle, \langle Y^0, \ldots, Y^k\rangle\rangle \qquad (1)$$

A *window* of an error trace from clock-cycles p to q, is defined as a consecutive subsequence of an error trace, $\mathcal{V}^q_p = \langle S^p, \langle X^p, \ldots, X^q\rangle, \langle Y^p, \ldots, Y^q\rangle\rangle$, where S^p is calculated by applying the initial state predicate and the first p primary input predicates to the transition relation, i.e. simulating the erroneous circuit for p cycles. Using this notation, a *prefix window* of length p for this trace can be written as $\mathcal{V}^{p-1}_0$ and a *suffix window* of length $k-p+1$ can be written as $\mathcal{V}^k_p$. We will occasionally omit the term window and use the term suffix or prefix in place of suffix window or prefix window respectively.

For this work, we assume that the error is first observed in the last clock cycle of the error trace. If this is not the case, a shorter error trace can be trivially generated by taking the shortest prefix that exhibits the erroneous behavior.

B. SAT-based Design Debugging

This section briefly describes background and notation for SAT-based design debugging that is relevant to our contribution. SAT-based design debugging [10] is a complete method that encodes the design debugging problem into a SAT instance for a given error trace and number of errors. The satisfying assignments of the SAT instance correspond to suspects which can be replaced with non-deterministic functions to correct the erroneous behavior in the error trace. The SAT instance is created in several steps. First, the transition relation is enhanced by introducing a set of *suspect variables*, $E = \{e_0, \ldots, e_n\}$, where each e_i corresponds to the i^{th} potential error location (gate, module etc.). The suspect variables are then added to the transition relation such that if $e_i = \mathtt{1}$ then the i^{th} potential error location is disconnected from its fan-in and become free variables. This can be achieved either through a hardware construction using multiplexors, or directly in conjunctive normal form (CNF). Note that each e_i can correspond to multiple gates depending on the type of the error location. The enhanced transition relation is denoted by $T_{en}(s^i, s^{i+1}, x^i, y^i, E)$.

Next, T_{en} is unrolled as a time-frame expanded model for the length of the error trace, such that the next-state of time-frame i is connected to the current-state of time-frame $i+1$. Note that the suspect variables are not replicated since they represent the same location regardless of the time-frame. The error trace predicates are then applied to the initial state, input and output variables of the replicated enhanced transition relation.

Finally, the number of simultaneous active suspect variables, denoted as the *error cardinality*, is constrained to a given constant N using cardinality constraints $\Phi_N(E)$ which can be generated from a network of adders [10]. Given an error trace $\mathcal{V}^k_0$ or a window of an error trace $\mathcal{V}^q_p$, design debugging can encoded by the following SAT problems respectively:

$$\begin{aligned} Debug^k_0 &= S^0(s^0) \wedge \Phi_N(E) \wedge \\ &\Big(\bigwedge_{i=0}^{k} X^i(x^i) \wedge Y^i(y^i) \wedge T_{en}(s^i, s^{i+1}, x^i, y^i, E)\Big) \\ Debug^q_p &= S^p(s^p) \wedge \Phi_N(E) \wedge \\ &\Big(\bigwedge_{i=p}^{q} X^i(x^i) \wedge Y^i(y^i) \wedge T_{en}(s^i, s^{i+1}, x^i, y^i, E)\Big) \end{aligned} \qquad (2)$$

Note that for $N = 0$, $Debug^k_0$ is UNSAT, since the error trace applied to the erroneous design without any active error suspect variables cannot produce the corrects outputs defined in the error trace.

In a satisfying assignment of Equation 2, each active suspect variable corresponds to a possible component (gate, module etc.) whose function can be changed to correct the erroneous behavior. To find all such suspects, for each satisfying assignment, a *blocking clause* is added to the debugging instance to block the active suspect variables from appearing again as a satisfying assignment. This instance is then sent again to the solver. When the solver eventually returns UNSAT, all possible suspects have been found.

Example 1 *Figure 1 shows a two time-frame expanded circuit of an erroneous two gate design with one state element. The suspect variables $\{e_1, e_2\}$ are denoted as enables on the side of each gate. The incorrect gate is g_2 which should be a buffer instead of an inverter. The error trace:*

$$\mathcal{V}^1_0 = \langle \overline{s^0_0}, \langle x^0_1 \wedge x^0_2, x^1_1 \wedge x^1_2\rangle, \langle y^1_1 \wedge y^1_2\rangle\rangle$$

demonstrates an erroneous behavior of the circuit. For $N = 1$, a satisfying assignment for the suspect variables $\{e_1, e_2\}$ is $\overline{e_1} \wedge e_2$. Adding the blocking clause $\overline{e_2}$ to the problem causes it to be UNSAT. This implies that g_2 is potentially the only gate that can be modified to correct the erroneous behavior.

