Searching for Bugs using Probabilistic Suspect Implications

Neil Veira, *Student Member, IEEE*, Zissis Poulos, *Member, IEEE*,
and Andreas Veneris, *Senior Member, IEEE*

***Abstract*—Due to the excessive cost associated with manual RTL design debugging, automated tools are often employed to identify a set of suspect bug locations. To further accelerate the process, one observes that the anytime behaviour of these tools allows partial results to be analyzed before the suspect search is complete. Thus, it is preferable for the tool to maximize the number of suspects that are found in the early stages of its search. Towards this end, this paper proposes a new SAT–based debugging algorithm which predicts where solutions are most likely to be found and prioritizes examining these locations. Two techniques are proposed to predict solution locations by learning from historical debug data. The first technique does so using belief propagation on a probabilistic graph, while the second trains a neural network to classify candidate suspects as solutions or non-solutions. Intensive empirical evaluation demonstrates that these techniques can predict suspect sets with accuracies of 81% and 87%, respectively, but the second method requires more training data and careful hyperparameter tuning in order to do so. Furthermore, when guided by these suspect prediction models, the proposed debugging algorithm finds an average of 83% more suspects within a given amount of time.**

***Keywords*—Debugging, Verification, RTL, Suspect implications, Prediction models, VLSI**

I. Introduction

State-of-the-art verification tools provide a highly robust, automated framework to prove the functional correctness of Very Large Scale Integration (VLSI) designs. However, when verification fails, identifying the root cause of the failure, *i.e.* debugging, can be challenging. Debugging is typically based on fixing a counterexample or *error trace* — a sequence of input stimuli that exposes the erroneous behaviour and characterizes the failure — but the size and complexity of modern VLSI designs have made this the most time–consuming stage of the verification cycle [1].

To help alleviate this cost, a line of work in Computer Aided Design (CAD) aims to automate the bug searching process. Most popular approaches to this task compute a set of possible or probable bug locations, called *suspects*. Approaches based on Boolean Satisfiability (SAT) [2] or 0-1 Integer Linear Programming (ILP) [3] do so by searching for design locations at which a change can rectify the error trace. This provides a formal guarantee that the returned suspect set will include the bug location, when defined at the same level of granularity as the CAD tool.

This provides a formal guarantee that the bug location will be among the returned suspects, each of which can then be investigated in greater detail.

Many techniques have since been developed to improve the scalability of SAT–based debugging. Techniques involving design abstraction [4], time frame abstraction [5], or unsatisfiable cores [6] are able to greatly reduce the size of the SAT instance. Others make use of alternative formal engines such as Maximum Satisfiability [7] or Quantified Boolean Formulas [8] to reduce the problem size. Once all suspect locations have been found, further techniques can be applied to narrow down or prioritize the possibilities [9], [10] in order to pinpoint the actual bug location from among them. Yet despite these advances, debugging remains an expensive and time–consuming process on industrial–scale designs.

In this paper we develop a complementary enhancement methodology by capitalizing on the anytime behaviour of the bug search: because most automated debugging tools return candidate solutions "on-the-fly", suspects found early on can be analyzed before the search is complete. As such, an ideal search algorithm would prioritize areas of the search space that contain solutions, while non-solution areas would be examined later. Our methodology realizes this behaviour by predicting which candidate suspects are most likely to be solutions and then guiding the search accordingly. This reduces the average time required to find a given number of solutions, allowing for detailed analysis of the identified suspects to begin earlier. It also means that on average, more solutions will be found within a given amount of time, which leads to greater flexibility in correcting the error. This can also be particularly beneficial for applications in which a large set of suspects is desired, including design rewiring [11], failure triage [12], and unreachability diagnosis [13].

Our method draws upon work on the use of structural dominance relationships in SAT–based debugging [14]. The key insight of [14] is that the presence of a suspect at a certain design location implies the presence of suspects at all structural dominators of this location. This allows the SAT search to avoid explicitly modeling and examining dominator locations, as the suspects can instead be inferred from other suspects.

While this technique improves debugging efficiency, it is limited by considering only suspects implied via structural dominance. In general, suspects can be related in much more complex ways, and other, weaker forms of implication relationships may exist. For instance, it may be the case that the presence of one suspect induces the presence of another in all usual design behaviour, while only under rare stimuli can the former be a suspect while the latter is not. Such a relationship cannot be detected using only structural dominance. Yet if one could learn this fact, then it would be reasonable to *guess* that the latter suspect (the consequent) will exist having observed the former (the antecedent). We call such relationships *probabilistic suspect implications*.

Probabilistic suspect implications are key to achieving the objective outlined above, as we can predict that probabilistically implied suspects are more likely to be solutions. More specifically, we use probabilistic suspect implications to rank all candidate suspects by their probability of being solutions. If

this ranking is reasonably accurate, then on average, a search algorithm which prioritizes candidate suspects accordingly would find solutions faster. Probabilistic suspect implications also lead way to an approximate debugging methodology: one can first use an incomplete SAT search to find a subset of the suspects, and then predict the remaining suspects from the implications. This can greatly mitigate the cost of the bug search, at the expense of some inaccuracies in the returned suspect set.

Unlike dominance relationships, probabilistic implications are defined by empirical observation rather than design structure. We propose two methods to compute probabilistic suspect implications using statistical techniques that rely on data from historical debugging sessions. Both methods are complementary to one another and offer different sets of advantages and tradeoffs. The first method builds a probabilistic graph of implications between pairs of suspects, and estimates the probability of a suspect using a pass of belief propagation on this graph [15]. We refer to this method as suspect implication graphs (SIG). Experiments show that SIG is effective for ranking candidate suspects, but loses accuracy when predicting which candidate suspects will actually be in the solution set.

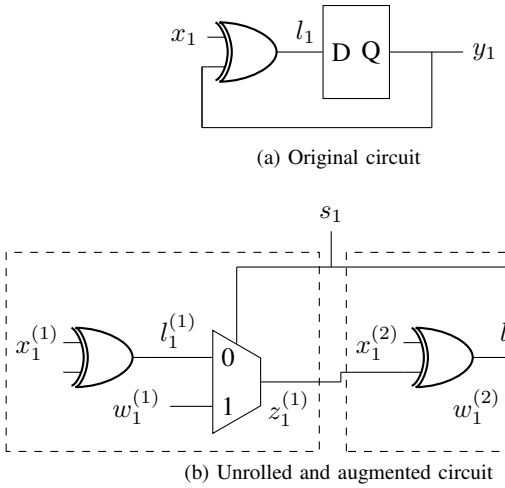
The second method, named `suspect2vec` [16], addresses this shortcoming by instead formulating the primary objective as a binary classification task. A single–layer neural network is trained to classify each candidate suspect as a solution or non-solution. The output of the network also yields a suspect ranking as a by-product. `Suspect2vec` can outperform SIG on average, achieving an accuracy of 93.5% and 86.9% in the suspect ranking and set prediction tasks, respectively. However, it requires more training data and careful hyperparameter tuning in order to achieve these results.

As a third contribution, we describe how suspect prediction can be incorporated into a SAT–based debugging algorithm so as to prioritize areas of the search space that are more likely to contain solutions. The proposed algorithm accomplishes this by partitioning the search into multiple passes, each of which models only a subset of potential bug locations in the SAT instance. Suspect priority is enforced by placing higher priority suspects in earlier passes. To evaluate the new algorithm we measure the suspect recall — defined as the fraction of suspects that have been found at a specific point in time — and take the mean over the period of execution. Results indicate that on difficult debug instances, our method can improve the average suspect recall by 83%.

The remainder of this paper is organized as follows. Section II introduces the prerequisite concepts on SAT–based debugging and suspect implicature . Sections IV and V describe each of methods for estimating probabilistic suspect implications, while Section VI explains the new debugging algorithm which incorporates this information. Section VII then presents independent experimental evaluations of both the suspect prediction and debugging techniques. Finally, Section VIII concludes the paper.

II. Preliminaries

A verification failure occurs when the observed design behaviour differs from the expected (golden) behaviour. Erroneous behaviour is often exposed by simulators or property checkers, in the form of an assertion failure or a mismatch between the primary output signal values of the design under test and the golden values. These tools can then produce an



(a) Original circuit

(b) Unrolled and augmented circuit

Fig. 1. Example circuit augmentation for SAT–based debugging. A potential bug is modeled at location l_1 over two time frames.

associated counterexample or *error trace* — a sequence of signal values leading up to the point of failure.

Given an error trace, debugging aims to identify the design location at which an incorrect logic element initiated the erroneous behaviour, and at which a change can rectify the error. In general, many such locations may exist, and so a deeper understanding of the design — beyond the constraints of the error trace — is required to identify the bug. Therefore, a common approach to debugging involves first finding all possible bug locations, called *suspects*, using automated tools. Further techniques can then be applied to rank or filter the suspects to help identify the bug faster. Throughout this paper we also use the term *candidate suspects* to refer to design locations that are being examined during the suspect search.

A. SAT-Based Design Debugging

SAT-based design debugging [2] formulates the suspect search problem as an instance of Boolean Satisfiability (SAT). Given a buggy circuit and an error trace, the circuit is augmented as shown in Figure 1. First, the circuit is unrolled over the length of the error trace, yielding a fully combinational iterative logic array (ILA) representation [17]. Each circuit location is augmented with error select logic, which effectively selects between applying a change at this location or retaining the original behaviour. The augmented circuit is then transformed into a conjunctive normal form (CNF) formula, with the input and output signals constrained to the test vectors and the expected outputs, respectively. Another constraint is added to ensure that at most N select signals may be activated in a satisfying assignment, where N is a configurable parameter. In practice, solutions of cardinality $N = 1$ are often the most valuable, as they indicate single bug locations which can be more easily fixed. Therefore, throughout this paper we focus on the task of finding all single–cardinality solutions.

After constructing the CNF formula, the complete set of single–cardinality solutions is found with an iterative procedure. In each iteration a SAT solver returns a satisfying assignment, and the activated select variable s_i is added to the solution set. The clause $\neg s_i$ is then added to the formula so that this solution is no longer valid, and the solver must find an assignment with a different select variable activated. This process is repeated until the formula becomes unsatisfiable.

