Root-Cause Analysis for Memory-Locked Errors

John Adler [1], Djordje Maksimovic[1], Andreas Veneris[1,2]
[1]Department of Electrical and Computer Engineering, University of Toronto
[2]Department of Computer Science, University of Toronto
{adler, djordje, veneris}@eecg.toronto.edu

***Abstract*—Half of the time in the design cycle today is spent on verifying and debugging the correctness of a design. Although some debugging tasks have been automated, determining the root-cause of errors that have been locked in memory for a number of clock cycles before they propagate to an observation point remains a time consuming effort. This is because the error traces exposing such behavior can be excessively long, a fact that requires modeling the circuit for many time-frames. This paper introduces a performance-driven debugging methodology for pinpointing the root-cause of memory-locked errors. The technique models only a sliding time window and a final time window explicitly at any one time, while interstitial time-frames are linked with a lightweight memory model. This technique is later extended to a complete methodology that diagnoses errors that may be missed. Experiments on industrial designs with memory-locked errors demonstrate a 72% reduction in peak memory usage with a comparable runtime to existing methodologies.**

I. Introduction

The increasing design costs of modern Very Large Scale Integration (VLSI) systems drive the development of sophisticated Computer-Aided Design (CAD) tools. Verifying functional correctness, along with localizing and correcting error sources, consumes up to 70% of the chip design cycle [1], [2]. Debugging, a sub task of verification, comprises half of the total verification effort [3]. Disproportionate growth in the effort and cost of verification and debugging, coined as the verification gap, is projected to rise even further over the next few years [1].

SAT-based automated design debugging techniques [4], [5] have proven to be effective at decreasing the manual workload of debugging. However, ever-increasing design sizes coupled with long error trace lengths still limit their use. This is because the performance of these techniques deteriorates with increasing design size and length of the error trace. The problem posed by design size has been largely solved through the use of abstraction and refinement techniques [6]. On the other hand, recent research in Bounded Model Debugging (BMD) [7] has been orthogonally focused on reducing the memory usage as the length of the error trace increases. Nevertheless, these techniques also face an explosion in memory usage as the trace length size gets longer. This, combined with the use of abstraction to partially alleviate this problem may result in a large number of spurious solutions, increasing the debug iterations. This fact may render them ineffective for traces in which the erroneous behavior propagates beyond a few thousand clock cycles before being observed at some failure point.

In essence, the primary cause of excessive memory usage for long traces is the use of an iterative logic array (ILA) [5] to model the debugging instance. Modeling the problem using an ILA involves unrolling the sequential circuit, whereby the combinational part of the circuit is replicated k times to model the k clock cycles (time-frames for single-clock designs) of the given error trace. In an ILA representation, the current state assignment of time-frame $i + 1$ is connected to the next state assignment of time-frame i. This may present a problem in an industrial setting where the increasing use of large memory blocks [8] makes *memory-locked* errors a common occurrence. In detail, memory-locked errors are initially excited and propagate only to be written into a memory block. These erroneous values may be held for many clock cycles in memory before they exit and propagate to a point where the failure(s) is finally observed. As a result, the design may operate correctly for many clock cycles from the time the bug is excited and before it fails, generating an excessively long error trace that presents a challenge for conventional automated debugging techniques. This is because the clock cycles where the error is memory-locked still need to be modeled within the debugging instance.

To address the challenge of memory-locked errors, this paper presents a novel debugging technique that makes use of BMD along with a succinct memory model [9] to efficiently model long traces. The proposed technique models only two time windows at any one time: a sliding window and a final observation window. The error trace is split into non-overlapping time windows and for each iteration the sliding window is moved to model an earlier time window. The observation window remains constant as the last time window. The sliding window, observation window, and interstitial time windows are connected with a "lightweight" memory model. The technique originally proposed may miss errors that propagate across multiple time windows before entering a memory block. In the latter part of this paper, we extend it by adding an optimized nested iteration of BMD. The additional extended time windows are not connected to the observation window directly, but rather only to the sliding window.

An extensive set of experimental results on industrial designs with very long error traces from both `OpenCores` [10] and industrial partners showcases the benefits of this work. In detail, the proposed debugging technique achieves an average of 72% reduction in peak memory usage while it maintains a comparable runtime to the state-of-the-art debugging solutions.

The remainder of this paper is organized as follows. Section II provides the work background and prior art. Sections III and IV introduce the proposed debugging technique for long traces and its complete extension. Section V presents experimental results and Section VI concludes this work.

