A Succinct Memory Model for Automated Design Debugging

Brian Keng[1] Hratch Mangassarian[1] Andreas Veneris[1,2]

***Abstract*— In today's complex SoC designs, verification and debugging are becoming ever more crucial and increasingly time-consuming tasks. The prevalence of embedded memories adds to the difficulty of the problem by exponentially increasing the state-space of the design. In this work, a novel memory model for design debugging is presented. It models memory succinctly by avoiding an explicit representation for each memory bit. The method uses the simulation of the erroneous design to guide the debugging process. This results in a parameterizable formal encoding that grows linearly with the erroneous trace length, significantly reducing the memory requirements of the debugging problem. In addition, the proposed model is extended to handle an arbitrary initial memory configuration, as well as non-cycle accurate output traces where only a final expected memory state is available for comparison. Experiments on industrial designs show a 96% average reduction in memory usage along with a noticeable performance improvement compared to previous work.**

I. Introduction

Ensuring the functional correctness of modern VLSI designs, as well as localizing the source of errors and correcting them, are major parts of the design cycle consuming up to 70% of the design time [1]. These tasks are becoming more difficult as the use of embedded memories becomes more prevalent with the shift towards System-on-Chips (SoCs). Embedded memories give rise to an exponential increase in the state-space of a design, dramatically increasing the verification and debugging complexity of the system. This compounds the state-space explosion problem which already limits the practical application of many verification tools [2], [3].

Design debugging starts after an *error trace* is produced by a verification tool demonstrating some erroneous behavior in the design. It takes this error trace along with the erroneous design and attempts to return potential error locations that could explain the erroneous behavior. SAT-based automated design debugging [4], [5] has proven to be an effective technique compared to earlier methods [6], [7]. It involves *unrolling* a sequential circuit for the length of the trace in order to model the behavior of the circuit over time. A *cycle accurate* trace, where pin input/output values are defined for each clock cycle, is used to constrain the problem. The naïve approach to modeling memory in design debugging involves replicating each memory bit explicitly along with the rest of the design. In practice, this can lead to large run-time and memory requirements that make the problem intractable for even modest sized designs.

A general abstraction/refinement framework for design debugging proposed in [8] attempts to abstract state variables by replacing them with primary inputs. The refinement stage is necessary because some of the abstracted states may be causing the error, leading to incomplete results. However, [8] does not account for memories explicitly. A memory model is proposed by [9] for Bounded Model Checking (BMC). The technique eliminates memory arrays but keeps the memory interface and the control logic. It preserves the memory semantics by adding additional constraints to the BMC problem. More recently, [10] presents a set of memory abstraction algorithms for use in the verification of RTL models that must be specified within their tool-specific language. Although these memory models significantly reduce the size of the state-space compared to the naïve approach, the number of required constraints to model memory grows quadratically in the size of the unrolling k. This can lead to excessive memory requirements for large error traces.

In this work, we propose a novel memory model for design debugging that avoids the need to represent each memory bit explicitly. This is done in a SAT-based design debugging framework. The proposed method makes use of the erroneous simulation trace to produce a significantly smaller SAT encoding. In particular, the major contributions of the paper can summarized as follows:

- We give the first succinct memory model for design debugging, which replaces the whole memory with a number of clauses that is linear in the trace length k and a parameter B of the memory model.
- We extend this memory model to handle an arbitrary initial memory configuration.
- We broaden the proposed method to handle *non-cycle accurate* design debugging, where trace output values are not available for each clock cycle and only the expected final state of memory is known.

An extensive set of experimental results on industrial designs demonstrate the benefit of this work. The proposed cycle accurate debugging scheme achieves a memory usage reduction of several orders of magnitude compared to an explicit memory representation. When compared with previous methods, a 96% average reduction in the memory model size is observed as well a 20% improvement in run-time. Non-cycle accurate debugging results show a 98% decrease in the size of the memory model and comparable run-times to the explicit state representation.

The remaining sections of this paper are organized as follows. Section II provides background. Section III introduces the proposed design debugging memory model along with its extensions. Section IV presents experimental results and Section V concludes this work.