III. Motivation

In practice, suspect bug locations tend to be strongly related to one another. One important form of relationship is *structural dominance*. Formally, in a gate–level netlist representation of the design, a node u is said to be a dominator of node v if every path from v to a primary output passes through u [18]. This means that if a change at node v can fix the erroneous behaviour, then there must also exist a change at the dominator node u which can fix the erroneous behaviour. Thus, in the context of design debugging, a structural dominance relation between nodes is equivalent to an implication relation between suspects: $s_v \implies s_u$, where s_u and s_v denote bug suspects corresponding to u and v.

This concept can be generalized to dominance between groups of nodes or suspects. This is useful for RT–level analysis, where a single block may contain multiple nodes and multiple input and output signals. A group of nodes $U = \{u_1, ..., u_n\}$ is said to dominate another group of nodes $V = \{v_1, ..., v_n\}$ if every path from a node V to a primary output passes through some node in U. Analogously, for RT–level debugging we can say that a group of suspects $S_V = \{s_1, ..., s_n\}$ implies a suspect s_u if the existence of a debugging solution for every $s_i \in S_V$ implies the existence of a debugging solution at s_u.

Structural dominance relationships have many applications [19], [20], and the corresponding suspect implications have been used to enhance the performance of SAT-based debugging [14]. The idea is to use a suspect implication $s_v \implies s_u$ to infer that suspect s_u will be a solution, having observed the suspect s_v, rather than relying on the SAT solver to find the solution s_u. This reduces the number of queries to the SAT solver and greatly improves debugging runtime.

In this work we extend this idea by considering a more general form of suspect relationships, which we coin as *probabilistic suspect implications*. Intuitively, two nodes u and v may be closely tied together even if neither node structurally dominates the other. For example, it may be the case that most signal propagation paths from v to a primary output pass through u, and only under rare circumstances can a signal propagate from v without passing through u. In such cases we can say that the existence of a solution at s_v implies the existence of a solution at s_u *with high probability*. More generally, we can extend the concept to groups of RTL suspects and say that $S_V \overset{p}{\Rightarrow} s_u$ if the existence of a solution for every $s_i \in S_V$ implies the existence of solution s_u with probability p.

Such relationships can be identified by observations from a *debug history*, which is a set of solution sets from previous debug sessions on a given design. Such data makes it possible to identify suspect locations that tend to occur together using statistical methods, which has advantages over formal structural analysis. In particular, statistical methods can identify a much wider variety of relationships than would be feasible using structural analysis alone. This is because if a probabilistic relationship exists between s_i and s_j, whether or not s_i and s_j occur together as suspects for a bug (essentially "activating" the implication) depends on the stimuli of the circuit. Determining the probability that the stimuli will cause s_i and s_j to occur together (*i.e.*, the strength of the implication) may require considering an exponential number of execution paths.

Additionally, structural analysis is unable to account for the different likelihoods of different stimuli occurring during realistic operation of the design. This information should be reflected in data generated from the design's testbench, which would allow it to be discovered by statistical methods.

A potential drawback of the observational approach is its reliance on historical debug data.However, such data is often readily available during the later phases of the development cycle, at which point many bugs have already been created, debugged, and fixed. For instance, design modules are often maintained and updated for many years after their initial deployment. Any new bugs that are introduced by these updates can be more easily resolved by learning from previous bugs. As long as suspect sets from these debugging sessions are persisted, then the proposed methodology can be applied at no additional cost.

Formally, let $\mathcal{F}_H = \{F_1, ..., F_N\}$ denote a set of N historical failures, each of which may be a single assertion failure or an incorrect signal value. Let $\mathcal{S}_H = \{S_1, ..., S_N\}$ denote their associated suspect sets, where each S_i is the set of all single–cardinality solutions for the SAT–based debugging instance corresponding to F_i. We also define $SU = S_1 \cup ... \cup S_N$ to be the set of all historically observed suspect locations.

As with suspect implications based on structural dominance, probabilistic implications based on historical data can be useful in SAT–based debugging. Having observed a set of suspects S_V, we can guess that the probabilistically implied suspects of S_V will also be solutions, and guide the SAT search accordingly. Moreover, because probabilistic implications are necessarily approximate in nature, these relationships can be estimated using highly efficient statistical procedures. The following two sections present two methods to precisely define and compute probabilities between suspects and to identify probabilistic implications.

IV. Suspect Prediction via Implication Graphs

In this section we present the first method for estimating probabilistic implication relations between suspects. We then show how these relationships can be used to predict the solution set of a debug instance, given a subset of the solutions.

Let F denote a failure and S denote its set of single–cardinality debug solutions. We assume the availability of a debug history $(\mathcal{F}_H, \mathcal{S}_H)$, where $\mathcal{F}_H$ may or may not contain F. The set S is thus not known, however, we are given a subset of suspects $S_{\text{obs}} \subseteq S$. Then the relation $s_i \overset{p}{\Rightarrow} s_j$ between some pair of suspects s_i and s_j is characterized by the probability $p = P(s_j \in S | s_i \in S)$. To simplify the presentation, we use the shorthand notation $P(s_j|s_i)$ to mean the same thing. These probabilities can then be used for the *suspect set prediction task*, whose goal is to estimate the suspect set S given only $\mathcal{S}_H$ and S_{obs}; that is, find a function Pred such that $\text{Pred}(S_{\text{obs}})$ is as similar to S as possible.

A. Learning Suspect Relationships

Consider two suspects $s_i, s_j \in SU$. Let $\text{count}(s_i) = |\{S : s_i \in S \wedge S \in \mathcal{S}_H\}|$ (*i.e.*, the number of times that s_i occurs in the historical data). We also let $\text{count}(s_i, s_j) = |\{S : s_i \in S \wedge s_j \in S \wedge S \in \mathcal{S}_H\}|$ (*i.e.*, the number of times that both s_i and s_j occur together in the historical data).

From a statistical perspective, we can view $\text{count}(s_i, s_j)$ as a data point which is generated by the underlying parameter $P(s_j|s_i)$. Here the random variable $X_j = I[s_j \in S]$, where

I is the indicator function, is a Bernouilli random variable with probability $p = P(s_j|s_i)$ of being 1. Each occurrence of s_i in $\mathcal{S}_H$ is a Bernouilli trial, of which there are count(s_i). Therefore, the number of successful trials is the random variable count(s_i, s_j), which follows a binomial distribution:

$$P(\text{count}(s_i, s_j)|P(s_j|s_i), \text{count}(s_i)) = \mathcal{B}(x = \text{count}(s_i, s_j); n = \text{count}(s_i); p = P(s_j|s_i)) \quad (1)$$

Eq. 1 is the *data likelihood* with respect to s_i and s_j — the probability of observing the data count(s_i, s_j) given the parameter $P(s_j|s_i)$. As such, we can estimate the value of $P(s_j|s_i)$ by the *maximum likelihood estimation* (MLE), which is defined as the value that maximizes the data likelihood. For a binomial distribution, this is given by:

$$P_{\text{MLE}}(s_j|s_i) = \frac{\text{count}(s_i, s_j)}{\text{count}(s_i)} \quad (2)$$

Unfortunately, a major deficiency of the MLE estimate is that it is prone to overfitting, especially when the amount of data is small, as it can be often the case in the application described here. For instance, suppose a suspect s_i occurs only once in $\mathcal{S}_H$, as part of some suspect set $S_k \in \mathcal{S}_H$. Eq. 2 would give $P_{\text{MLE}}(s_j|s_i) = 0$ for all $s_j \notin S_k$. However, it would much too strong of a conclusion that all suspects $s_j \notin S_k$ can never co-occur with s_i, having only observed s_i once. Eq. 2 would also give $P_{\text{MLE}}(s_j|s_i) = 1$ for all $s_j \in S_k$, which is similarly problematic.

This issue is typically dealt with by using the *maximum a posteriori* (MAP) estimate [21] instead of the MLE. With MAP, the parameter values are selected so as to maximize the posterior distribution rather than the likelihood. Using Bayes' rule, this is equivalent to maximizing the product of the likelihood and the prior distributions: $W_{\text{MAP}} = \text{argmax}_W\, P(W|D) = \text{argmax}_W\, P(D|W)P(W)$, where W and D denote the parameters and data of a model, respectively. The prior distribution $P(W)$ is chosen to reflect one's prior belief or bias regarding the values of W.

In our scenario, we heuristically choose the prior to be a Gaussian distribution $\mathcal{N}(x; \mu, \sigma^2)$ with $\mu = 0.5$ and $\sigma^2 = 0.2$. The Gaussian shape is chosen to reflect our subjective bias that most suspect pairs are unrelated (with $P(s_j|s_i)$ near 0.5), while relatively few suspect pairs are strongly related (with $P(s_j|s_i)$ near 0 or 1). The mean of 0.5 reflects the fact that the parameters are probabilities and must lie between 0 and 1. The variance of 0.2 was chosen to be sufficiently large so as to allow the model to fit the data, while being sufficiently small to prevent extreme overfitting. Empirically, we found that the end results are quite robust to the value of σ^2; anything in the range of 0.1–0.5 tends to work well.

Overall, this gives the following estimate for the suspect implication probabilities:

$$P_{\text{MAP}}(s_j|s_i) = \underset{p}{\text{argmax}} \exp\left(-\frac{(p-0.5)^2}{0.4}\right) \times \binom{\text{count}(s_i)}{\text{count}(s_i, s_j)} p^{\text{count}(s_i,s_j)}(1-p)^{\text{count}(s_i)-\text{count}(s_i,s_j)} \quad (3)$$

Eq. 3 has the effect of pulling extreme values towards 0.5 when the amount of data for s_i is small, thereby regularizing them. This is illustrated by the following example.

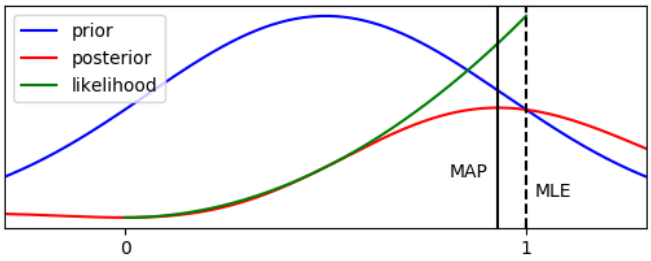


Fig. 2. Likelihood, prior, and posterior distributions for Example 1.