II. Preliminaries

A. SAT-based Design Debugging

When a design is verified it may expose a set of failures. To identify and correct the causes of these failures debugging is employed. Recent SAT-based debugging techniques [5] automate bug localization by finding potential error locations (*suspects*) *i.e.,* RTL components that may be erroneous. Each suspect can be excited and propagate error effects to the observation point resulting in the verification failure. To return suspects, SAT-based techniques require an error trace and a user-defined cardinality of errors. These parameters are then encoded into a SAT instance whose satisfying assignments indicate a set of suspects for the given cardinality N.

Specifically, the construction of the SAT instance begins with enhancing the combinational part of the circuit (*i.e.,* transition relation) denoted as T by adding an error model to each circuit location. The enhanced transition relation is denoted T_{en}. Each error model l allows the associated location to be either left unchanged, or replaced by a free variable by activating the associated suspect variable e_l. In that sense, activating the suspect variable allows the circuit location to model any function through the assignment of the free variable. To model the circuit behavior for $k + 1$ clock cycles an ILA is created by unrolling T_{en}, replicating it into $k+1$ time-frames. For each time-frame i, the vector of inputs X^i and the vector of expected outputs Y^i are constrained. Finally, the initial state S^0 and the cardinality $\Phi(N)$ are constrained. The equation that follows shows the mathematical representation of a debugging instance:

978-3-9815370-6-2/DATE16/ ©2016 EDAA

$$Debug_0^k(N) = S^0 \wedge \Phi(N) \wedge \bigwedge_{i=0}^{k} X^i \wedge Y^i \wedge T_{en}^i \quad (1)$$

B. Bounded Model Debugging

We observe from Eq. 1 that the size of the debug instance is proportional to the length of the debug window. Therefore, instead of modeling the entire error trace at once, BMD's window expansion [7] uses a debug window of the final w time-frames prior to the observation of a failure, known as a suffix window. The foundation for this technique lies in the fact that Eq. 1 can be generalized to model an arbitrary window of length w of the error trace starting at time-frame p, shown in Eq. 2 below:

$$Debug_p^{p+w-1}(N) = S^p \wedge \Phi(N) \wedge \bigwedge_{i=p}^{p+w-1} X^i \wedge Y^i \wedge T_{en}^i \quad (2)$$

Only errors that are excited within the suffix window will be found. To find all the suspects, the suffix is iteratively lengthened, as seen in Fig. 1. After each iteration, if the solver finds suspects on the initial states of the debugging instance, the suffix is lengthened and another iteration is started. If not, the algorithm terminates and returns all the suspects that were found. It has been shown that the benefit of reduced average peak memory usage of BMD, when compared to brute-force debugging with a complete error trace, can be offset by the increased runtime of multiple debug iterations [7]. In fact, in the worst case, the suffix window can expand to the complete trace where BMD degenerates to brute-force debugging.

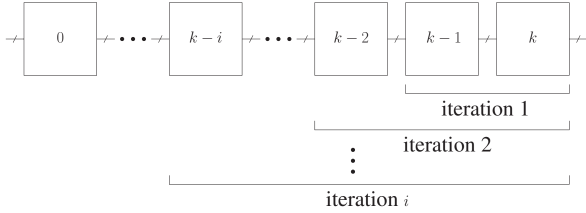

Fig. 1. BMD window expansion

C. Succinct Memory Model

While BMD's window expansion can alleviate memory usage for longer traces, designs containing memories need special consideration. The naïve approach to modeling memories by explicitly modeling each memory bit may cause an explosion in peak memory usage, discussed in [11]. The succinct memory model of [9] replaces the memory blocks in a design with a set of constraints that model forwarding lines. Intuitively, forwarding involves connecting a memory write to a memory read with a wire, as seen in Fig. 2. The wires and associated control logic are then modeled in the SAT instance.

The first set of forwarding lines are based on the assumption that the simulated value is correct, and forwards according to the error trace. For example, raising se_3 in Fig. 2 connects RD_3 to WD_1 and RA_3 to WA_1, allowing the data written into memory at time-frame 1 to be read at time-frame 3.

The second set of forwarding lines are based on the assumption that the simulated value is incorrect, and forwards according to a set of newly introduced buses that allow any time-frame to forward data to a subsequent time-frame. For example, raising $bre_{3,1}$ and $bwe_{1,1}$ in Fig. 2 creates a forwarding path between time-frame 1 and time-frame 3 through bus 1, allowing data to be forwarded according to the satisfying assignments found by the solver. The number of such bus lines B is specified by the user. Prior work [9] has shown that for practical purposes a value of $B \leq 3$ is sufficient.

Additionally to these forwarding constraints, constraints enforcing memory semantics are added to the debug instance. Memory semantics include behavior such as: a memory write must occur before a memory read and only one time-frame can write to a single bus line at a time. Since the original memory blocks are no longer needed with the inclusion of the succinct memory model, they can be removed from the circuit. This modified circuit is denoted $T_{en}^{'}$.