II. Preliminaries

A. Boolean Satisfiability (SAT) and SAT-based Debugging

Given a propositional logic formula Φ, SAT asks whether there exists an assignment to its variables that evaluates Φ to 1 or whether no such assignment exists. Modern SAT solvers typically accept Φ in conjunctive normal form (CNF), which is a conjunction of *clauses*, each made up of a disjunction of *literals*. For example, the formula $\Phi = (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b} \vee \bar{c}) \wedge (b \vee \bar{c})$ is SAT because $\{a = 1, b = 0, c = 0\}$ is a *satisfying assignment* evaluating Φ to 1. It should be noted that a logic circuit can be converted into CNF in linear time [11]. Recent advances in modern SAT solvers have allowed them to solve industrial instances with millions of variables and clauses efficiently, making them a popular tool for solving many VLSI problems [12], [13].

Design debugging begins after verification has returned a trace demonstrating an erroneous behavior of the design according to the specifications. Cycle-accurate design debugging takes as input the buggy circuit, as well as the erroneous trace along with the *correct*, or *expected*, output values for each clock cycle, or *time-frame*, from the specifications. Design debugging aims to locate all potential error locations that could explain the erroneous behavior in the trace.

SAT-based automated design debugging [5], [14] encodes the problem into a SAT instance in four steps: *a)* The circuit is enhanced with error-modeling hardware, *b)* the enhanced circuit is unrolled for each time-frame in the trace, *c)* the initial state, inputs and outputs of this unrolled circuit are constrained to the expected behavior corresponding to the erroneous trace, and *d)* the number of simultaneous error locations, or the error cardinality, is constrained. The error-modeling

[1]University of Toronto, ECE Department, Toronto, ON M5S 3G4 ({briank, hratch, veneris}@eecg.toronto.edu)

[2]University of Toronto, CS Department, Toronto, ON M5S 3G4

hardware allows arbitrary functions to replace suspect modules. If replacing a module with one of these functions can produce a satisfying assignment, then that module is deemed to be a potential error location.

When a solution is returned by the SAT solver, additional constraints are added to block this solution from being found again. This process repeats until no more satisfying assignments are found. Thus, given an error cardinality, every solution returned by the SAT solver corresponds to a potential error location.

B. Embedded Memory Semantics

A simple model of a dual-port embedded memory is shown in Figure 1. The embedded memory is connected to the design through several interface signals. A thin line (lowercase variable) indicates a single bit signal and a bold line (uppercase variable) indicates a bus signal. At any given clock cycle, both a write and a read from memory can occur. *Write Data* (WD) can be written to memory at *Write Address* (WA) when signal *write enable* (we) is set to 1 and the data written becomes available to read from in the next clock cycle. Similarly, *Read Data* (RD) can be read from memory at *Read Address* (RA) if signal *read enable* (re) is set to 1. The read data becomes available in the current clock cycle. A subscripted variable (e.g. re_p) indicates the value of that variable during the subscripted time-frame (e.g. time-frame p).

The complete memory semantics can be described in terms of these memory interface signals as follows. The data read from memory at time-frame q (RD_q) is equal to the data written to memory at time-frame p (WD_p), where $q > p$, if the following conditions are met: *a)* The read and write are done from and to the same memory address (i.e. $WA_p = RA_q$), *b)* the write enable is active at time-frame p (i.e. $we_p = 1$), *c)* the read enable is active at time-frame q (i.e. $re_q = 1$), and *d)* there are no other time-frames in between during which the same address is written to. When this situation occurs, we say that the data is *forwarded* from p to q, or there exists a forwarding path $p \rightsquigarrow q$, or that time-frame q is reading from time-frame p, or equivalently, time-frame p is writing to time-frame q. The writing time-frame p is occasionally referred to as the data forwarding *source*.

This can be formally expressed with the following set of constraints:

$$\bigwedge_{p=1}^{k} \bigwedge_{q=p+1}^{k} \Big\{ [(WA_p = RA_q) \wedge we_p \wedge re_q \wedge \bigwedge_{l=p+1}^{q-1} ((WA_l \neq RA_q) \vee \overline{we_l})] \rightarrow (WD_p = RD_q) \Big\} \quad (1)$$

C. Existing Memory Models

The explicit approach for modeling memory consists of representing each bit of memory as a state bit (flip-flop) and modeling the necessary address decoding and control circuitry. This usually results in a state-space explosion and quickly renders the problem intractable even for modest sized memories.

A memory model for BMC is described in [9]. In this work, we refer to the SAT-based memory model of [9] as the *crossbar model*. The crossbar model removes the memory array but maintains the control and interface signals. Constraints are then added to each time-frame of the SAT instance to ensure that the memory semantics are preserved according to Equation 1. These constraints give the ability to forward any WD_p to any future RD_q such that $q > p$ while preserving the memory semantics. [9] adds a forwarding path between each time-frame and all its previous time-frames resulting in a number of CNF clauses that grows quadratically ($O(k^2)$) with the number of unrollings k.