Example 1. *Consider the following historical debug data:*

$$\begin{aligned}
\mathcal{S}_H &= \{S_1, S_2, S_3, S_4, S_5\} \\
S_1 &= \{s_1, s_4, s_5\} \\
S_2 &= \{s_2, s_5\} \\
S_3 &= \{s_1, s_2, s_3, s_4, s_5\} \\
S_4 &= \{s_3, s_5, s_6\} \\
S_5 &= \{s_1, s_3, s_4\}
\end{aligned}$$

To estimate the probability of the implication $s_2 \overset{p}{\Rightarrow} s_5$, we begin by computing count$(s_2) = 2$ and count$(s_2, s_5) = 2$. By Eq. 2, the MLE estimate for $P_{MLE}(s_5|s_2)$ would be 1.0. *By Eq. 3, the MAP estimate is:*

$$\begin{aligned}
P_{MAP}(s_5|s_2) &= \underset{p}{\text{argmax}} \exp\left(-\frac{(p-0.5)^2}{0.4}\right) \\
&\quad \times \binom{2}{2} p^2(1-p)^{2-2} \\
&= 0.93
\end{aligned}$$

Figure 2 plots the likelihood, prior, and posterior distributions for this example. The MLE estimate of 1.0 is severely overfit to a small amount of data. On the other hand, MAP provides a regularized estimate by pulling the value closer to 0.5. The degree of regularization can be tuned by adjusting the prior variance.

B. Multiple–Suspect Implications

In this section we extend the preceding methodology to estimate the probability of a multiple–suspect implication $S_A \overset{p}{\Rightarrow} s_j$, where $S_A \subseteq SU$ denotes an antecedent *set* of suspects, and s_j denotes a single consequent suspect. We denote this probability by $P(s_j|S_A)$. The approach taken in Eq. 3 would not be suitable for this task, because most possible antecedent sets S_A will have never occurred in the historical data (assuming a small data set), meaning that count(S_A) would be 0. Instead, we show how to approximately derive $P(s_j|S_A)$ from the estimated single–suspect probabilities $P(s_j|s_i)$ for every $s_i \in S_A$. This approach can be applied to any possible S_A and s_j.

Consider the directed graph $G = (V, E)$, where each vertex $v_i \in V$ corresponds to the suspect $s_i \in SU$. For each ordered pair of suspects (s_i, s_j), there exists a directed edge $(v_i, v_j) \in E$ weighted by $P(s_j|s_i)$. Thus, G is a graphical model for the events of each suspect being a solution, with edges modeling conditional dependencies between these events. Given the antecedent set S_A, $P(s_j|S_A)$ for any s_j can be estimated using a single pass of belief propagation on G. The result is given by the following proposition [22].

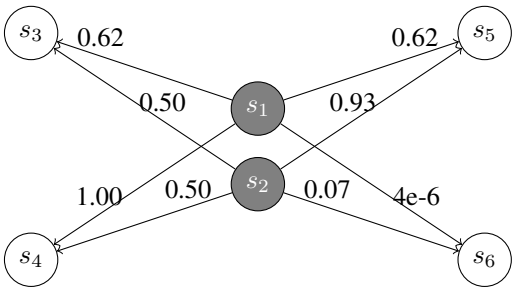


Fig. 3. Suspect implication graph for Example 2. Only edges from $s_i \in S_{\text{obs}}$ to $s_j \in SU \setminus S_{\text{obs}}$ are shown.

Proposition 1. *Under the assumption of independence between the weights of E, the probability of suspect s_j occurring given S_A is:*

$$P(s_j|S_A) = \begin{cases} 1, & s_j \in S_A \\ 1 - \prod_{s_i \in S_A} (1 - P(s_j|s_i)), & s_j \notin S_A \end{cases} \quad (4)$$

Proof: For $s_j \in S_A$, $P(s_j|S_A)$ is trivially 1. For $s_j \notin S_A$, consider the complementary event of suspect s_j not being a solution, and let $P(\bar{s_j}|S_A)$ denote its probability. The only way this event can occur is if s_j is not implied by any of the suspects $s_i \in S_A$. Assuming independence between implication events, this gives $P(\bar{s_j}|S_A) = \prod_{s_i \in S_A} (1 - P(s_j|s_i))$. Eq. 4 then follows immediately. ■

While the derivation of Eq. 4 involves several assumptions and approximations which may not hold perfectly in practice, it nonetheless serves as a useful tool for the purposes of suspect ranking and suspect set prediction, as shown in Section VII. The following example demonstrates its use.

Example 2. *Consider the data $\mathcal{S}_H$ from Example 1. Suppose the suspects $S_{obs} = \{s_1, s_2\}$ have been found as solutions. We can build the suspect implication graph G using edge weights $P(s_j|s_1)$ and $P(s_j|s_2)$ for each $j = 3, 4, 5, 6$ computed with Eq. 3. The result is shown in Figure 3. Next, the multiple–suspect implication probabilities $P(s_j|S_{obs})$ are computed with Eq. 4:*

$$\begin{aligned} P(s_3|S_{obs}) &= 1 - (1 - P(s_3|s_1))(1 - P(s_3|s_2)) \\ &= 0.89 \end{aligned}$$

And similarly,

$$\begin{aligned} P(s_4|S_{obs}) &= 0.999998 \\ P(s_5|S_{obs}) &= 0.973 \\ P(s_6|S_{obs}) &= 0.070 \end{aligned}$$

C. Suspect Set Prediction

In this subsection we show how probabilistic suspect implications can be used to predict the debugging solution set S, given a subset of suspects $S_{\text{obs}} \subseteq S$. First, we can compute the probability of every candidate suspect $s_j \in SU$ being a solution using Eq. 4. This leads to a ranking of all candidate suspects, which we denote by $\text{Rank}(SU|S_{\text{obs}}) = s_{r_1}, ..., s_{r_{|SU|}}$. This ranking is key to the enhanced debugging search algorithm that we present in Section VI.

This ranking does not tell us which candidate suspects will actually be in the solution set and which will not, which may be useful for approximate debugging [12], [15]. However, if the ranking is good then there should exist some k such that s_{r_i} is a solution for most $i \leq k$, and s_{r_i} is not a solution for most $i > k$. Intuitively, k draws a line through $\text{Rank}(SU|S_{\text{obs}})$ to approximately separate the solutions from the non-solutions. The following definition explains how to find such a k.

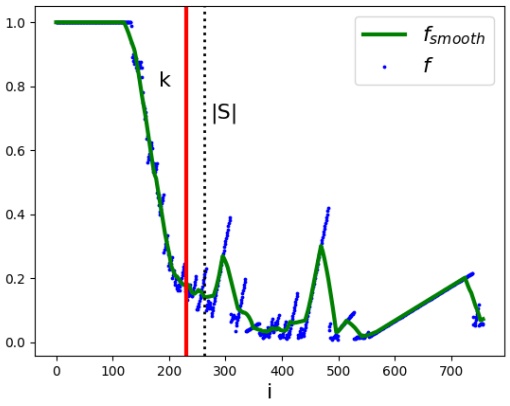


Fig. 4. Illustration of the termination point for Example 3.

Definition 1. *Termination Point*

Given $\text{Rank}(SU|S_{obs})$ obtained from Eq. 4, for each i such that $|S_{obs}| < i \leq |SU|$, define the function $f : \mathbb{Z} \to \mathbb{R}$ where

$$\begin{aligned} f(i) &= P(s_{r_i}|\{s_{r_1}, ..., s_{r_{i-1}}\}) \\ &= 1 - \prod_{1 \leq j < i} (1 - P(s_{r_i}|s_{r_j})) \end{aligned} \quad (5)$$

Let $f_{smooth}(i) = \frac{1}{2\delta+1} \sum_{j=i-\delta}^{i+\delta} f(j)$. Then the termination point, k, is the smallest index i such that $f_{smooth}(i)$ is a local minimum.

Intuitively, $f(i)$ can be thought of as the incremental conditional probability of the next ranked suspect being a solution, given that the previous $i - 1$ suspects are solutions. Because f can be quite noisy, smoothing is applied by taking the running average with a smoothing width of δ, whose value is determined empirically. Local minima of f_{smooth} are points at which, according to the data set $\mathcal{S}_H$, further solutions are unlikely.

Example 3. *To properly illustrate the termination point a larger data set is needed. Figure 4 plots f and f_{smooth} for a failure from our experimental data (see Section VII). The horizontal axis is the suspect rank index, while the blue points show $P(s_{r_i}|\{s_1, ..., s_{r_{i-1}}\})$. The solid green line plots f_{smooth}. As shown in the figure, the estimated stopping point is $k = 231$, while the actual number of suspects is $|S| = 262$.*

To summarize, the full procedure to predict a suspect set is outlined in Algorithm 1. We refer to this method as Suspect Implication Graph (SIG) prediction.

V. SUSPECT2VEC: A NEURAL PREDICTION MODEL FOR SUSPECT IMPLICATIONS

In this section we present `suspect2vec` — an alternative method to estimate suspect implications and infer the solution set of a debug instance, given a subset S_{obs}. `Suspect2vec` addresses the main performance bottleneck of the SIG method,

Algorithm 1 SIG-PREDICTION($\mathcal{S}_H, S_{\text{obs}}$)

1: Compute $P(s_i|s_j)$ for every ordered pair s_i, s_j using Eq. 3
2: Compute $P(s_i|S_{\text{obs}})$ for every s_i using Eq. 4
3: Sort SU by $P(s_i|S_{\text{obs}})$ to obtain Rank$(SU|S_{\text{obs}})$
4: Compute k using Definition 1
5: Return Pred$(S_{\text{obs}}) = \{s_{r_1}, ..., s_{r_k}\}$

which is estimating the number of suspects k. Instead of trying to predict k directly, `suspect2vec` aims to classify each candidate suspect as a solution or non-solution. As a side–effect, `suspect2vec` is also able to estimate probabilities for single–suspect and multiple–suspect implications, as well as a suspect ranking. As shown in Section VII, `suspect2vec` is able to outperform SIG in prediction accuracy in the majority of cases.

A. Prediction Model

`Suspect2vec` can be formulated as a neural prediction model, taking as input an encoding of S_{obs} and outputting a value y_i for each $s_i \in SU$. Value y_i is the predicted label (solution or non-solution) for candidate suspect s_i, but can also be interpreted as $P(s_i|S_{\text{obs}})$. To produce these values, the network's internal parameters learn *embeddings* of all candidate suspects, which are representations of the suspects as d-dimensional vectors.

Embedding representations have been shown to be highly effective for capturing relationships between objects. For example, the `word2vec` model [23] is able to learn embedding representations of words, which can predict the occurrence of a word given its context in a sentence. `Word2vec` assigns similar embeddings to semantically similar words because such words tend to occur in similar contexts [24]. The `suspect2vec` model repurposes this property in order to capture correlations between nodes in a logic circuit. Intuitively, the model should learn more similar embeddings for more closely related suspects, because related suspects tend to occur in the same or similar suspect sets.