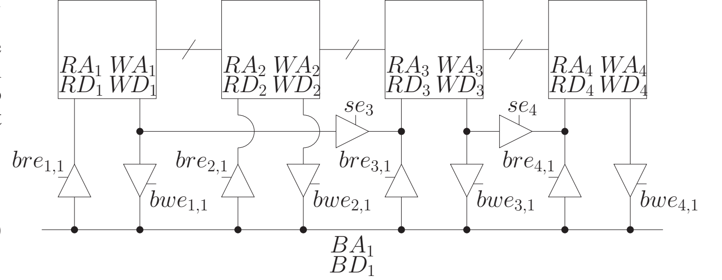

Fig. 2. Succinct memory model

III. Window Expansion for Memory-Locked Errors

As discussed in Section II, the size of the debugging SAT instance scales with the length of the debug window as the size of the ILA depends on the number of time-frames to be modeled. Because of this, when a trace exposes an error that remains locked in memory for many clock cycles, traditional debugging techniques may experience an explosion in memory usage. In this section, we present a novel design debugging technique specifically targeting memory-locked errors. The technique leverages BMD's window expansion [7] with a succinct memory model [9] to generate debugging instances that only explicitly model two fixed-length time windows at any one time.

The succinct memory model utilizes two sets of forwarding lines: *simulated lines* that forward values from the error trace and *bus lines* that forward corrected values or addresses different from the error trace. These lines will be used to connect time windows rather than the traditional state transition lines that exist between time-frames in the ILA. This will allow errors that propagate through memory rather than the remainder of the circuit to be found. For the purpose of this work, the cycle-accurate version of the succinct memory model is used, in which circuit values are known at all times in the error trace.

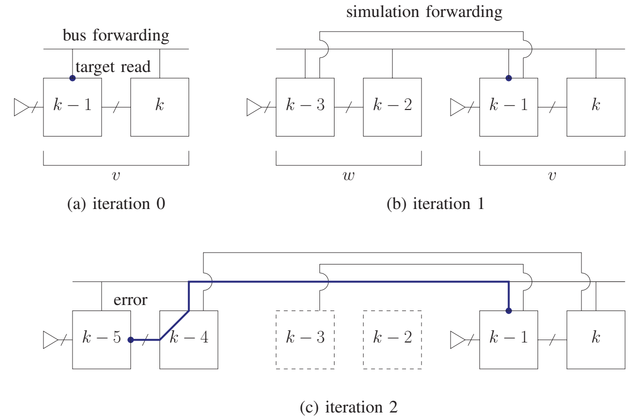

Fig. 3. Window expansion for memory-locked errors

For practical purposes, the proposed technique begins with a sanity check pass that is used to decide if a non-memory-locked error is the potential root cause of the observed failure. Such errors can be

ruled out using traditional debugging techniques [4], [5], [7] before running window expansion for memory-locked errors. For a suffix window of length v, called the *observation window*, a debugging pass using existing techniques is performed, as shown in Fig. 3(a). The results are then manually analyzed in an attempt to trace the erroneous behavior back to an incorrect memory read, called the *target read*. If a non-memory-locked error is determined to be at least partially responsible for the erroneous behavior, it can be fixed and the sanity check pass repeated until either the circuit behaves correctly or a target read is found.

The presence of such a target read gives a high level of confidence that a memory-locked error is indeed the root cause of the observed failure, assuming the memory block itself is correct. The latter is a realistic assumption since memories can generally be verified independently [12], [13]. Additionally, if a target read is found, the following two conditions hold: the read address of the target read is correct, and the read data propagates correctly from the target read to the primary output. This is proved in the next lemma.

Lemma 1 *If a set of memory-locked errors is the root cause of the observed failure, then the read address of the target read is correct and the read data propagates correctly from the target read to the primary output.*

Proof: Let E be a solution to the suffix debugging instance $Debug_p^{p+w-1}(N)$ in the form of a set of N active suspect variables. If the read address for the observed data is incorrect, then a solution E will be found with n ($0 < n \leq N$) activated read address suspect variables. Likewise, if the data does not propagate correctly from the read data port to the observed output, a solution E will be found with n ($0 < n \leq N$) activated suspect variables for the fanout cone of the read data port. ■

As stated in Lemma 1, the lack of a target read implies that the read address port of the memory is being sent incorrect values, or the fanout cone of the read data port is incorrectly modifying the value received from memory. Suspects at these locations will allow the user to manually find and fix the associated errors. Once fixed, the sanity check pass is repeated until the design no longer exhibits the erroneous behavior or a target read is identified.