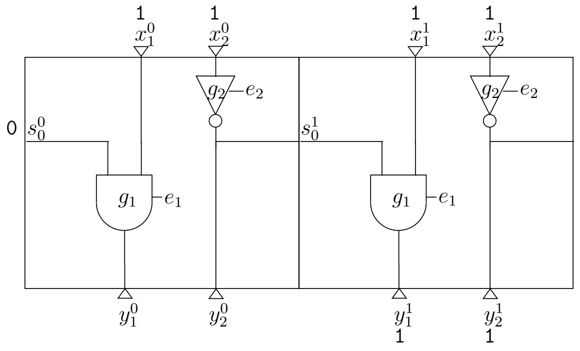


Fig. 1. SAT-based Debugging

C. *Unsatisfiable Cores and Interpolants*

An UNSAT core U is a subset of clauses that is unsatisfiable in an UNSAT propositional Boolean formula written in CNF. Modern DPLL [16] solvers can generate a proof of unsatisfiability along with a corresponding resolution graph that shows that a SAT instance is unsatisfiable [17]. The resolution graph demonstrates how clauses in the original SAT instance can be combined to generate the empty clause. The root nodes of the graph are the original clauses, the intermediate nodes correspond to the learned clauses and the leaf node is the empty clause.

An *interpolant* [18] is a Boolean formula that can be generated from an UNSAT core. For a given unsatisfiable formula whose clauses can be partitioned into two subsets, A and B, an interpolant is a formula, P, with the following properties:

(a) $A \rightarrow P$
(b) $B \wedge P$ is unsatisfiable.
(c) P only contains common variables of A and B.

There exists an algorithm [19] that can generate an interpolant as a Boolean circuit whose gates correspond to the vertices in the resolution graph and whose inputs correspond to the common variables. This algorithm takes $O(V+L)$ time, where V is the number of vertices in the resolution graph and L is the total number of literals in the proof. However, in the worst case, the size of the resolution graph can be exponential in the size of the problem.

III. Scalable Debugging with Interpolants

This section proposes a scalable SAT-based debugging algorithm that uses interpolants to reduce the number of simultaneous time-frames that need to be stored in memory. The algorithm analyzes windows of time-frames along the length of the error trace beginning with a suffix window and iteratively moving the window backwards until a prefix window of the error trace is analyzed. The interpolants are used to over-approximate constraints for a suffix of the error trace that is not modelled in the current window, ensuring that the erroneous behavior is properly constrained. Additionally, this method is shown to be complete and a technique using multiple interpolants is presented to improve its resolution.

In Section III-A and Section III-B, we show how to generate debugging instances for a suffix window and prefix window of an error trace. Using these two ideas, a complete scalable algorithm for debugging is described in Section III-C which partitions the original problem into smaller debugging instances. Finally, Section III-D shows how to improve resolution by using multiple interpolants.

A. *Suffix Window Debugging*

Debugging a suffix of an error trace can be achieved by applying the original SAT-based debugging scheme given in Equation 2. By using a suffix, only errors that are both excited within this window and propagate to primary outputs can be found. The following lemma describes a useful characteristic of suspects found in a suffix debugging instance.

Lemma 1 *Any suspect found in a debugging instance, $Debug_p^k$, for a suffix of an error trace, $\mathcal{V}_p^k$, will be found as a suspect to the debugging instance, $Debug_0^k$, for the entire error trace, $\mathcal{V}_0^k$.*

Proof: Let $M(E)$ be an assignment to the suspect variables in E such that $Debug_p^k \wedge M(E)$ is satisfiable. We wish to prove the lemma which can be written as:

$$Debug_p^k \wedge M(E) \text{ is SAT} \rightarrow Debug_0^k \wedge M(E) \text{ is SAT}$$

From Equation 2, we know that $Debug_0^{p-1} \wedge Debug_p^k$ and $Debug_0^k \wedge S^p(s^p)$ generate the same clauses. $Debug_0^{p-1}$ is SAT regardless of the error trace because the error has not been observed yet, so there is no mismatch in primary outputs. $Debug_0^{p-1} \wedge S^p(s^p)$ is SAT when no suspect variables are active because the instance $Debug_0^{p-1}$ amounts to simulating the circuit for the first p cycles of the error trace generating the same values as $S^p(s^p)$. Finally, $Debug_0^{p-1} \wedge S^p(s^p) \wedge M(E)$ is SAT because each active suspect variable allows the corresponding component to become an arbitrary non-deterministic function, which will not change the satisfiability of an instance if it was already satisfiable.

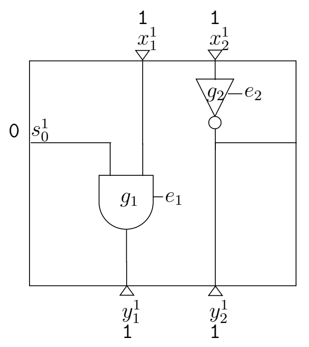
Therefore, if $Debug_p^k \wedge M(E)$ is SAT then $Debug_0^{p-1} \wedge Debug_p^k \wedge M(E)$ is SAT, since the only common variables are s^p and E which are fully assigned. As a result, $Debug_0^k \wedge S^p(s^p) \wedge M(E)$ is SAT implying that $Debug_0^k \wedge M(E)$ is SAT as required. ■

Lemma 1 guarantees that suspects found in the suffix are suspects that will be found in the entire error trace. However, if the error is excited before the current suffix, then there is no guarantee that the error will be found in $Debug_p^k$. Even though analyzing a suffix of an error trace may not result in a complete algorithm, valuable information can be extracted from the resulting UNSAT core as stated in the following theorem.