In detail, with each candidate suspect $s_i \in SU$ we associate an input vector and an output vector, denoted by $\mathbf{v}_i$ and $\mathbf{v}'_i$, respectively. Intuitively, the use of two vectors for each suspect allows the model to treat suspects differently depending on whether they belong to the antecedent or the consequent of the implication. Then the probability of each single–suspect implication $s_i \overset{p}{\Rightarrow} s_j$ is defined as:

$$P(s_j|s_i) = \sigma(\mathbf{v'_j} \cdot \mathbf{v_i}) \tag{6}$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ denotes the logistic function. Defined in this way, the product $\mathbf{v'_j} \cdot \mathbf{v_i}$ between the input and output vectors measures the strength of the implication from s_i to s_j: strongly related suspects will have similar vectors and hence a large positive score, while dissimilar suspects will receive large negative scores. The logistic function σ then maps these scores to the range $(0, 1)$ so that they can be interpreted as probabilities.

The probability of a multi–suspect implication S_{obs} can be defined in a similar manner, because suspect embeddings make it possible to derive representations for an arbitrary set of suspects. We define the vector representation of S_{obs}, denoted $\mathbf{v_{obs}}$, as the mean vector of all suspects in S_{obs}:

$$\mathbf{v}_{\text{obs}} = \frac{1}{|S_{\text{obs}}|} \sum_{s_i \in S_{\text{obs}}} \mathbf{v_i} \tag{7}$$

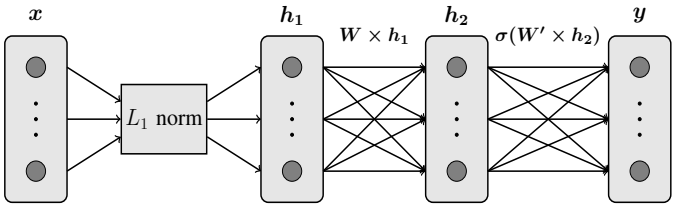


Fig. 5. The `suspect2vec` neural prediction model

This is similar to the common practice in natural language processing of representing a sentence by the mean of word vectors within the sentence [25], and is motivated by the observation that sums of embedding vectors produce semantically meaningful vectors. The mean is taken rather than the sum in order to eliminate the dependence of the model on the size of the input set.

With Eq. 7, we can define $P(s_j|S_{\text{obs}})$ as:

$$P(s_j|S_{\text{obs}}) = \sigma(\mathbf{v'}_j \cdot \mathbf{v}_{\text{obs}}) \tag{8}$$

This leads to a simple scheme for predicting an unknown suspect set S: for each possible suspect s_j, predict $s_j \in S$ if $P(s_j|S_{\text{obs}}) \geq 0.5$, and $s_j \notin S$ otherwise. Note that it is not possible to predict S in this way with SIG, because the candidate suspect scores produced by Eq. 4 do not have an interpretation as classification labels. In contrast, `suspect2vec` is trained such that Eq. 8 can be interpreted in this way. This is explained further in Section V-B.

Figure 5 shows how `suspect2vec` can be expressed as a neural network. The embedding vectors $\mathbf{v}_i$ and $\mathbf{v}'_i$ are collected into weight matrices $\mathbf{W}$ and $\mathbf{W}'$, where $\mathbf{v}_i$ is the i^{th} column of $\mathbf{W}$, and $\mathbf{v}'_i$ is the i^{th} row of $\mathbf{W}'$. The input set S_{obs} is encoded as a bag-of-suspects vector $\mathbf{x}$ of length $|SU|$, with $\mathbf{x}_i = 1$ if $s_i \in S_{\text{obs}}$, or $\mathbf{x}_i = 0$ otherwise. This is passed through an L_1 normalization layer so that $\mathbf{h_1} = \frac{1}{|S_{\text{obs}}|}\mathbf{x}$. Multiplying $\mathbf{h_1}$ by the weight matrix $\mathbf{W}$ produces $\mathbf{h_2} = \mathbf{v}_{\text{obs}}$ in the next hidden layer. The final layer multiplies by $\mathbf{W}'$ and applies the σ function, producing a vector $\mathbf{y}$ of length $|SU|$ at the output. The vector $\mathbf{y}$ contains the predicted labels for each suspect, because $\mathbf{y}_j = P(s_j|S_{\text{obs}})$ as defined by Eq. 8.

B. Training Procedure

We now describe the optimization objective and procedure which, based on the historical data $\mathcal{S}_H$, trains `suspect2vec` to maximize prediction accuracy and learn representative suspect embeddings. For a suspect set S, each output y_i should be interpreted as a binary classification label for candidate suspect s_i. Therefore, we wish to encourage y_i to be close to 1 for $s_i \in S$ and close to 0 for $s_i \notin S$. This is achieved by means of the cross-entropy loss function. Letting $\hat{y}_i$ denote the target class label for candidate suspect s_i (1 if $s_i \in S$ and 0 otherwise), the cross-entropy loss is defined as:

$$L_{\text{CE}} = -\sum_{i=1}^{|SU|} \hat{y}_i \log y_i + (1 - \hat{y}_i) \log(1 - y_i)$$

To be able to predict any suspect set S, given any subset $S_{\text{obs}} \subseteq S$, the full optimization objective is to minimize the

Algorithm 2 SUSPECT2VEC-TRAIN($\mathcal{S}_H, d, \eta, e$)

```
1: for i ← 1 to |SU| do
2:     v_i ← RANDOM-VECTOR(d)
3:     v'_i ← RANDOM-VECTOR(d)
4: end for
5: for iter ← 1 to e do
6:     for each S ∈ S_H do
7:         S_obs ← RANDOM-SUBSET(S)
8:         for i ← 1 to |SU| do
9:             v_i ← v_i − η∇_{v_i} L
10:            v'_i ← v'_i − η∇_{v'_i} L
11:        end for
12:    end for
13: end for
```

following loss function:

$$L = -\sum_{S \in \mathcal{S}_H} \frac{1}{2^{|S|}} \sum_{S_{\text{obs}} \in 2^S} \sum_{i=1}^{|SU|} \Big[\hat{y}_i(S) \log P(s_i | S_{\text{obs}}) + (1 - \hat{y}_i(S)) \log(1 - P(s_i | S_{\text{obs}})) \Big] \tag{9}$$

where 2^S denotes the power set of S. The term $\frac{1}{2^{|S|}}$ is introduced to balance the total contributions of all $S \in \mathcal{S}_H$ regardless of their differing sizes.

Eq. 9 is minimized using gradient descent. However, an exact minimization is not feasible in practice because the power set 2^S has size $2^{|S|}$, which can be extremely large. Therefore, we approximate L by randomly sampling multiple subsets. For each sample S_{obs}, the gradients are given by:

$$\nabla_{\mathbf{v}_i} L = \begin{cases} \frac{1}{|S_{\text{obs}}|} \mathbf{v'}_i \left[\sigma(\mathbf{v'}_i \cdot \mathbf{v}_{obs}) - \hat{y}_i \right], & s_i \in S_{\text{obs}} \\ 0, & s_i \notin S_{\text{obs}} \end{cases} \tag{10}$$

$$\nabla_{\mathbf{v'}_i} L = \mathbf{v}_{\text{obs}} \left[\sigma(\mathbf{v'}_i \cdot \mathbf{v}_{\text{obs}}) - \hat{y}_i \right]$$

Algorithm 2 describes the full optimization procedure, named SUSPECT2VEC-TRAIN. This procedure takes as input the training data $\mathcal{S}_H$ and three hyperparameters: the embedding dimensionality, d, the learning rate, η, and the number of iterations over the data set, e. Embedding vectors are initialized by the subroutine RANDOM-VECTOR(d), which returns a vector of length d with random values between 0 and 1. For each iteration and suspect set S, a random subset is generated by randomly including or excluding each $s_i \in S$ with probability 0.5. Gradients are then computed using Eq. 10 and used to update the embeddings.

It is worth noting that the subset sampling strategy RANDOM-SUBSET always produces subsets that are close to $\frac{|S|}{2}$ in size. Alternatively, one could set the probability of including a suspect differently for each training sample. This would allow the model to observe a greater variety of set sizes during training, which might allow it to perform better when predicting from a small S_{obs}. However, it was found empirically that a fixed inclusion probability of 0.5 performed equally well compared to a training procedure that selected a different inclusion probability between 0.25 and 0.75 for each sample. This may be attributable to the L_1-normalization step in the network, which greatly reduces sensitivity to the input set size.

VI. A DIRECTED SEARCH ALGORITHM USING SUSPECT PREDICTION

In this section we show how a suspect prediction model such as SIG or `suspect2vec` can be used to enhance the bug search procedure. One possible approach is to find a subset of suspects by running the search non-exhaustively, and then approximate the remainder of the solution set from the implied solutions [15]. This can greatly accelerate the suspect search process, but it sacrifices the formal guarantees of SAT–based debugging that all returned suspects are possible bug locations and that all possible bug locations are returned.

For applications that demand such formal guarantees, we propose a new SAT–based debugging algorithm that uses suspect prediction only to guess where solutions are most likely to be found, and then guides the search to prioritize these locations. The resulting algorithm remains sound and complete, but on average finds more suspects earlier in the search. This also means that within a given amount of time, more suspects will be available for downstream tasks such as detailed suspect analysis or triage.

One might naturally hope to achieve this behaviour by ordering the suspect variables by implication probability, $P(s_i|S_{\text{obs}})$, and searching for each individual suspect in turn. An individual suspect could be enforced, for example, by adding an assumption literal of the form s_i to the CNF formula, which would constrain the solver to only consider solutions in which the variable s_i is activated. However, it was observed empirically that such an approach can severely slow down the overall search in many cases. This is because in order to proceed to the next suspect, any constraints that forced the previous suspect must be removed. This can be highly detrimental to the incremental performance of modern SAT solvers because removing constraints often forces learned clauses to be invalidated. Instead, we propose an approach which only infrequently requires the SAT solver to be reset.