If the sanity check pass finds a target read then the observation window's length v is resized so time-frame $k - v + 1$ corresponds to the time-frame of the target read. Modeling earlier time-frames is not necessary since a target read indicates that the incorrect data did not pass through non-memory state elements before reaching the data out port.

For subsequent iterations, a sliding window of length w is modeled in addition to the observation window. The sliding window and observation window are not connected via the state transition, as seen in Fig. 3(b). Rather, the two windows are connected with the succinct memory model, and the initial states to the observation window are set from the error trace. As mentioned above, this connection will allow errors that propagate through memory to be found, but more importantly it will disallow errors that propagate into the observation window from non-memory state elements. Since the sanity check pass determined that incorrect data was being read directly from the memory, the error must propagate into the observation window's target read from the memory block. After each iteration, the sliding window is offset to an earlier time by w time-frames.

Once the sliding window and observation window are separated by at least one time-frame as seen in Fig. 3(c), interstitial frames are truncated leaving only the simulated forwarding lines from the succinct memory model. These lines, whose fanins are now disconnected, are set to their respective values from the cycle-accurate error trace. Formally, this is expressed in Eq. 3 and 4 below. Eq. 3 is identical to the debug window instance formulation in Eq. 2, but with the cardinality constraint and memory blocks removed:

$$D_p^{p+w-1} = S^p \wedge \bigwedge_{i=p}^{p+w-1} X^i \wedge Y^i \wedge T'^{\,i}_{en} \tag{3}$$

Using the equation above, Eq. 4 is constructed, where M_b are the bus forwarding lines, M_s are the simulated forwarding lines, and

Algorithm 1 Window Expansion for Memory-Locked Errors

1: $sols \leftarrow \emptyset$
2: $p \leftarrow (k - v - w + 1)$
3: **while** $p \geq 0$ **do**
4: $\quad sols_p \leftarrow$ SOLVEALL($MemLockDebug_p^k(N)$)
5: $\quad sols \leftarrow sols \cup sols_p$
6: $\quad$**if** TERMINATE(p, $sols_p$) **then**
7: $\quad\quad$**return** $sols$
8: $\quad$**end if**
9: $\quad p \leftarrow (p - w)$
10: **end while**

M are the enforced memory semantics. These memory constraints are added directly to the debugging instance as Conjunctive Normal Form (CNF) clauses. It is not needed to model the bus forwarding lines for interstitial frames since each truncated frame would have been modeled explicitly at a previous iteration. Similarly to BMD's window expansion, if the solutions to the debugging instance do not include the initial states, then the algorithm terminates. Otherwise this process is repeated, in the worst case until the sliding window reaches the beginning of the error trace. As such, Eq. 4 looks as follows:

$$\begin{aligned} MemLockDebug_p^k(N) = &D_p^{p+w-1} \wedge D_{k-v+1}^k \wedge \Phi(N) \\ &\wedge M_b \wedge M_s \wedge M \end{aligned} \tag{4}$$

Algorithm 1 presents pseudo-code for an observation window of length v and a sliding window of length w. Before running this algorithm, a sanity check pass(es) using traditional debugging techniques is used to ensure that a target read can be found. If not, there is no need to run the algorithm. Next, the debug window is iteratively lengthened (lines 3-10) and the resulting instance solved (line 4). If no initial state suspects are found (line 6) [7], the algorithm terminates, returning all solutions found so far.

Note that the proposed technique will miss suspects that are not wholly contained within the sliding window since interstitial frames do not model bus forwarding lines and their fanins. Experiments later in this paper show that the number of suspects missed is usually small, and is offset by the quick runtime and peak memory savings. An extension is presented in the next section that will allow all suspects to be found at the cost of increased runtime and memory usage. This extension makes the proposed methodology complete.

IV. Extension and Completeness

This section extends the technique in Section III to find all suspects including ones whose correction effects propagate across time windows. Additional constraints for the succinct memory model to disallow certain incorrect forwardings are also discussed. Finally, its completeness is proved.

A. Nested Window Expansion

The basic technique presented earlier is subject to similar shortcomings as previous windowing techniques: suspects will only be found if they are contained within the modeled time windows. Unlike BMD's window expansion however, window expansion for memory-locked errors only models two fixed-length time windows explicitly. Allowing for more time windows to be modeled while preserving most of the peak memory usage savings is the motivation behind the proposed extension.

The key addition of this technique is the use of a nested set of window expansions. At each iteration of the basic technique, an additional run of BMD's window expansion is performed. The resulting modeled *extended windows* allow suspects located outside of the original [p, k] time window to be found by the solver. Fig. 4 demonstrates the new technique in action, starting from iteration 2 of the basic technique using an observation window of length $v = 2$ and a sliding window of length $w = 1$.