Theorem 1 *Let U be an UNSAT core generated after blocking all satisfying assignments to suspects for $Debug_p^k$. If $U \cap S^p(s^p) = \emptyset$ then the suspects found in $Debug_p^k$ will be exactly the suspects found in the entire debugging instance, $Debug_0^k$.*

Proof: From Lemma 1, any suspect found in $Debug_p^k$ is a suspect found in the entire debugging instance, $Debug_0^k$.

Now we prove by contradiction that any suspect found in $Debug_0^k$ will be found in $Debug_p^k$. Assume towards a contradiction that, $M(E)$ is an assignment to the suspect variables



$$(\overline{s_0^1})(x_1^1)(x_2^1)(y_1^1)(y_2^1)$$
$$(s_0^1+\overline{y_1^1}+e_1)(x_1^1+\overline{y_1^1}+e_1)$$
$$(\overline{s_0^1}+\overline{x_1^1}+y_1^1+e_1)$$
$$(x_2^1+y_2^1+e_2)(\overline{x_2^1}+\overline{y_2^1}+e_2)$$
$$(e_1+e_2)(\overline{e_1}+\overline{e_2})$$

Fig. 2. Suffix Window Debugging

such that $Debug_0^k \wedge M(E)$ is SAT and $Debug_p^k \wedge M(E)$ is UNSAT. And let U be the UNSAT core derived after blocking all satisfying assignments to suspects for $Debug_p^k$, which contains no clauses in $S^p(s^p)$.

Since $Debug_p^k \wedge M(E)$ is UNSAT, $M(E)$ is not blocked by any of the blocking clauses to $Debug_p^k$, which we denote by $blocking\_clauses_p^k$. This means that $Debug_0^k \wedge blocking\_clauses_p^k$ is satisfiable. However we know that in terms of clauses, $U \subseteq (Debug_p^k \wedge blocking\_clauses_p^k - S^p(s^p)) \subseteq Debug_0^k \wedge blocking\_clauses_p^k$, since U does not contain any clauses from $S^p(s^p)$. However, $Debug_0^k \wedge blocking\_clauses_p^k \wedge M(E)$ is satisfiable, so $U \wedge M(E)$ is satisfiable. But U is an UNSAT core, which is a contradiction. So it must be the case that $Debug_p^k \wedge M(E)$ is SAT. ■

Theorem 1 gives a condition to omit a prefix debugging analysis with very little additional computation beyond extracting the UNSAT core from the suffix debugging instance. Notice that the proof does not depend on the error cardinality since the same UNSAT core will exist in the suffix instance as well as the entire debugging instance. However, in the case where Theorem 1 is not valid, the prefix of the error trace must be analyzed to get a complete set of suspects. The following example illustrates a case where Theorem 1 can not be applied because there are clauses in the UNSAT core from the initial state predicate.

Example 2 *A suffix debugging instance derived from Example 1 is shown in Figure 2. The suffix, $\mathcal{V}_1^1$, is used to produce a suffix debugging instance $Debug_1^1$ with $N = 1$. The clauses for the suffix debugging instance are shown to the right of the circuit diagram. This instance is unsatisfiable. The following is an UNSAT core from the instance:*

$$(x_2^1)(y_1^1)(y_2^1)(\overline{s_0^1})(\overline{e_1}+\overline{e_2})(\overline{x_2^1}+\overline{y_2^1}+e_2)(s_0^1+\overline{y_1^1}+e_1)$$

Using Theorem 1, we know that $Debug_1^1$ does not result in the complete set of suspects to $Debug_0^1$ because the UNSAT core contains the clause $\overline{s_0^1} \subseteq S^1(s^1)$, so the prefix of the error trace still needs to be analyzed.

B. Prefix Window Debugging

Debugging a prefix of an error trace can be formulated in two parts. The first part uses the conventional SAT-based formulation (Equation 2) using a prefix of the error trace. The second part is an interpolant approximating time-frames for the corresponding suffix of the error trace.

Recall that the erroneous behavior is only observed in the last time-frame. If only a prefix of the error trace is modelled then the instance will not be properly constrained with the erroneous behavior. To avoid this situation, the interpolant is used as an over-approximation for the constraints that model the corresponding suffix. This ensures that the prefix debugging instance is properly constrained.