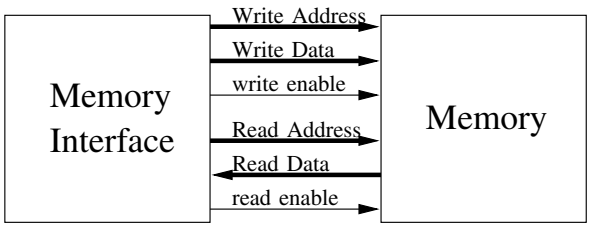

Fig. 1. Dual-port memory interface signals

III. A Novel Design Debugging Memory Model

As designs with embedded memories become more prevalent, the need for scalable memory models for design debugging becomes crucial. Since SAT-based design debugging involves replicating the circuit many times, having an explicit state representation quickly becomes infeasible. Larger trace lengths also pose a problem for memory models with a space complexity of $O(k^2)$. In this section, we introduce a memory model that is asymptotically more efficient. A unique feature of the proposed method is that it makes use of the simulated values in the erroneous circuit to guide the debugging process. The given formulation is for a simple dual-port memory but can be easily extended to a generic multi-port memory.

The proposed memory model replaces the memory array by a set of CNF constraints to be added to the SAT instance. These constraints control memory accesses by allowing two types of data forwarding across time-frames: *a)* forwarding according to the simulation of the design in the erroneous trace, or *b)* forwarding through newly introduced buses, used to fix forwarding errors in the erroneous trace.

The intuition behind the use of a simulated forwarding path in the erroneous trace lies in the fact that many design bugs do not affect memory accesses. In fact, if the error does not excite a change in memory read/write addresses, then data can be forwarded through memory according to the simulation of the design in the erroneous trace. On the other hand, if the error affects the memory addresses, a parameterizable number of extra buses B are introduced, which allow for changes to the memory accesses from the erroneous trace. This gives the solver the flexibility to choose alternative forwarding paths.

The idea is to increase the number of extra buses B until the SAT solver has enough flexibility to fix all forwarding errors and find the error location. If the actual error is not identified using a certain value of B by the designer, B is incremented and the debugging process runs again. On subsequent debugging runs, SAT solutions from previous runs are blocked, since they have been shown not to be the error source. Experimental results show that small values for B are typically enough to find most design bugs.

The constraints used in this memory model can be categorized into three different types: forwarding using simulated values, forwarding using bus variables, and enforcing memory semantics.

Forwarding using simulated values: For each time-frame q, let $l(q)$ be the index of the most recent time-frame $p < q$ such that there exists a forwarding path $p \rightsquigarrow q$ during the simulation of the design using the erroneous trace. Formally, $l(q) = \max\{p | \exists$ path $(p \rightsquigarrow q)$ in the erroneous trace$\}$. Let se_q denote the *simulation enable* variable that enables the forwarding of data to time-frame q according to simulated values. The following constraints, expressible in $O(k)$ clauses, describe the functionality of the se_q variables:

$$\bigwedge_{1 \leq q \leq k | \exists l(q)} [se_q \rightarrow (WA_{l(q)} = RA_q) \wedge (WD_{l(q)} = RD_q)] \quad (2)$$

Note that $l(q)$ does not exist for time-frame q if no previous time-frame writes to it during simulation.

If the data forwarding path $l(q) \rightsquigarrow q$ is not affected by the error, the SAT solver can choose to assign $se_q = 1$ to allow this forwarding to occur. On the other hand, if a design bug has excited a change in the forwarding addresses, the simulated forwarding paths might be erroneous. Therefore setting $se_q = 1$ might not result in a satisfying assignment. In order for the SAT solver to be able to find a correct forwarding path, we introduce new bus variables, which enable alternative means of forwarding.