Without the addition of any other clauses, the first nested iteration would immediately degenerate into BMD's window expansion and

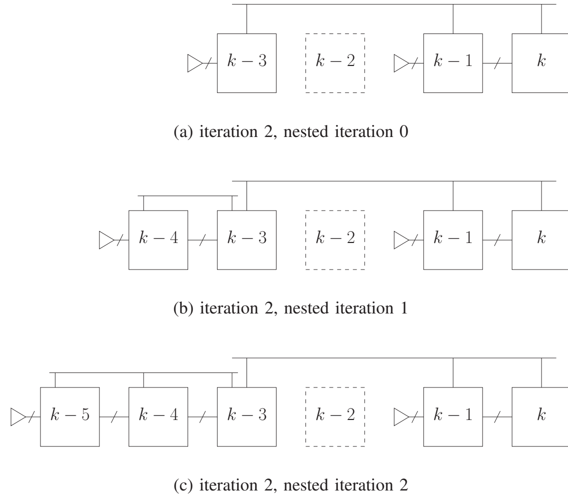

Fig. 4. Extended window expansion for memory-locked errors

run to completion, finding all suspects corresponding to memory-locked errors. Intuitively, this is because the extended windows could write a fix into memory that would be read at the target read, bypassing the sliding window entirely. The degenerated pass would then have a similar memory usage profile as window expansion, negating the effort done until now to avoid this. To prevent this, for a set of extended windows starting at time-frame q, additional clauses are added to the overall debug instance of the form:

$$BlockWrite_q^{p-1} = \bigwedge_{b=1}^{B} \bigwedge_{i=q}^{p-1} \bigwedge_{j=k-v+1}^{k} [bre_{j,b} \rightarrow \overline{bwe_{i,b}}] \quad (5)$$

These clauses disallow a time-frame in an extended window from writing to the same bus that is read from the observation window's target read. This can be thought of as the extended windows sharing bus forwarding lines with the sliding window, but not the observation window. Formally, the resulting debug instance is as follows:

$$\begin{aligned} ExtDebug_q^k(p, N) =& D_q^{p+w-1} \wedge D_{k-v+1}^{k} \wedge \Phi(N) \\ & \wedge M_b \wedge M_s \wedge M \\ & \wedge BlockWrite_q^{p-1} \end{aligned} \quad (6)$$

Algorithm 2 shows the necessary changes to the basic technique in order to find all suspects. The key addition is an inner loop at lines 6-13 that will perform a nested set of window expansions.

B. Memory Model Considerations

The succinct memory model originally presented in [9] does not model memory behavior exactly. Specifically, the behavior that subsequent writes will overwrite previous writes to the same address is not modeled for the bus forwarding lines. The motivation behind its exclusion is that the additional clauses needed to correctly model it would increase the memory footprint of the model, while the number of additional spurious suspects is relatively small. This trade-off is acceptable for the basic technique presented in Section III, but a complete technique requires the memory model to be enhanced with the following clauses:

$$\bigwedge_{i=q}^{k} \bigwedge_{b=1}^{B} [bre_{i,b} \rightarrow \overline{IWE_{i,b}}]$$

$$\bigwedge_{i=q}^{k-1} \bigwedge_{b=1}^{B} [\overline{IWE_{i,b}} \rightarrow \overline{IWE_{i+1,b}}]$$

$$\bigwedge_{i=q}^{k} \bigwedge_{b=1}^{B} [IRE_{i,b} \wedge IWE_{i,b} \wedge we_i \rightarrow WA_i \neq BA_b]$$

The first two sets of clauses define an intermediate write enable $IWE_{i,b}$ for time-frame i writing to bus b. Intuitively, time-frames where $IWE_{i,b} = 1$ correspond to time-frames before the bus read occurs. $IRE_{i,b}$ is defined in [9] in a similar fashion, but for time-frames after the bus write occurs. The last set of clauses disallows time-frames in between a bus write and read from writing to the same memory address, BA_b.

Algorithm 2 Extended Window Expansion for Memory-Locked Errors

```
1: sols ← ∅
2: p ← (k − v − w + 1)
3: while p ≥ 0 do
4:     sols_p ← ∅
5:     q ← p
6:     while q ≥ 0 do
7:         sols_q ← SOLVEALL(ExtDebug_q^k(p, N))
8:         sols_p ← sols_p ∪ sols_q
9:         if TERMINATE(q, sols_q) then
10:            break
11:        end if
12:        q ← (q − w)
13:    end while
14:    sols ← sols ∪ sols_p
15:    if q = p then
16:        return sols
17:    end if
18:    p ← (p − w)
19: end while
```

C. Method Completeness

The methodology presented in this section is complete in the sense that it will return all solutions corresponding to memory-locked errors. This is the topic of the theorem that follows.