The interpolant can be generated by using an UNSAT core of the solved suffix debugging instance. To generate the interpolant, a partition of $Debug_p^k \wedge blocking\_clauses$ is defined by partitioning the clauses into two sets A and B. Set A represents the clauses modelling the enhanced transition function from p to k along with the primary input and output predicates from the error trace. Set B represents the initial state predicate, the error cardinality constraints and the blocking clauses. The clauses forming $Debug_p^k \wedge blocking\_clauses$ can be separated into A and B as follows:

$$\begin{aligned} A =& \bigwedge_{i=p}^{k} X^i(x_i) \wedge Y^i(y_i) \wedge T(s^i, s^{i+1}, x^i, y^i, E) \\ B =& S^p(s^p) \wedge \Phi_N(E) \wedge blocking\_clauses \end{aligned} \tag{3}$$

The common variables of A and B are the state variables s^p and the suspect variables E. Using this partition, an interpolant for the suffix, denoted P_p^k, can be generated from the resolution graph using the algorithm from [19].

P_p^k can be interpreted as an *over-approximation* of the suffix debugging instance. P_p^k will involve a subset of state and suspect variables that are directly related to the erroneous behavior observed at the primary outputs. The benefit of P_p^k is that it retains only the useful information that causes the erroneous behavior instead of modelling all the time-frames for the suffix of the error trace. In cases where the interpolant gets too large, the original clauses can be used in place of the interpolant, bounding the size of the constraints used to model the erroneous behavior. However experimental results show that in most cases, the interpolant is much smaller than the instance it was generated from, confirming the efficacy of using interpolants for debugging.

Example 3 *Figure 3 shows the resulting resolution graph on the left and interpolant on the right from the UNSAT core in Example 2. Notice how many of the root nodes of the resolution graph generate constants values in the interpolant. This is a common occurrence and generally leads to a small interpolant relative to the UNSAT core that it was derived from.*

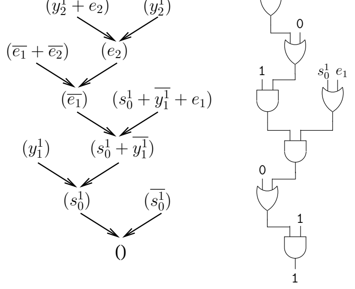


Fig. 3. Resolution graph and Interpolant

Using P_p^k, $Debug_0^{p-1}$ can be constrained with the cause for the erroneous behavior. The interpolant ensures that any suspect found in the prefix debugging instance resolves the erroneous behavior from the UNSAT core. The debugging instance for a prefix of an error trace with an interpolant, which we denote as $DebugItp_0^{p-1}$, can be written as follows:

$$DebugItp_0^{p-1} = Debug_0^{p-1} \wedge P_p^k \tag{4}$$

$DebugItp_0^{p-1}$ will be UNSAT when no suspect variables are active because $Debug_0^{p-1}$ will be equivalent to simulating the design for clock cycles 0 to $p-1$ and will implicitly generate the initial state predicate S^p which is known to be UNSAT with P_p^k. The next example builds from previous ones to show how a prefix debugging instance can be created.

Example 4 *Figure 4 shows how the interpolant generated in Example 3 can be used to debug a prefix of an error trace. Notice that the interpolant is significantly smaller once the constants have been propagated through the gates. In Figure 4, activating suspect variable, e_2, leads to the only satisfying assignment. This is consistent with the solution found in Example 1.*

The interpolant constrains the prefix debugging instance but it is an over-approximation. In other words, it will not miss suspects, as stated in the next theorem.

Theorem 2 *Any suspect found in $Debug_0^k$ will be found in $DebugItp_0^{p-1}$.*

Proof: By definition, $Debug_0^k = Debug_0^{p-1} \wedge A$, where A is defined in Equation 3. So any satisfying assignment to $Debug_0^k$ will satisfy $Debug_0^{p-1}$ and A. But $A \rightarrow P_p^k$, so it also satisfies P_p^k satisfying $DebugItp_0^{p-1}$. ∎

Theorem 2 guarantees that solving the prefix debugging instance will result in a complete method where no suspects will be missed. However, it does not guarantee that spurious suspects will not be found. P_p^k is used as an over-approximation for the suffix, so it does not provide as many constraints as explicitly modelling time-frames p to k. This results in $DebugItp_0^{p-1}$ possibly returning suspects that will not be found when debugging the entire error trace. Section III-D aims to reduce these extra suspects and improve the resolution.