A. Suspect Search in Multiple Passes

As a first step towards this goal, we explain how the SAT–based suspect search algorithm of [2] can be modified to process suspects or groups of suspects independently from one another. This is a key prerequisite to being able to prioritize certain candidate suspects above others. The idea is to partition the search space into multiple passes, with only a subset of the design locations examined in each pass. Letting P_j denote the set of locations examined during the j^{th} pass, the CNF formula $\Phi(P_j)$ is constructed as described in Section II-A for $N = 1$, but with error select logic only inserted at locations $l_i \in P_j$. $\Phi(P_j)$ can be searched for all satisfying assignments as usual, but the search space is greatly reduced.

Pseudocode for the full algorithm is given in Algorithm 3, which takes as input a circuit C, an error trace err, and parameters TL (the time limit) and split_factor (the degree of partitioning). The algorithm maintains a queue of passes to be run, which is initialized with a pass containing all design locations. The subroutine BUILD-CNF(C, err, P) constructs a CNF formula for the debugging problem defined by (C, err), but it only models design locations in P as potential error sources. Each pass is popped from the queue and searched for solutions by the subroutine SOLVE. Most crucially, this subroutine is only allowed to run for at most TL seconds, after which execution returns to line 9. This means that the set of returned solutions, S_P, may not be complete. Line 10 checks

Algorithm 3 MULTI-PASS-DEBUG(C, err, TL, split_factor)

```
1: S ← φ
2: P_init ← all locations in C
3: pass_queue ← empty queue
4: pass_queue.push(P_init)
5: while not pass_queue.empty() do
6:     P ← pass_queue.pop()
7:     Φ ← BUILD-CNF(C, err, P)
8:     S_P ← SOLVE(Φ, TL)
9:     S ← S ∪ S_P
10:    if SOLVE timed out and |P \ S_P| > 0 and |P| > 1 then
11:        m ← min(split_factor, |P \ S_P|)
12:        (P_1, ..., P_m) ← PARTITION(P \ S_P, m)
13:        for i = 1 to m do
14:            pass_queue.push(P_i)
15:        end for
16:    end if
17: end while
18: return S
```

whether this is the case, and if so, then the remaining candidate suspects $P \setminus S_P$ are partitioned uniformly into smaller passes $P_1, ..., P_m$, where $m = \min(\text{split\_factor}, |P \setminus S_P|)$. Each subpass is added to the queue. The process repeats until all passes and subpasses have been executed.

The benefit of MULTI-PASS-DEBUG is that the size of the CNF formula is dynamically adjusted based on difficulty. If the search cannot be completed within the time limit, then the problem is scaled down by reducing the number of suspects under consideration. This can greatly reduce the peak memory requirements and total runtime of the search process.

However, the method is not without tradeoffs. In some cases, interrupting the search to partition the pass can cause performance to deteriorate, particularly if TL is set too low, because it resets the SAT solver and abandons any progress that has been made up until the time limit. Furthermore, if TL is set lower than the time required to find a single suspect, then MULTI-PASS-DEBUG is not guaranteed to find all solutions. Theoretically, this issue can be solved by allowing unlimited processing time when $|P| = 1$. Despite these potential drawbacks, we show in Section VII that MULTI-PASS-DEBUG frequently outperforms single–pass debugging on difficult instances.

B. Implication–Guided Suspect Search

The greatest advantage of MULTI-PASS-DEBUG is that it lends itself more easily to guidance from suspect implications. Because passes are independent of one another, the order in which candidate suspects are examined can easily be controlled. If this order follows Rank(S_{obs}) from Eq. 4 or 8, then, assuming a reasonably accurate ranking, most of the design locations examined in the initial passes will be solutions, while non-solution locations will be deferred to later passes. As a result, more suspects will be available earlier.

In MULTI-PASS-DEBUG, candidate suspects are ordered randomly; therefore, any ranking that is more accurate than a random ordering should improve performance. However, we expect that more accurate rankings should lead to greater improvements. Intuitively, and as shown empirically in Section VII, the suspect ranking techniques tend to be more accurate when more known suspects are given (*i.e.* S_{obs} is larger). This suggests that to maximize performance, the global suspect ranking should be updated regularly during the search so as to always use the most up-to-date S_{obs}.

Algorithm 4 IMPLICATION-GUIDED-DEBUG(C, err, TL, split_factor)

```
1: S ← φ
2: Pool_1 ← all locations in C
3: Pool_i ← empty array for 2 ≤ i < ∞
4: n_1 ← 1
5: n_i ← 0 for all 2 ≤ i < ∞
6: for lv = 1 to ∞ do
7:     if |Pool_lv| = 0 then
8:         return S
9:     end if
10:    while n_lv ≥ 1 do
11:        RANK-SUSPECTS(Pool_lv, S)
12:        P ← FIRST-PARTITION(Pool_lv, n_lv)
13:        Φ ← BUILD-CNF(C, err, P)
14:        S_P ← SOLVE(Φ, TL)
15:        S ← S ∪ S_P
16:        if SOLVE timed out and |P| > 1 then
17:            Pool_lv+1 ← Pool_lv+1 ∪ S \ P
18:            n_lv+1 ← n_lv+1 + min(split_factor, |P \ S_P|)
19:        end if
20:        Pool_lv ← Pool_lv \ P
21:        n_lv ← n_lv − 1
22:    end while
23: end for
```

Unfortunately, this idea is at odds with the initial idea of storing pending candidate suspects and passes in a queue, because once candidate suspects are inserted into passes and pushed onto the queue, their ordering is fixed. If more suspects are found afterward and S_{obs} is updated, all pending candidate suspects should be re-ordered and re-partitioned into a new set of passes. Therefore, we present a reformulation of the multi-pass search strategy which builds passes no earlier than they are needed; this allows us to use the most accurate possible ranking when choosing the next candidate suspects to examine.

The new algorithm, named IMPLICATION-GUIDED-DEBUG, is described in Algorithm 4. The algorithm processes candidate suspects in levels, beginning with level 1. Pool_{lv} stores the candidate suspects to be processed in level lv. Within each level, Pool_{lv} is partitioned into multiple passes. Candidate suspects which cannot be searched within a time limit of TL are added to $\text{Pool}_{\text{lv}+1}$ to be re-examined later. The algorithm terminates in line 8 once no candidate suspects remain.

The partitioning of Pool_{lv} is controlled by the parameter n_{lv} and the subroutine RANK-SUSPECTS. n_{lv} is the number of passes that Pool_{lv} is partitioned into, and it is set so that higher levels examine fewer candidate suspects per pass. n_{lv} is approximately increased by a factor of split_factor for each level, with adjustments made in line 18 in case fewer than split_factor candidate suspects remain.

The prioritization of candidate suspects is governed by the subroutine RANK-SUSPECTS. Given the set of discovered solutions S, RANK-SUSPECTS(Pool_{lv}, S) computes Rank($\text{Pool}_{\text{lv}}|S$) using either Eq. 4 or Eq. 8 and sorts Pool_{lv} accordingly. This is done immediately before each pass is built. In the pseudocode, the partitioning itself is performed in line 12 by

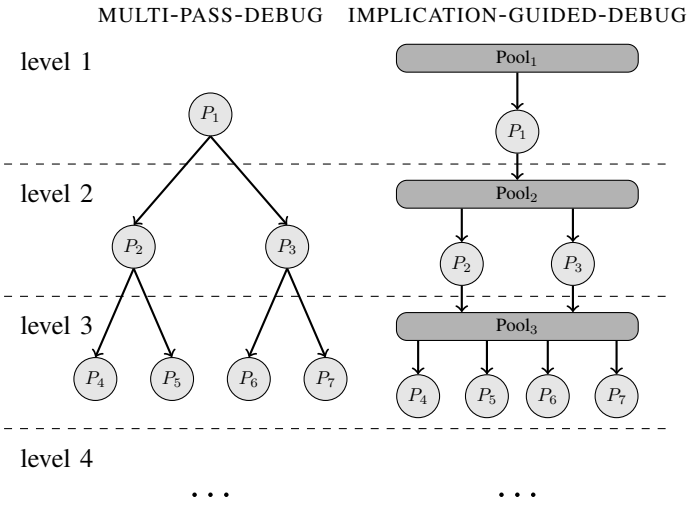


Fig. 6. Suspect processing flow for MUTLI-PASS-DEBUG (left) and IMPLICATION-GUIDED-DEBUG (right) with split_factor = 2. Passes are solved in numerical order (P_1, P_2, P_3, etc.).

the subroutine FIRST-PARTITION(Pool_{lv}, n_{lv}), which returns the first partition of Pool_{lv} out of n_{lv} equally–sized partitions. n_{lv} is then decremented so that each pass at a given level will have approximately the same size.

The connection between MULTI-PASS-DEBUG and IMPLICATION-GUIDED-DEBUG is apparent in Figure 6, which illustrates the flow of suspect processing in each algorithm. Despite their apparent disparities when expressed in pseudocode, both algorithms operate in a similar manner. The key distinction is that IMPLICATION-GUIDED-DEBUG accumulates all candidate suspects at each level and reorders them before partitioning them into passes. This pooling strategy offers two sources of improvement over the queuing strategy of MULTI-PASS-DEBUG: it uses the most up-to-date S_{obs} before committing to a suspect ranking, and it balances the suspects more uniformly across passes within the same level.

VII. EXPERIMENTAL RESULTS

In this section we evaluate the proposed methodologies using a large and diverse set of failures from several benchmark designs. We begin with an evaluation of SIG and `suspect2vec` for estimating suspect implications, rankings, and set prediction. We then evaluate the proposed IMPLICATION-GUIDED-DEBUG algorithm and compare it against MULTI-PASS-DEBUG and a single-pass baseline.

A. Data Set

Our data set consists of eight benchmark designs obtained from Opencores [26], Titan23 [27], and an industry partner. The designs and their sizes are listed in Table I.

For each design, a number of bugs were injected by randomly selecting and corrupting a fragment of the HDL code. Four types of such bugs were created, as listed below.

1) Assignment bug: randomly replace a signal assignment statement with an assignment to 0 or 1.
2) Incorrect condition bug: randomly replace a conditional statement of the form `if (<expression>)` with `if (1)`.
3) Incorrect operator bug: randomly replace a binary operator with a different (but semantically valid) operator.
4) Missing port connection bug: remove a port connection in a module instantiation.

In addition, we manually created several bugs for each design in order to improve the diversity and representativeness of the data set. These bugs include missing pipeline stages, incorrect state transitions, bad stimulus, and more complex corruptions of logical and arithmetic expressions. The breakdown of bugs by type is also given in Table I. The relative frequencies of bug types in the data set approximately reflects the relative difficulty of generating bugs of each type.