Theorem 1 *Assume all memory blocks in the design are error-free. Let $sols_{ml}$ be the set of suspects found by the extended window expansion for memory-locked errors and let $sols_{bmd}$ be the set of suspects found by BMD's window expansion whose correction effects do not propagate into the state transition of time-frame $k-v+1$ (the initial state transition of the observation window). Then $sols_{ml} = sols_{bmd}$.*

Proof: As BMD's window expansion is complete [7] and the newly-enhanced memory model correctly models memory semantics, using both techniques together will also result in a complete set of suspects. Any suspect in $sols_{bmd}$ that is contained within $[k-v+1, k]$ will also be in $sols_{ml}$ since that window would be modeled explicitly by both techniques. Any suspect in $sols_{bmd}$ that is contained within $[0, k-v]$ and whose correction effects do not propagate through the non-memory state transition of $k-v+1$ will be in $sols_{ml}$. This is because the extended technique models as many time-frames as needed to find all suspects within $[0, k-v]$. Finally, suspects that cross the state transition $k-v+1$ would indicate the absence of a target read and can be excluded, as such an error could be corrected as shown in Lemma 1. ■

D. Practical Considerations

While the extended algorithm presented in this section will find all suspects, it has the possibility of degenerating into BMD's window expansion in the worst case. This can happen if an error propagates through non-memory state elements for many clock cycles before being written into memory. Statistically, most errors do not exhibit this behavior [7], a result also confirmed by the results presented in the next section. However, for the rare case that such an error is present, the proposed extension would lengthen the expanded window until it is found, causing an explosion in memory usage.

A practical workaround to this problem is limiting the maximum number of expansions performed by the nested window expansion. This constraint will still allow most suspects to be found, while preventing a runaway corner case that, even if it does not cause a mem-out it could negatively impact runtime.

V. Experiments

This section presents experimental results for the proposed window expansion for memory-locked errors along with the extended version. A workstation with an Intel Core i5 3.4 GHz quad-core processor and 8 GB of RAM, with a timeout of 7200 seconds is used as the benchmark for all experiments. BMD's window expansion [7] is compared against both Algorithm 1 (window expansion for memory-locked errors) and Algorithm 2 (extended window expansion) with the memory model enhancements discussed in Section IV. BMD [7] is chosen as the baseline since the two newly-proposed algorithms utilize it as a base and it is a technique that shares the same goal of reducing peak memory usage of long traces. Minisat [14] is used as the underlying solver for all SAT instances.

Industrial Verilog designs from `OpenCores` [10] and two commercial designs (`fifo`, `scam_core`) from our industrial partners are used to profile peak memory usage and runtime of the two algorithms. Failing instances of each design are created by modifying a line in the RTL to specifically create a erroneous behavior that will remain locked in memory for many clock cycles. The failing design is then simulated with the provided testbench and the resulting error trace is recorded.

Before running any of the window expansion techniques, cone of influence reduction [15] is used to prune the parts of the design that do not directly contribute to the erroneous behavior. This, combined with the insertion of a different error, may result in different instances of the same circuit having different sizes. All three window expansion techniques are then run on each design with an error cardinality $N = 1$ and window length $v = w = 50$. This length is chosen such that window expansion will run for at least two iterations even on the biggest designs. The two newly-proposed window expansion techniques are configured to use up to $B = 3$ bus forwarding lines, which for cycle-accurate error traces has been shown to be more than adequate for finding all suspects [9]. Lastly, the extended window expansion is limited to a maximum of five expansions.

Experimental results are found in Table I. Each row in the table gives results for a different failing instance. Since each design has been made to fail in more than one way, instance names are differentiated by using the original name of the design with an index appended, each corresponding to a different introduced error.

The first three columns in Table I give the instance name, the gate count for the instance in thousands of gates, and the total number of clock cycles in the error trace exposing the failure. The next three columns give results for BMD's window expansion: peak memory usage in megabytes, runtime in seconds, and the total number of clock cycles analyzed. The following two sets of four columns give corresponding results for the basic and extended window expansion for memory-locked errors, in addition to the total number of suspects found. A **TO** (**MO**) is used to denote that a time-out (mem-out) occurred for that particular run. Partial results for these situations are given in the table.

The most apparent benefit of the proposed algorithms is the significantly reduced peak memory usage for all instances. A direct consequence of this is a reduced number of instances hitting a **MO** condition when compared to previous work. Algorithm 1 is able to complete analysis on 78% more instances while Algorithm 2 completes 67% more. This corresponds to 9 completed instances for window expansion, 16 completed instances for Algorithm 1 and 15 completed instances for Algorithm 2. For instances that completed, Algorithm 1 shows an 83% decrease in average peak memory usage and a 73% decrease in average runtime versus window expansion. On the other hand, Algorithm 2 shows a 72% decrease in average peak memory usage and comparable runtime.