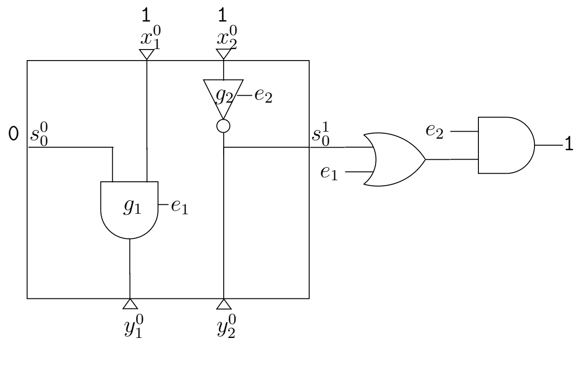


Fig. 4. Prefix Window Debugging with an Interpolant

Algorithm 1 Debugging with Interpolants

```
1:  $step$ := maximum number of time-frames
2:  procedure DEBUGINTERPOLANT($step$)
3:      $N$ := error cardinality
4:      $E$ := set of potential suspect variables
5:      $k$ := length of error trace
6:      $solutions$ := suspects found by algorithm
7:      $solutions \leftarrow \emptyset$, $P \leftarrow 1$
8:      while $k >= 0$ do
9:          $p \leftarrow max(k - step, 0)$
10:         $inst \leftarrow Debug_p^{k-1}(N, E) \wedge P$
11:         $solutions \leftarrow solutions \cup$ SOLVEALL($inst$)
12:         $U \leftarrow$ EXTRACTUNSATCORE($inst$)
13:         if $U \cap S^p(s^p) = \emptyset$ then
14:             return $solutions$
15:         end if
16:         $P \leftarrow$ GENERATEINTERPOLANT($U$)
17:         $E \leftarrow E$ - $solutions$
18:         $k \leftarrow k$ - $step$
19:     end while
20:     return $solutions$
21: end procedure
```

C. Scalable Debugging Algorithm

By using suffix and prefix debugging instances, it is possible to further divide the debugging problem into smaller windows that model no more than a user-defined number of time-frames. Algorithm 1 presents pseudo-code for a scalable debugging algorithm that divides an error trace of length k into $\lceil k/step \rceil$ windows, where $step$ is a user-defined parameter that specifies the maximum number of simultaneous time-frames to be modelled.

The algorithm iteratively analyzes windows of the error trace, starting with a suffix (lines 8-19). In each iteration, it begins by analyzing the current window of the error trace and finds all suspects shown on line 11. It then proceeds to generate an UNSAT core from the same instance and checks whether it contains any variables corresponding to the initial state predicate for the current instance. If it does not have any, it returns the current set of suspects. This is shown on lines 12-15. This condition allows for an early exit from the algorithm which in Theorem 1 was shown to be complete. If an early exit is not taken, it proceeds to generate an interpolant from the UNSAT core. Finally, it removes any suspects found in this iteration for consideration in the next iteration of the loop, pruning the search space for future iterations.

By iteratively analyzing consecutive windows of an error trace, the peak memory usage will be dramatically lowered with potential improvements in run-time. Even though the algorithm divides the error trace beyond just a suffix and prefix, it still guarantees completeness. This can be seen by analyzing each iteration of the loop. In the first iteration of the loop, if an early exit is taken (line 14), only a suffix debugging instance is run and Theorem 1 can be applied. After the first iteration of the loop for any given $step$, the prefix $\mathcal{V}_0^{k-step-1}$ of the error trace needs to be analyzed. This can be analyzed by using the debugging instance $DebugItp_0^{k-step-1}$ which is complete from Theorem 2.

However, instead of analyzing it directly, we can analyze a suffix of it by treating the interpolant as a constraint on the last time-frame. This is equivalent to another iteration of the loop. Using induction, this can be extended to the entire error trace and we can conclude that the last iteration of the loop will be a prefix debugging instance, $DebugItp_0^{k^*}$, where $V_0^{k^*}$ is the window used in the last iteration of the loop. By Theorem 2, this results in a complete algorithm.

Although Algorithm 1 is complete, there is a trade-off between the $step$ parameter and the final resolution. Each successive interpolant generated will potentially be a weaker constraint than the previous one. By setting $step$ to a small value, too many suspects can be returned. One way to cope with this is to provide a ranking of the suspects to the user so they can concentrate their effort on the most likely suspect. Algorithm 1 implicitly gives a useful ranking of suspects. More confidence can be given to suspects found in earlier iterations because a stronger constraint is used for the approximation of the suffix. In the case of the first iteration, all suspects found in the suffix will be found when debugging the entire error trace, as stated in Lemma 1.

D. Improving Resolution using Multiple Interpolants

The resolution of using this debugging method can be improved by using multiple UNSAT cores to generate multiple interpolants. Algorithm 1 guarantees completeness but may result in too many suspects if parameter $step$ is too small. This is due to the interpolant being an approximation to sets of constraints modelling the erroneous behavior. However by using multiple interpolants, the approximation will more closely match the original constraints potentially reducing the number of suspects that are found.

Using this fact, Algorithm 1 can be improved (line 12) by extracting multiple UNSAT cores to generate multiple interpolants. However, extracting multiple UNSAT cores can be an expensive process in general [20]. Algorithm 2 presents pseudo-code for a fast procedure for finding multiple UNSAT cores specifically for use in Algorithm 1.

The algorithm begins with an UNSAT instance and finds an UNSAT core (line 4). If the UNSAT core doesn't contain any clauses involving the initial state predicate, it exits and returns all UNSAT cores found so far (line 6-8). Otherwise, it randomly removes a subset of clauses from the initial state predicate that were involved in the current UNSAT core and is sent to the SAT solver again (line 10). This process is repeated until the instance is found to be satisfiable.