Each buggy design was then simulated, and one or more failures was identified for each bug. Failures under consideration include assertion errors and incorrect values on the primary output signals when compared to the golden design. The resulting number of failures is given in Column 9 of Table I. SAT–based debugging was performed at the RTL level on each failure to obtain the complete suspect sets.

Each benchmark design is considered independently in all experiments. In particular, training is performed separately because data from one design is not helpful to understanding suspect relationships in a different design.

B. Suspect Ranking and Set Prediction

In this section we evaluate the two proposed methods for computing probabilistic suspect implications: SIG and `suspect2vec`. Both methods are evaluated on the tasks of suspect ranking and suspect set prediction. In essence, suspect ranking considers the *relative* strengths of suspect implications, whereas suspect set prediction considers the *absolute* implication strengths (*i.e.*, whether or not each suspect is implied). Suspect ranking is of particular importance for the IMPLICATION-GUIDED-DEBUG algorithm, while suspect set prediction can be useful in itself for approximate debugging.

Letting $\text{Pred}(S_{\text{obs}})$ and S denote predicted and true suspect sets, respectively, the prediction quality is characterized by three key metrics: precision, recall, and F_1 score. Precision is defined as the fraction of predicted suspects which are correct:

$$\text{Prec} = \frac{|\text{Pred}(S_{\text{obs}}) \cap S|}{|\text{Pred}(S_{\text{obs}})|}$$

Recall is defined as the fraction of correct suspects which are predicted:

$$\text{Rec} = \frac{|\text{Pred}(S_{\text{obs}}) \cap S|}{|S|}$$

F_1 score is defined as the harmonic mean of precision and recall, and it provides a metric which balances the two:

$$F_1 = \frac{2}{\frac{1}{\text{Prec}} + \frac{1}{\text{Rec}}} = \frac{2|\text{Pred}(S_{\text{obs}}) \cap S|}{|S| + |\text{Pred}(S_{\text{obs}})|}$$

To characterize the quality of a suspect ranking, we measure the area under the precision versus recall curve (AUC-PR), defined as follows. Let $s_{r_1}, ..., s_{r_n}$ denote the suspect ranking. Then,

$$\text{AUC-PR} = \sum_{i=1}^{n} \text{Prec}(i) \times (\text{Rec}(i) - \text{Rec}(i-1))$$

TABLE I. DATA SET CHARACTERISTICS

Design	Gates	Total bugs	Assignment bugs	Incorrect condition bugs	Incorrect operator bugs	Missing port connection bugs	Manual bugs	Failures
ethernet	82803	24	14	3	3	0	4	80
fdct	546878	32	20	2	5	0	5	35
mips789	55248	40	22	3	5	5	5	79
scam_core	1315446	19	8	1	3	0	7	69
smoac_core	879920	30	11	3	7	2	7	61
sudoku_check	649819	30	9	2	6	2	11	65
vga	44579	44	20	2	5	4	13	52
wb_dma	222302	29	9	3	6	2	9	39

where

$$\text{Prec}(i) = \frac{|S \cap \{s_{r_1}, ..., s_{r_i}\}|}{|S|}$$
$$\text{Rec}(i) = \frac{|S \cap \{s_{r_1}, ..., s_{r_i}\}|}{i}$$

Note that an optimal ranking, which places all solution suspects before all non-solutions, has an AUC-PR of 1.0.

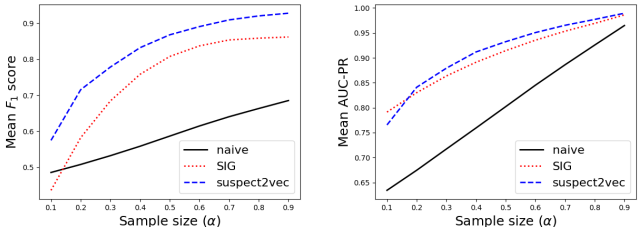
Both methods are compared against a baseline method named "naive" which ranks suspects by non-increasing count(s_i); that is, suspects are ranked higher when they occur more frequently in the training data. To predict a set, the naive method chooses the set size k to be the median of set sizes in the historical data. This serves as a lower bound which any useful method should outperform.

All experiments are performed using the *leave-one-out* methodology. That is, for each suspect set $S_i \in \mathcal{S}_H$, we use $\mathcal{S}_H \setminus \{S_i\}$ as training data and S_i as the test point. We then take the mean or median over all i. To test on S_i, we select the observed subset $S_{\text{obs},i} \subseteq S_i$ to be the first $\alpha|S_i|$ suspects found by the SAT search on failure F_i, where α ($0 \leq \alpha \leq 1$) is the sample size. This replicates that subset that would be obtained in the IMPLICATION-GUIDED-DEBUG algorithm.

Hyperparameter choices for SIG include a prior variance of 0.2 and a smoothing width of $\delta = \frac{|SU|}{50}$ for the termination point. For `suspect2vec` we use an embedding dimensionality of $d = 20$, a learning rate of $\eta = 0.01$, and $e = 4000$ training epochs. We choose a small dimensionality so as to limit the representational capacity and avoid overfitting to the small data sets under consideration here.

1) Detailed Results at 50% Sample: Table II gives the results for each design with a sample size of $\alpha = 0.5$. We compare the SIG and `suspect2vec` (s2v) methods against the baseline (naive) for the metrics of precision, recall, F_1 score, and AUC-PR. It is clear that both SIG and `suspect2vec` perform significantly better than naive in all metrics and across all designs. `Suspect2vec` performs best of all in most cases, although SIG sometimes achieves greater precision.

To better understand these results, columns 14-16 report the relative error in the estimated set size (k), defined as $\frac{|k-|S||}{|S|}$. We take the median over all $S \in \mathcal{S}_H$ rather than the mean for this metric due to the presence of some extreme outliers. The results suggest that in SIG, poor estimation of k is the main impediment to achieving a high F_1 score. By doing away with the termination point estimate and casting the problem in terms of binary classification, `suspect2vec` is able to estimate k much more accurately. This is also reflected in the AUC-PR metric, which is agnostic to the set size estimate and only measures the ranking quality. Here the performance of SIG is much closer to that of `suspect2vec` in most cases.



(a) F_1 score (b) AUC-PR

Fig. 7. F_1 score and AUC-PR for different sample sizes. The mean is taken over all designs.

2) Effect of Sample Size: We now investigate how suspect prediction performs when given suspect subsets of different sizes. Intuitively, we expect that performance should improve with larger samples, as larger samples are more informative and characterize the failure more precisely. Additionally, with a sample size of α, $\alpha|S|$ suspects are given; in our experiments the given suspects are always ranked first regardless of how the prediction model would score them.

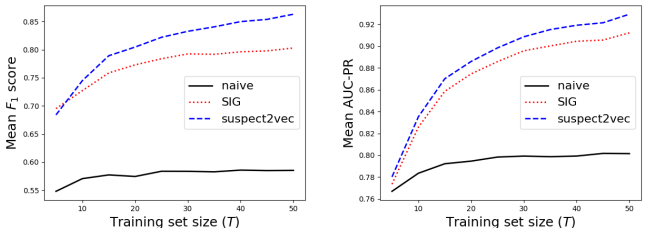
Figure 7 plots the F_1 score and the AUC-PR for each method against α, ranging from 0.1 to 0.9 in increments of 0.1. The trend with increasing α matches our expectation. The naive method also improves with increasing α, even though it does not make use of the sample to guide its output, simply because more suspects are given with larger α. Both SIG and `suspect2vec` significantly outperform naive at all sample sizes. We also observe that for very small α, the gap between SIG and `suspect2vec` widens in F_1 score, while it narrows in AUC-PR. This further confirms that the two methods are comparable for the suspect ranking task, but SIG struggles to estimate the size of the suspect set.

3) Effect of Training Set Size: We aim to determine how much training data each of the proposed suspect prediction methods requires. Let $\mathcal{S}_H$ denote the complete data set for a design, and let T denote the size of the training set used. In this experiment, for each $S_i \in \mathcal{S}_H$, we build the training set by choosing $\min(T, |\mathcal{S}_H \setminus S_i|)$ items randomly from $\mathcal{S}_H \setminus S_i$. Figure 8 plots the F_1 scores and AUC-PR values with T varying from 5 to 50 in increments of 5. A sample size of $\alpha = 0.5$ is used. Again, the mean is taken over all failures and all designs.

Unsurprisingly, the average performance consistently improves with more training data, with a greater rate of increase seen at small T. Both methods begin to reach very good performance at approximately 20-30 training instances. If only a small amount of data is available, then it is important to note that the performance gap between SIG and `suspect2vec` narrows to essentially zero at very small T. This is consistent with well–established literature on neural networks, which has

TABLE II. SUSPECT SET PREDICTION RESULTS AT $\alpha = 0.5$ FOR NAIVE, SIG, AND SUSPECT2VEC (S2V)

Design	Mean precision			Mean recall			Mean F_1 score			Mean AUC-PR			Median set cardinality error		
	naive	SIG	s2v	naive	SIG	s2v	naive	SIG	s2v	naive	SIG	s2v	naive	SIG	s2v
ethernet	0.625	0.832	**0.930**	0.582	0.864	**0.924**	0.521	0.827	**0.917**	0.765	0.951	**0.966**	0.476	0.207	**0.038**
fdct	0.733	0.823	**0.972**	0.725	**0.879**	0.863	0.651	0.842	**0.909**	0.891	0.932	**0.937**	0.606	**0.105**	0.135
mips789	0.704	**0.904**	0.890	0.651	0.731	**0.797**	0.623	0.803	**0.827**	0.821	0.907	**0.927**	0.369	0.220	**0.173**
scam_core	0.713	**0.953**	0.914	0.774	0.713	**0.921**	0.704	0.808	**0.910**	0.874	0.948	**0.963**	0.223	0.284	**0.051**
smoac_core	0.597	0.824	**0.887**	0.596	0.832	**0.848**	0.529	0.814	**0.859**	0.729	0.901	**0.916**	0.458	0.146	**0.080**
sudoku_check	0.702	0.763	**0.946**	0.552	0.857	**0.876**	0.553	0.787	**0.901**	0.772	0.931	**0.942**	0.468	0.269	**0.071**
vga	0.659	0.822	**0.846**	0.585	0.781	**0.840**	0.538	0.788	**0.827**	0.758	0.890	**0.914**	0.569	0.206	**0.180**
wb_dma	0.671	**0.830**	0.799	0.665	0.737	**0.844**	0.566	0.771	**0.797**	0.811	0.867	**0.903**	0.682	0.168	**0.131**
mean	0.676	0.844	**0.898**	0.641	0.799	**0.864**	0.586	0.805	**0.868**	0.803	0.916	**0.934**	0.481	0.201	**0.107**



(a) F_1 score (b) AUC-PR

Fig. 8. F_1 score and AUC-PR for amounts of training data. The mean is taken over all designs.

shown that they generally require large amounts of data in order to outperform statistical methods.