Fig. 5 shows the peak memory usage for the instances `ethernet1`, `fifo2` and `vga1` for window expansion, the basic technique, and the extended technique in order. It can be seen that Algorithm 1 shows a consistently greatly reduced memory footprint. Algorithm 2 on the other hand shows varying reduced memory usage when compared to window expansion. The increased memory usage when compared to the basic algorithm is caused by the additional clauses need to constrain the expanded windows and memory semantics, along with the potential for modeling more time-frames explicitly. The variation in memory savings is due to how many times the nested window expansions need to expand in order to find all suspects.

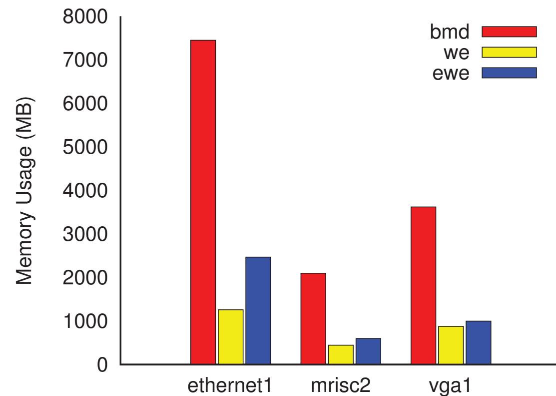

Fig. 5. Memory usage of window expansion and novel techniques

The runtime results for Algorithm 1 show improvements across the board, which is to be expected as the number of iterations would be identical to window expansion (assuming no **MO**), but the generated debugging instance is of greatly reduced size. This effect is demonstrated in Fig. 6, which compares BMD to Algorithm 1's memory usage versus the number of clock cycles analyzed for `vga1`. It can be seen that while window expansion's memory usage grows quickly with the number of time-frames modeled, window expansion for memory-locked errors' memory usage features very slow growth. This is because the latter technique only explicitly models two fixed-length time windows. Interstitial frames are modeled with comparatively very few clauses: the simulated buses and linear-scaling constraints enforcing memory semantics.

When runtime results for Algorithm 2 are compared with those of previous work [7], it is shown that it will perform better or will compare fairly well. This is due to the fact that statistically most errors do not propagate through non-memory state elements for many clock cycles. In the case of `mips2` however, both window expansion and Algorithm 2 terminate but the latter shows an increase in runtime. This is caused by the nested window expansions having performed several expansions at each iteration. It is also depicted in Fig. 7 which shows the count of the number of expansions performed. It can be seen that while the distribution is left-skewed, a trend that confirms the statistical observation discussed earlier, there are enough instances of higher number of expansions being performed to negatively impact the total runtime. Admittedly, the benefit of the proposed techniques lies in the greatly reduced peak memory usage allowing the analysis of instances that would not have been possible with previous techniques. As such, runtime is a secondary priority.

The number of suspects each of the proposed algorithms finds can be seen in columns 10 and 14 in Table I. It can be noted that Algorithm 1 usually finds a number of suspects comparable to the

TABLE I
WINDOW EXPANSION FOR MEMORY-LOCKED ERRORS RESULTS

Instance Info			**Window Expansion [7]**			**Basic**				**Extended**			
instance name	# gates (k)	# clk cycles	mem (MB)	time (sec)	# debug cycles	mem (MB)	time (sec)	# debug cycles	# sols	mem (MB)	time (sec)	# debug cycles	# sols
ac97_ctrl1	22.9	51000	5602	2497	1500	509	533	1500	43	695	851	1500	42
ac97_ctrl2	22.9	37000	6830	2841	1750	551	670	1750	55	741	937	1750	59
ethernet1	70.1	3920	7448	1057	950	1258	497	950	79	2467	1009	950	103
ethernet2	63.3	830	5931	1268	700	1026	459	700	40	2196	1396	700	87
fifo1	79.9	95000	**MO**	3451	800	2418	6394	10850	156	5390	7068	10850	218
fifo2	78.6	34000	**MO**	3705	800	1961	2172	4600	128	3607	4573	4600	155
fpu1	73.2	10000	**MO**	1278	750	1840	2459	3750	169	4607	5193	3750	221
fpu2	71.9	10000	**MO**	1493	850	1972	3094	5000	147	5599	5816	5000	240
mips1	49.7	26000	**MO**	5125	1000	1709	1755	1900	115	3276	**TO**	950	76
mips2	50.2	10500	7819	4691	700	1435	857	700	101	2864	6840	700	103
mrisc1	15.8	1540	1562	926	1200	419	361	1200	48	563	907	1200	50
mrisc2	15.7	1540	2097	873	1350	443	393	1350	56	598	1262	1350	68
scam_core1	499.7	2060	**MO**	593	300	3817	6071	2750	140	5930	7106	2750	167
scam_core2	499.3	4180	**MO**	672	350	3650	4846	1900	178	5004	6416	1900	193
vga1	42.0	5530	4025	1973	950	1095	568	950	89	1350	742	950	95
vga2	30.8	7390	3619	1708	1100	876	497	1100	99	997	616	1100	108