The size of the subset of initial state predicate clauses removed is a parameter to the algorithm. A smaller subset will leave more constraints in the problem having a higher chance of generating another UNSAT core but potentially taking more time and memory. By limiting the size of the subset and the number of cores found, the user can effectively trade-off run-time and memory for improved resolution.

Algorithm 2 Extracting multiple UNSAT cores

```
1:  procedure EXTRACTMULTIPLECORES($instance$)
2:      Cores ← $\emptyset$
3:      while $instance$ is UNSAT do
4:          $U \leftarrow$ EXTRACTCORE($instance$)
5:          CORES ← CORES $\cup \{U\}$
6:          IF $U \cap S^p(s^p) = \emptyset$ THEN
7:              RETURN CORES
8:          END IF
9:          $to\_remove \leftarrow$ SELECT_CLAUSES($S^p(s^p) \cap U$)
10:         $instance \leftarrow instance - to\_remove$
11:     END WHILE
12:     RETURN CORES
13: END PROCEDURE
```

IV. Experiments

This section presents experimental results for the proposed scalable SAT-based debugging algorithm as well as the algorithm to generate multiple interpolants. The results are compared to the SAT-based debugging work in [10] for the entire error trace which we will denote as *orig* in this section. MINISAT-V1.14 [21] with proof logging is used to solve the SAT instances and as well as generate the UNSAT cores. Experiments were run on a Pentium Core 2, 2.4 GHz workstation with 8GB of memory with a timeout of 7200 seconds.

We show the effectiveness of our algorithm on large designs from OpenCores.org [22]. Instances are generated by inserting a common RTL error such as a wrong assignment, missing case statement or incorrect operator. The error trace for each instance is generated by simulating the erroneous circuit through its testbench. Each suspect corresponds to a location in the RTL that can be corrected to satisfy the error trace.

Table I presents the results for the proposed debugging algorithm with interpolants. Four different sets of experiments are shown in this table. The first set of experiments in columns 5-7 correspond to running SAT-based debugging on the entire error trace (orig). The other three sets of experiments in column 8-16 correspond to debugging with interpolants varying the number of iterations ($r = \lceil k/step \rceil$) of the loop in Algorithm 1, ranging from 2 to 4. Each run uses one interpolant.

The first four columns in Table I show the instance name, number of clock cycles in the error trace, the gate count of the design and the total number of potential suspects. The next 12 columns show the run-time, peak memory and number of suspects returned for the four sets of experiments. For run-time and peak memory, the column with the lowest value is emphasized in bold.

For $r = 2$, the proposed algorithm shows on average a 24% decrease in run-time and 34% decrease in peak memory compared to orig, while increasing the number of suspects returned relative the total number of suspects on average by only 1%. With $r = 3$, the decrease in run-time is 26%, peak memory 48% and relative increase in suspects is 3%. $r = 4$ shows a similar trend by decreasing run-time by 23%, peak memory by 57%, but the relative increase in suspects is only 2% on average.

Figure 5 plots the run-time results from Table I from two different views. Figure 5(a) shows performance results of debugging with interpolants against orig on a log-log scale. Most points lie below the 45 degree line indicating faster runs on average. However, for several instances orig runs faster. In addition, `fdct1` and `fdct2` timed-out with orig while

TABLE I
DEBUGGING WITH INTERPOLANTS RESULTS

Instance Info				**Orig**			**Interpolant, r=2**			**Interpolant, r=3**			**Interpolant, r=4**		
instance	# cycles	# gates	total suspects	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols
ac971	675	25,314	1,086	588	6,040	34	357	2,842	34	253	2,022	34	**222**	**1,398**	34
ac972	300	25,314	1,086	314	2,674	41	133	1,187	41	112	838	47	**95**	**682**	47
divider1	40	5,799	1,092	10	180	32	**6**	127	41	6	110	41	6	**97**	41
divider2	40	5,799	1,092	5	188	21	**5**	121	21	5	109	21	6	**96**	38
fdct1	40	377,849	4,568	TIMEOUT			592	3,633	58	**412**	2,893	59	470	**2,437**	62
fdct2	40	377,849	4,568	TIMEOUT			851	4,819	54	460	2,889	54	**419**	**2,500**	57
fpu2	312	81,303	939	MEMOUT			295	6,704	4	206	4,692	4	**149**	**3,621**	4
fpu5	300	81,303	939	MEMOUT			841	7,764	34	**168**	4,448	44	810	**4200**	42
mem_ctrl1	100	46,425	2,451	174	2,901	12	150	1,655	12	94	1,187	12	**71**	**899**	12
mem_ctrl2	100	46,425	2,451	94	3,012	6	76	1,702	6	57	1,291	6	**44**	**944**	6
mrisc1	42	18,034	631	**31**	546	61	37	353	80	38	**305**	86	39	315	93
rsdecoder2	196	11,380	1,623	**20**	393	47	28	356	47	35	300	47	45	**245**	47
spi1	576	2,103	223	270	871	27	175	596	37	187	620	79	**84**	**338**	38
vga1	40	154,213	1,337	**203**	5,150	9	219	2,845	9	275	2,213	35	307	**1,832**	51
vga2	40	154,213	1,337	**383**	5,187	30	447	2,913	82	520	2,253	128	481	**1,637**	115
wb1	132	3,552	407	6	240	8	4	154	8	**3**	128	8	5	**122**	8
wb2	132	3,552	407	5	233	5	4	153	5	**3**	124	5	3	**120**	5

debugging with interpolants were able to successfully solve these instances.