Forwarding using bus variables: We introduce B *data and address buses*, BD_b and BA_b ($1 \leq b \leq B$), that allow the creation of new forwarding paths in the case of a mismatch in the simulated memory accesses. Each time-frame is allowed to write to or read from these bus variables. This added flexibility makes it possible for the SAT solver to produce satisfying assignments even if a design bug affects the correctness of the memory accesses. Let $bwe_{p,b}$ denote the *bus write enable* signal which enables time-frame p to write to bus b (i.e.

to BD_b and BA_b). The following constraints, expressible in $O(kB)$ clauses, describe the functionality of the $bwe_{p,b}$ variables:

$$\bigwedge_{p=1}^{k}\bigwedge_{b=1}^{B}[bwe_{p,b} \rightarrow (WA_p = BA_b) \wedge (WD_p = BD_b)] \quad (3)$$

Setting $bwe_{p,b} = 1$ allows time-frame p to write its address and data values to bus b. This is the first step in forwarding data from time-frame p using the bus variables.

The next step involves a time-frame q reading from the same bus. Let $bre_{q,b}$ be the *bus read enable* signal that allows time-frame q to read from the data and address buses BD_b and BA_b. The following constraints, expressible in $O(kB)$ clauses, describe the functionality of the $bre_{q,b}$ variables:

$$\bigwedge_{q=1}^{k}\bigwedge_{b=1}^{B}[bre_{q,b} \rightarrow (RA_q = BA_b) \wedge (RD_q = BD_b)] \quad (4)$$

Setting $bre_{q,b} = 1$ allows data to be forwarded from bus b to time-frame q. Therefore, setting $bwe_{p,b} = 1$ and $bre_{q,b} = 1$ for some p, q and b creates a data forwarding path $p \rightsquigarrow q$.

If the simulated data forwarding path $l(q) \rightsquigarrow q$ is wrong because of a design bug, the SAT solver will attempt to use one of the unused buses to forward the data in a way that satisfies the design debugging problem constraints. This is done by setting $se_q = 0$, which bypasses the simulated forwarding and instead setting $bwe_{p,b} = 1$ and $bre_{q,b} = 1$ for some appropriate b and p, in order to enable time-frame q to read from time-frame p.

Enforcing memory semantics: Additional constraints are needed in order to ensure that enable signals for forwarding are set correctly, and that the memory semantics are preserved.

To ensure that each time-frame q can only have one source to read from, we constrain the number of sources that are able to forward to it as follows:

$$\bigwedge_{q=1}^{k}\left[\left(\sum_{b=1}^{B} bre_{q,b} + se_q\right) = re_q\right] \quad (5)$$

Equation 5 captures the idea that if $re_q = 1$, only one of the forwarding sources, either a simulated forwarding path or a path using one of the buses, can be enabled. If $re_q = 0$, they are all disabled and no forwarding occurs to time-frame q.

Constraining the sum of Boolean variables to a certain cardinality can be achieved in different ways as shown in [15]. Our implementation uses bit-wise hardware adders, which are built using a linear number of clauses in the number of added bits [5]. For Equation 5, we need k adders of B bits each, resulting in a total of $O(kB)$ clauses.

Next, we must ensure that only one value is written to a single bus, using the following constraints:

$$\bigwedge_{b=1}^{B}\left[\sum_{p=1}^{k} bwe_{p,b} = 1\right] \quad (6)$$

Equation 6 is also implemented using adders amounting to a total of $O(kB)$ clauses. We must also make sure that a time-frame p does not forward its data if $we_p = 0$, with the following constraints, expressible in $O(kB)$ clauses:

$$\bigwedge_{p=1}^{k}\bigwedge_{b=1}^{B}[\overline{we_p} \rightarrow \overline{bwe_{p,b}}]$$

$$\bigwedge_{q=1}^{k}[\overline{we_{l(q)}} \rightarrow \overline{se_q}]$$

A final set of constraints is added to satisfy the causality of the memory semantics. That is, a write must occur to an address before a read from that address can take place. To ensure that this constraint is met, we use an intermediate bus read enable variable $PE_{p,b}$ between each time-frame p and bus b, as follows:

$$\bigwedge_{p=1}^{k}\bigwedge_{b=1}^{B}[bwe_{p,b} \rightarrow \overline{PE_{p,b}}]$$

$$\bigwedge_{p=2}^{k}\bigwedge_{b=1}^{B}[\overline{PE_{p,b}} \rightarrow \overline{PE_{p-1,b}}]$$

$$\bigwedge_{p=1}^{k}\bigwedge_{b=1}^{B}[\overline{PE_{p,b}} \rightarrow \overline{bre_{p,b}}]$$

If time-frame p writes to bus b, these constraints force the SAT solver to disable all reads from bus b by time-frames q such that $q \leq p$. This ensures the causality of memory writes and reads using $O(kB)$ clauses.