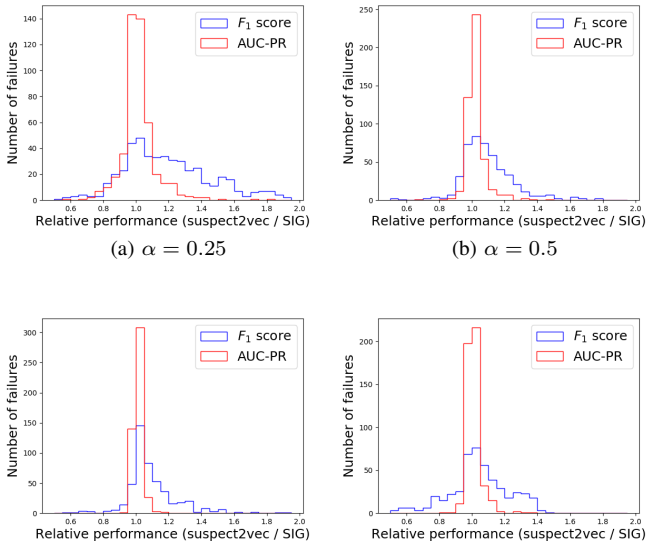
4) Discussion: Our results show that both SIG and suspect2vec are effective methods for computing probabilistic suspect implications and predicting suspect locations. For predicting suspect sets, suspect2vec consistently outperforms SIG on average due to its superior ability to estimate the set size. However, on a case-by-case basis results are much more varied, and in many cases SIG performs better than suspect2vec.

This is shown in Figure 9, which plots the distributions of the relative performance of the two methods. Specifically, for each failure in the data set we compute the ratio of the suspect2vec F_1 score to the SIG F_1 score. We then plot these points in a histogram. We do the same with the AUC-PR metric. Note that the figure includes all designs in aggregate.

Figures 9 (a), (b), and (c) show the distributions for $\alpha = 0.25$, 0.5, and 0.75, respectively, while Figure 9 (d) shows the distribution for a small training set ($T = 10$). For each of these experiments, Table III gives the number of test instances in which each method performs better. While suspect2vec performs better in the majority of cases, there are still many instances in which SIG performs better, especially with small amounts of training data. We also observe that the distribution is much more narrow for AUC-PR, indicating that performance is very similar in this metric.

These experiments conclusively show that suspect2vec offers superior performance over SIG for set prediction when a large data set is available, but relatively little advantage in other scenarios. Moreover, SIG may be easier to use due to its simplicity and intuitiveness, whereas the operation of suspect2vec is generally not interpretable by humans.

SIG is also much less sensitive to hyperparameter settings than suspect2vec. SIG requires only the prior variance σ^2, and — for set prediction only — the smoothing width δ, neither of which have a dramatic impact on overall performance. In contrast, during our experimentation with suspect2vec



(a) $\alpha = 0.25$ (b) $\alpha = 0.5$

(c) $\alpha = 0.75$ (d) $T = 10$ ($\alpha = 0.5$)

Fig. 9. Distribution of relative F_1 scores and AUC-PRs of suspect2vec versus SIG

TABLE III. COMPARISON OF SIG AND SUSPECT2VEC BY NUMBER OF FAILURES WITH BETTER PERFORMANCE (WINS).

Experiment	F_1 score		AUC-PR	
	SIG wins	s2v wins	SIG wins	s2v wins
$\alpha = 0.25$	118	362	217	263
$\alpha = 0.5$	128	352	151	329
$\alpha = 0.75$	84	396	141	339
$T = 10$ ($\alpha = 0.5$)	193	287	211	269

we observed that results can vary considerably with different values of the hyperparameters d, e, and η, and it is only with careful tuning that it is able to outperform SIG. Finally, the training procedure for SIG is generally more computationally efficient than that of suspect2vec, although training time depends on multiple factors including design size, training data size, and hyperparameter settings. To produce the results presented in this section, training of both methods required less than one minute in all cases.

C. Suspect Search Algorithm

In this section we evaluate the proposed bug search algorithm IMPLICATION-GUIDED-DEBUG, which uses suspect implications to prioritize suspect candidates that are most likely to be solutions. We compare the algorithm against the baseline SAT–based debugging algorithm as described in [2]. We also compare against the MULTI-PASS-DEBUG algorithm in order to isolate the effects of dividing the SAT search into multiple passes and of guiding the search using suspect

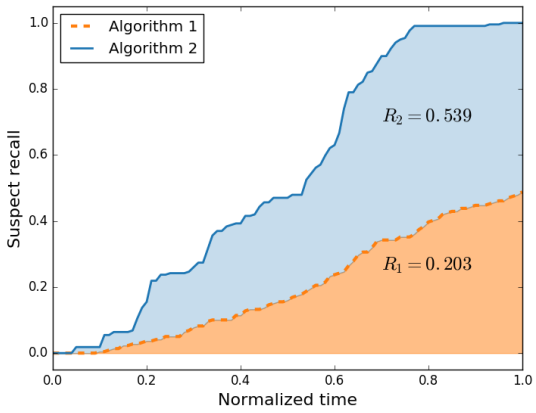


Fig. 10. Recall–time curves for two algorithms on a failure from the `wb_dma` design.

implications. All debugging algorithms use MiniSat [28] as the backend SAT solver.

We focus only on difficult debug instances, so all experiments in this section include only instances with a runtime of at least 15 minutes. The resulting number of failures is given in column 2 of Table IV. However, for training the suspect prediction models, all failures (other than the test failure) are included in the training data. Each failure and debugging algorithm is run with a time limit of 3 hours. All experiments are run on a i5-3570K 3.4 GHz machine with 16 GB of RAM.

1) Evaluation Methodology: Unlike prior work on SAT–based debugging, our primary objective is not necessarily to reduce the overall runtime, but to improve the anytime behaviour of the algorithm by returning more solutions in the early stages of the search. This would allow for tasks such as detailed suspect analysis, triage, or design rewiring, which require many bug suspects, to begin sooner. To quantify this property of an algorithm, we introduce the metric of *average suspect recall*, denoted by R.

Consider the suspect recall (fraction of solutions found) at each point in time over the execution of the search. A plot of this value is shown in Figure 10 for two different algorithms on a failure from the `wb_dma` design. The time axis is normalized to range from 0 to 1. Intuitively, the more desirable algorithm has a recall–time curve that approaches 1.0 as early as possible. This is captured by the area under the curve, or equivalently, the average value of the curve (because the time axis is normalized).

Formally, let t_i be the time at which the i^{th} solution is found for a failure with suspect set S, and let T denote the total runtime. Then the average suspect recall is defined as:

$$R = \sum_{i=1}^{|S|-1} \frac{i}{|S|} \frac{t_{i+1} - t_i}{T} + \frac{T - t_n}{T} \tag{11}$$

Figure 10 shows the area under the recall–time curves and the values of R. In this example, the second algorithm is better by a factor of $\frac{R_2}{R_1} = 2.65$.

We compute this metric for each algorithm and take the ratio versus the baseline algorithm to obtain the relative improvement in R. We then take the geometric mean over all debug instances in a design. In our experiments with MULTI-PASS-DEBUG and IMPLICATION-GUIDED-DEBUG we use a pass time limit (TL in Algorithms 3 and 4) of 300 seconds and split_factor $= 10$. To compute suspect implications, training is performed in a leave-one-out manner, with $\mathcal{S}_H \setminus S_i$ used as the training data for failure F_i.

TABLE IV. GEOMETRIC MEAN RELATIVE IMPROVEMENT IN AVERAGE SUSPECT RECALL FOR DEBUGGING ALGORITHMS

Design	Num failures	Base	MPD	IGD + SIG	IGD + s2v	IGD + opt
`ethernet`	5	1.0	1.16	**2.06**	1.91	1.36
`fdct`	7	1.0	1.89	2.81	**2.99**	2.33
`mips789`	11	1.0	1.46	1.61	1.65	**1.76**
`scam_core`	11	1.0	0.40	0.97	**1.02**	0.72
`smoac_core`	31	1.0	1.27	**2.25**	2.16	1.83
`sudoku_check`	7	1.0	0.93	0.84	**1.01**	0.85
`vga`	11	1.0	0.98	**1.15**	1.13	1.13
`wb_dma`	23	1.0	3.08	4.47	**5.36**	4.79
geomean	—	1.0	1.21	1.75	**1.83**	1.55

2) MULTI-PASS-DEBUG *Results:* Table IV gives the results for several algorithms: the baseline as described in [2], which has a relative R of 1.0 by definition (column 3), MULTI-PASS-DEBUG (MPD, column 4), IMPLICATION-GUIDED-DEBUG with implications computed by SIG (IGD + SIG, column 5), and IMPLICATION-GUIDED-DEBUG with implications computed by `suspect2vec` (IGD + s2v, column 6).

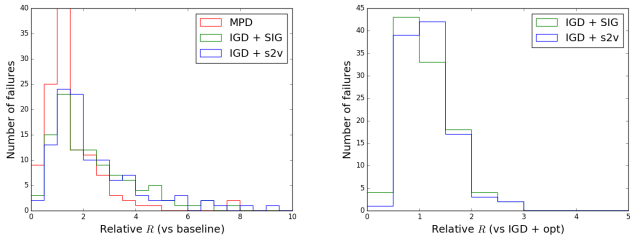
In most cases, MULTI-PASS-DEBUG outperforms the baseline, although there exist cases in which it does not, such as the `scam_core` and `sudoku_check` designs. As discussed in Section VI-B, setting TL too low can cause wasted effort during the SAT search, and estimating the optimal TL on a case-by-case basis is not feasible. Nonetheless, MULTI-PASS-DEBUG does not hinder performance on the average case, allowing for greater improvements using suspect implications.