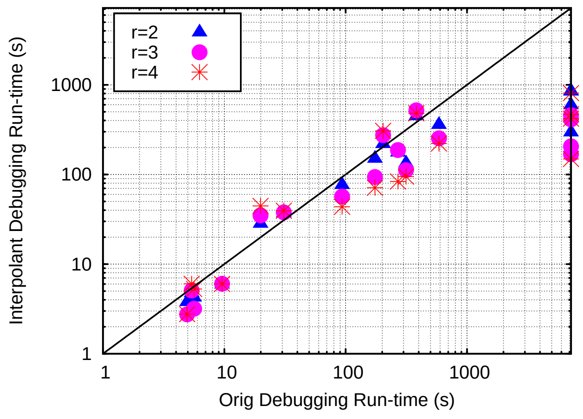
Taking a closer look at how run-time varies with the number of windows, r, Figure 5(b) shows how relative run-times of several designs vary with an increased r. The run-times are normalized to the orig instance indicated by $r = 1$. While most instances, show a reduction in run-time with larger r, `vga2` shows an increase. This can be attributed to the fact that the run-time of a debugging instance does not necessarily scale linearly with the problem size. However, most instances show a decrease in run-time as r is increased.

Figure 6 shows the benefit of using interpolants with respect to peak memory. Figure 6(a) shows the memory of using interpolants against orig on a log-log scale. All instances are below the 45 degree line indicating that they consistently require less memory.

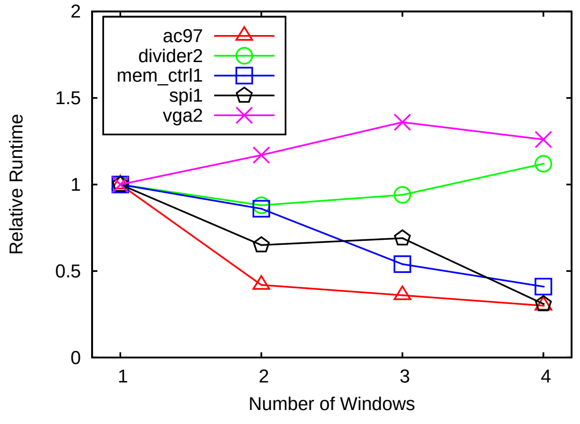
Looking more carefully at the relative memory usage for several instances, Figure 6(b) shows that the memory does not necessarily decrease inversely with r. `spi1` has a small relative increase from $r = 2$ to 3 but a much bigger relative decrease from $r = 3$ to 4. The reason for this is that the interpolant, which is not necessarily linear in size with the debugging instance, contributes to the peak memory. However, `ac972` shows a case where it does follow an inverse relation with r.

One would expect the early exit (line 14 from Algorithm 1) to contribute to this inverse relation. However, only the `mem_ctrl` and `wb` instances as well as `fpu2` used the early exit condition. This shows that the decrease in memory is due to the interpolant being significantly smaller than the instance it was generated from.

From Table I, we see that the number of suspects found generally increases as the r increases. This was explained in Section III-C due to potentially weaker interpolants being generated in later iterations of Algorithm 1. However, `spi1` and `vga2` show a case where the number of suspects actually decrease with increasing r. With $r = 3$ `spi1` generated 79 suspects while at $r = 4$ it generated 38. Similarly `vga2` had 128 at $r = 3$ and 115 at $r = 4$. These results suggest that although the interpolant is more likely to get weaker with increased r, how well it constrains the erroneous behavior can vary a great deal depending on the UNSAT core that was used.