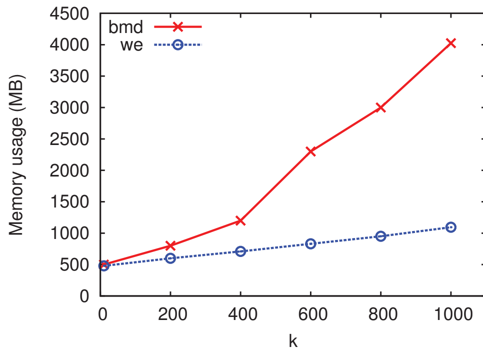

Fig. 6. Memory usage vs. clock cycles analyzed for `vga1`

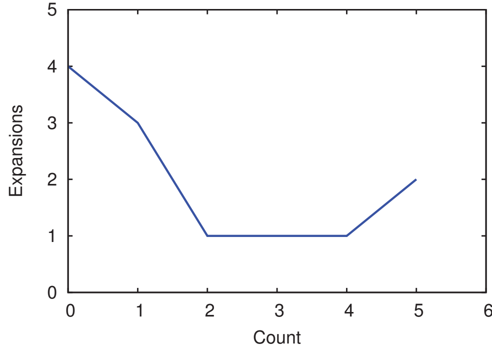

Fig. 7. Frequency of number of expansions performed for `mips2`

extended algorithm with an average of 23% missed suspects for instances that completed. In cases such as `ethernet1` and `fpu2`, the extended algorithm needs many nested expansions in order to find all suspects. This directly translates to both a higher peak memory usage, increased runtime and comparatively more suspects found.

It should be noted that Algorithm 1 finds additional spurious suspects owing to an imperfectly constrained memory model, as discussed in Section IV, and observed in the case of `ac97_ctrl1` in Table I. However, these turn out to be less than 4% of the total suspects found on average, confirming the statistical observations of [9]. The memory savings gained from allowing certain incorrect forwardings are in large part due to the reduced number of clauses needed to model memory behavior for interstitial truncated frames.

VI. CONCLUSION

In this paper, a novel technique for finding memory-locked errors is proposed. The technique models two time windows connected with a lightweight memory model. With each iteration the earlier window is offset, allowing memory-locked errors to be found across the error trace. The extended technique adds nested window expansion iterations to find errors that cross time windows. Experimental results on industrial designs with very long error traces show a large decrease in peak memory usage when compared against previous state-of-the-art work.

REFERENCES

[1] H. Foster, “From volume to velocity: The transforming landscape in function verification,” in *Design Verification Conference*, 2011.
[2] M. Prasad, A. Biere, and A. Gupta, *A Survey of Recent Advances in SAT-based Formal Verification*, 2005, vol. 7, no. 2.
[3] K. Karnane and G. Corey, “Automating root-cause analysis to reduce time to find bugs by up to 50%,” *Cadence*, 2015.
[4] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.
[5] 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.
[6] E. Clarke, A. Gupta, and O. Strichman, “Sat-based counterexample-guided abstraction refinement,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 7, pp. 1113–1123, July 2004.
[7] B. Keng, S. Safarpour, and A. Veneris, “Bounded Model Debugging,” *IEEE Trans. on CAD*, vol. 29, pp. 1790–1803, November 2010.
[8] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
[9] B. Keng, H. Mangassarian, and A. Veneris, “A succinct memory model for automated design debugging,” in *IEEE/ACM Int’l Conf. on CAD*, Nov 2008, pp. 137–142.
[10] OpenCores.org, “http://www.opencores.org,” 2007.
[11] M. Ganai, A. Gupta, and P. Ashar, “Verification of embedded memory systems using efficient memory modeling,” in *Design Automation Conf.*, March 2005, pp. 1096–1101 Vol. 2.
[12] M. Pandey, R. Raimi, R. Bryant, and M. Abadir, “Formal verification of content addressable memories using symbolic trajectory evaluation,” in *Design Automation Conf.*, June 1997, pp. 167–172.
[13] A. Cohen, J. O’Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck, “Verifying correctness of transactional memories,” in *Int’l Conf. on Formal Methods in CAD*, Nov 2007, pp. 37–44.
[14] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.
[15] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, January 2000.