A notable exception is the `scam_core` design, for which MULTI-PASS-DEBUG impairs performance by more than 2x. The reason for this result is that, because this design is so large, the error traces had to be truncated considerably in order to satisfy memory constraints. Most of the resulting debug instances could then be solved very quickly. This is exacerbated by the fact that solutions are highly abundant for this design and therefore easy to find. Thus, these test cases exhibit highly unfavorable conditions for multi-pass debugging with pass time limit of 300 seconds, which is best suited for longer–running debug instances with difficult-to-find solutions. Had the memory constraints allowed for longer error traces, we expect that the results would have greatly improved.

3) IMPLICATION-GUIDED-DEBUG *Results:* IMPLICATION-GUIDED-DEBUG shows consistent performance gains over both MULTI-PASS-DEBUG and the baseline when implemented with both SIG and `suspect2vec`. In particular, in `scam_core` the incorporation of guidance from suspect implications improves performance by more than 2x, which compensates for the loss caused by MULTI-PASS-DEBUG.

Figure 11 (a) shows the distributions of R for each algorithm relative to the baseline. The majority of instances see relatively modest improvements between 1.0 and 1.5, while some instances see improvements greater than 5x. There are also some instances which perform worse than the baseline, primarily due to the splitting of the search into multiple passes. This is evidenced by the distribution for MULTI-PASS-DEBUG (MPD), which has a significant number of instances below 1.0. Nonetheless, all designs see an overall positive mean improvement with IMPLICATION-GUIDED-DEBUG.

To better understand the behaviour of IMPLICATION-GUIDED-DEBUG, we also run it with suspect implications



(a) Relative to baseline (b) Relative to IGD + opt

Fig. 11. Distribution of average suspect recall

computed *optimally*; that is, with solution suspects always ranked ahead of non-solution suspects. While this algorithm is not realizable in practice, it serves as a useful reference point as it allows us to assess the impact of incorrect predictions by SIG and `suspect2vec` on overall performance. The results are given in column 7 of Table IV (IGD + opt).

Interestingly, in many cases IGD + opt performs worse than IGD + SIG or IGD + s2v. Thus, better prediction does not necessarily correspond to better average suspect recall. This is more apparent in Figure 11 (b), which plots the distribution of R for IGD + s2v and IGD + SIG relative to IGD + opt. While most of the results lie between 0.5 and 1.0, a significant amount are above 1.0.

Upon closer examination of the operation of each algorithm, we found that this apparent anomaly is caused by differences in ranking of the correct (solution) suspects. In some cases, IGD + opt builds passes containing suspects whose solutions are very difficult to find, causing the SAT solver to lose time on these passes early in the search. In other cases, IGD + opt performs better than IGD + SIG and IGD + s2v for the opposite reason. There does not appear to be any relationship between the strength of a suspect implication and the difficulty of finding the SAT solution. Because we rank suspects by the former rather than the latter, results vary between different ranking methods.

The final row of Table IV gives the geometric mean across all designs. Despite the variability, MULTI-PASS-DEBUG improves over the baseline by 20%, while IMPLICATION-GUIDED-DEBUG can improve by as much as 83%, depending on which prediction method is used.

4) Runtimes: Table V compares the debugging algorithms in terms of total runtime rather than R. Columns 2-5 give the number of instances completed within the time limit of 3 hours for each design. Columns 6-8 give the geometric mean runtimes relative to the runtime of the baseline algorithm, including only the instances which finished within the time limit. In many cases, MULTI-PASS-DEBUG and IMPLICATION-GUIDED-DEBUG complete significantly more instances than the baseline. Furthermore, IMPLICATION-GUIDED-DEBUG has a lower mean runtime than MULTI-PASS-DEBUG on all benchmarks. This appears to be a result of building passes dynamically from a suspect pool rather than maintaining a pass queue, because the former balances the number of suspects per pass more evenly. Note that because this only includes instances that were completed by the baseline and the new algorithms, for many designs the actual relative runtime is likely significantly lower than the numbers shown here.

Considering the overall debugging process – including both the training of the prediction models and the SAT search – the runtime is vastly dominated by the SAT search. Because the training scales polynomially with the size of the design while the SAT search is exponential in the worst case, the additional cost of incorporating suspect prediction is very small.

TABLE V. NUMBER OF DEBUG INSTANCES COMPLETED AND MEAN RELATIVE RUNTIMES

Design	Instances completed				Relative runtime		
	Base	MPD	IGD + SIG	IGD + s2v	MPD	IGD + SIG	IGD + s2v
`ethernet`	5	5	4	5	1.36	0.56	0.84
`fdct`	5	7	7	7	0.71	0.53	0.41
`mips789`	5	11	9	11	0.93	0.63	0.71
`scam_core`	11	11	11	11	1.66	1.13	1.17
`smoac_core`	24	30	31	31	0.72	0.47	0.47
`sudoku_check`	7	7	6	6	1.54	1.24	1.10
`vga`	5	8	10	9	1.16	0.68	0.74
`wb_dma`	6	23	23	23	0.48	0.34	0.31

VIII. CONCLUSION

This paper studies the novel concept of probabilistic suspect implications and their application to suspect set prediction and SAT–based debugging. Two methods are proposed to compute probabilistic suspect implications from historical debug data. The first method, named SIG, uses belief propagation on a probabilistic graph to score candidate suspects by their likelihood of being solutions. The second method, named `suspect2vec`, instead classifies candidate suspects as solutions or non-solutions with a single-hidden-layer neural network. This allows it to outperform SIG on average, particularly in the set prediction task, at the expense of larger data requirements and hyperparameter tuning. We then propose a new SAT–based debugging algorithm that can be guided to prioritize areas of the search space that are more likely to contain solutions. When guided by SIG or `suspect2vec`, this algorithm is able to find most suspects earlier in the search, allowing further tasks such as detailed suspect analysis or failure triage to begin sooner and accelerating the overall debugging process.

As future work, it would be interesting to analyze the connection between the probabilistic suspect implications studied here and implications by structural dominance as studied in [14]. In particular, both techniques can enhance the search procedure in different and complementary ways, so a hybrid algorithm that incorporates both may perform better than either individually.

REFERENCES

[1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

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

[3] K.-h. Chang, I. Wagner, V. Bertacco, and I. L. Markov, "Automatic error diagnosis and correction for RTL designs," in *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*. IEEE, 2007, pp. 65–72.

[4] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1597–1608, 2009.

[5] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *tcad*, vol. 29, no. 11, pp. 1790–1803, 2010.

[6] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*. ACM, 2008, pp. 77–82.

[7] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated design debugging with maximum satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 11, pp. 1804–1817, 2010.

[8] H. Mangassarian, A. Veneris, and M. Benedetti, "Robust QBF encodings for sequential circuits with applications to verification, debug, and test," *IEEE Transactions on Computers*, vol. 59, no. 7, pp. 981–994, 2010.

[9] A. Sulflow, G. Fey, C. Braunstein, U. Kuhne, and R. Drechsler, "Increasing the accuracy of SAT-based debugging," in *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2009, pp. 1326–1331.

[10] T.-Y. Jiang, C.-N. J. Liu, and J.-Y. Jou, "Accurate rank ordering of error candidates for efficient HDL design debugging," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 272–284, 2009.

[11] A. Veneris and M. S. Abadir, "Design rewiring using ATPG," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1469–1479, 2002.

[12] Z. Poulos and A. Veneris, "Failure triage in RTL regression verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1893–1906, 2018.

[13] R. Berryhill and A. Veneris, "Methodologies for diagnosis of unreachable states via property directed reachability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 6, pp. 1298–1311, 2018.

[14] H. Mangassarian, B. Le, and A. Veneris, "Debugging RTL using structural dominance," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 1, pp. 153–166, 2014.

[15] N. Veira, Z. Poulos, and A. Veneris, "Suspect set prediction in RTL bug hunting," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 1544–1549.

[16] ——, "Suspect2vec: a suspect prediction model for directed RTL debugging," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 681–686.

[17] N. K. Jha and S. Gupta, *Testing of digital systems*. Cambridge University Press, 2003.

[18] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup, "Dominators in linear time," *SIAM Journal on Computing*, vol. 28, no. 6, pp. 2117–2132, 1999.

[19] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*. ACM, 1987, pp. 502–508.

[20] T. Niermann and J. H. Patel, "Hitec: A test generation package for sequential circuits," in *Proceedings of the conference on European design automation*. IEEE Computer Society Press, 1991, pp. 214–218.

[21] K. Murphy, *Machine Learning: a probabilistic perspective*. MIT Press, 2012.

[22] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 137–146.

[23] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[24] M. Sahlgren, "The distributional hypothesis," *Italian Journal of Disability Studies*, vol. 20, pp. 33–53, 2008.

[25] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning*, 2014, pp. 1188–1196.

[26] OpenCores.org, "http://www.opencores.org," 2006.

[27] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD," in *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 2013, pp. 1–8.

[28] N. Eén and N. Sörensson, "An extensible SAT-solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

Neil Veira (S'17) received a B.A.Sc. degree in Engineering Science with a major in Electrical and Computer Engineering from the University of Toronto in 2017. He then received a M.A.Sc. in Electrical and Computer Engineering from the University of Toronto in 2019. His research focus has been on applications of machine learning and data science techniques to hardware verification algorithms. He is currently with SoundHound Inc.

Zissis Poulos (S13-M'18) received a Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 2011, an M.A.Sc degree in Electrical and Computer Engineering from the University of Toronto in 2014, and a Ph.D. degree in Electrical and Computer Engineering from the University of Toronto in 2018. He is currently a Postdoctoral Fellow at Rotman School of Management at the University of Toronto. His research interests include applied machine learning in finance, deep learning acceleration, statistical diagnosis and debugging of VLSI systems, modeling and optimization of information/influence diffusion in social graphs, and distributed ledger technologies. He is a member of IEEE and ACM.

Andreas Veneris (S'96-M'99-SM'05) received a Diploma in Computer Engineering and Informatics from the University of Patras in 1991, an M.S. degree in Computer Science from the University of Southern California, Los Angeles in 1992 and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1998. In 1998 he was a visiting faculty at the University of Illinois until 1999 when he joined the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto where today he is a Professor. Since 2018 he is a Connaught Scholar for his contributions to blockchain technology. His research interests include CAD for debugging, verification, synthesis and test of digital circuits/systems, crypto-economics, decentralized blockchain technology, and combinatorics. He has received several teaching awards, a best paper award and a Ten Year Best Paper Retrospective Award. He is the author of one book and he holds several patents. He is a member of IEEE, ACM, AMS, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society.