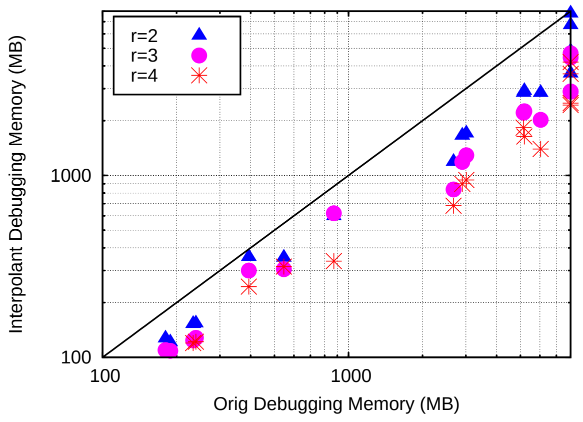
(a) Performance Scatter plot



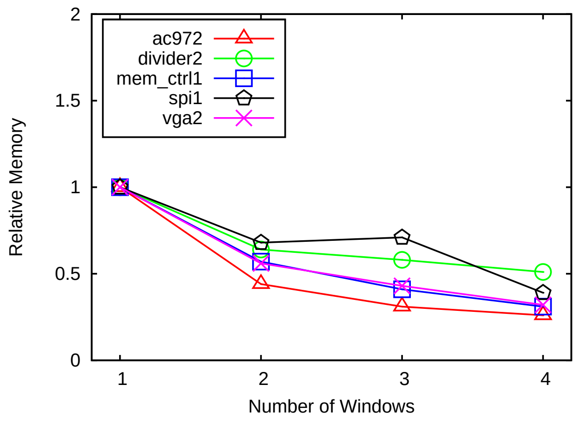
(b) Relative Performance Comparison

Fig. 5. Performance Results

Figure 7 shows the results from using multiple interpolants with $r = 4$ to constrain the debugging problem. The instances shown in this figure are ones where the number of suspects increased by a large amount over orig. For `spi1`, `vga1` and `vga2`, using multiple interpolants improved the quality of the debugging results by reducing the number of suspects.



(a) Memory Scatter plot



(b) Relative Memory Comparison

Fig. 6. Memory Results

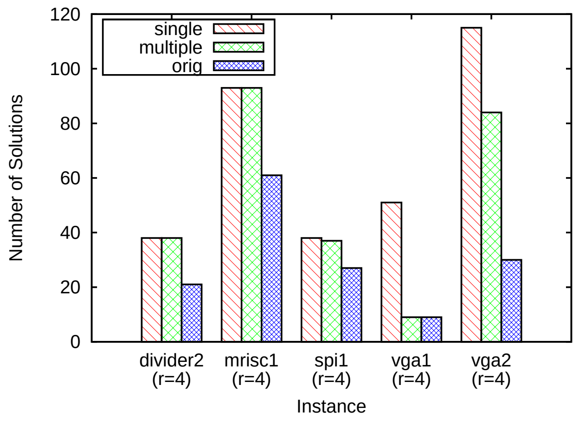


Fig. 7. Solutions using Multiple Interpolants

`vga1` shows a dramatic improvement in the number suspects returned where the interpolants help the suspects converge to the same value as orig. However, in some cases such as `divider2` and `mrisc`, multiple interpolants did not help constrain the problem further. These results show that the effectiveness of multiple interpolants is highly dependent on the debugging instance, where in some cases it can dramatically improve the resolution, while in others it does not help.

V. Conclusion

In this work, a scalable design debugging algorithm using interpolants is proposed. It partitions the problem into a sequence of smaller sub-problems that are easier to solve. Interpolants are used to reduce the number of simultaneous time-frames examined in the error trace by replacing sets of original clauses with a succinct approximation. The method is proven to be complete and an additional technique is presented to improve the quality of the debugging results using multiple interpolants. Experimental results show a large reduction in peak memory and improvements to run-time. This work encourages future research in design debugging using UNSAT cores and interpolation.

References

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, 2003.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[3] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *Int'l Conf. on CAD*, 2003, pp. 142–145.

[4] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in CAD*, 2004, pp. 159–173.

[5] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient sat-based unbounded symbolic model checking using circuit cofactoring," in *Int'l Conf. on CAD*, 2004, pp. 510–517.

[6] K.-H. Chang, V. Bertacco, and I. L. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *ICCAD*, 2005, pp. 1045–1051.

[7] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, 2008, pp. 5–10.

[8] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[9] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[10] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[11] S.Safarpour, M.Liffton, H.Mangassarian, A.Veneris, and K.A.Sakallah, "Improved design debugging using maximum satisfiability," in *Formal Methods in CAD*, 2007.

[12] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Great Lakes Symp. VLSI*, 2008.

[13] H. Mangassarian, A.Veneris, S.Safarpour, M.Benedetti, and D.Smith, "A performance-driven QBF-based on iterative logic array representation with applications to verification, debug and test," in *Int'l Conf. on CAD*, 2007.

[14] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, "Backspace: Formal analysis for post-silicon debug," in *Formal Methods in CAD*, 2008, pp. 1–10.

[15] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita, "A formal approach for debugging arithmetic circuits," in *IEEE Trans. on CAD*, vol. 28, no. 5, May 2009, pp. 742–754.

[16] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Comm. of the ACM*, vol. 5, pp. 394–397, 1962.

[17] L. Zhang, "Searching for truth: Techniques for satisfiability of Boolean formulas," Ph.D. dissertation, Princeton, 2003.

[18] W. Craig, "Linear reasoning. a new form of the herbrand-gentzen theorem," *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[19] K. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, 2003.

[20] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[21] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[22] OpenCores.org, "http://www.opencores.org," 2007.