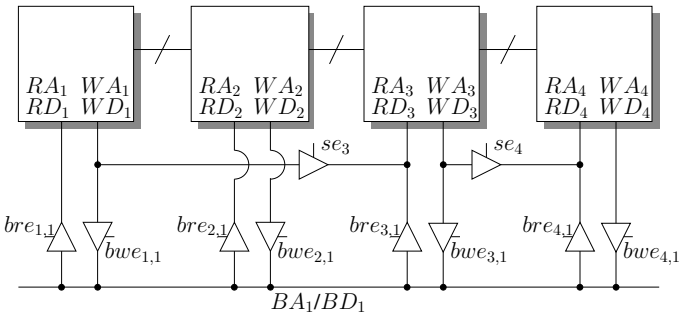

Fig. 2. Proposed memory model for design debugging

Example 1 *Figure 2 shows how the first two types of constraints can be added to an unrolled circuit. In this example, data is forwarded in the simulated erroneous trace from time-frames* 1 *to* 3 *and from time-frames* 3 *to* 4*. These connections are added to the SAT instance and are enabled via* se_3 *and* se_4*. We also have one set of bus variables* ($B = 1$)*. Every time-frame* p *can either read from or write to the bus depending on the enable signals* $bre_{p,1}$ *and* $bwe_{p,1}$*. Clearly, only one time-frame should write to the bus and only one source (either simulated forwarding or bus forwarding) should be enabled for each time-frame. This idea is captured with the additional constraints (not shown in the figure), which ensure that the memory semantics are preserved.*

It should be noted that this construction allows for additional solutions (i.e. potential error locations) to be found, beyond what the explicit memory representation would find. The reason is that certain data forwarding types are permitted which are not allowed by the memory semantics. A special case can occur when a time-frame q reads from time-frame p but the most recent write is at time-frame m such that $p < m < q$. The method will still find all solutions with up to B changes to the memory accesses, but might return extra solutions that are not potential error locations. These additional forwardings are permitted because they do not significantly increase the number of solutions returned. Additionally, disallowing these behaviors would greatly increase the size of the model.

The number of clauses produced by this construction grows according to $O(kB)$, where B is the number of buses used and k is the number of unrollings. In many cases, we have $B << k$, resulting in a significant decrease in the size of the problem compared with previous memory models.

A. Modeling Initial Memory Configurations

The above construction disallows reads from the initial state of memory (i.e. before the first time-frame), which is a valid assumption if the trace begins from a reset state. However, this may not always be the case since debugging a circuit from a reset state may require the use of an excessively long trace. Instead, a suffix of the trace is commonly used to reduce the problem size.

Let $M_{initial}$ be an initial state of memory comprising of I pairs of addresses and data, of the form (IA_i, ID_i), where $1 \leq i \leq I$.

We wish to allow time-frames to read values from $M_{initial}$ via the forwarding methods introduced in Section III. To that end, we use the two forwarding methods from our debugging memory model: forwarding according to simulated values and forwarding using bus variables.

An address and data entry $(IA_i, ID_i) \in M_{initial}$ is allowed to be forwarded to some time-frame q if that behavior occurs during simulation. Let $m(q)$ be the initial state location that time-frame q reads from during simulation. Note that $m(q)$ is unique because time-frame q can only read from one address. Now we can add the following constraints to Equation 2 to include data and addresses read from the initial state:

$$\bigwedge_{1 \le q \le k | \exists m(q)} [se_q \rightarrow (IA_{m(q)} = RA_q) \wedge (ID_{m(q)} = RD_q)]$$

Similarly, each entry in $M_{initial}$ can now be a source for any time-frame to read from using the bus variables. We must allow the initial state to write to the bus variables in a similar manner to Equation 3 for each (IA_i, ID_i) pair in $M_{initial}$. We extend the notation for $bwe_{p,b}$ to let $bwe^i_{0,b}$ denote the *bus write enable* signal that allows the initial memory location (IA_i, ID_i) to write to bus b. Using these variables, the following constraints need to be added to Equation 3:

$$\bigwedge_{i=1}^{I} \bigwedge_{b=1}^{B} [bwe^i_{0,b} \rightarrow (IA_i = BA_b) \wedge (ID_i = BD_b)]$$

In addition, we must also ensure that the bus variables are given the choice to connect to the initial state of memory by modifying Equation 6 as follows:

$$\bigwedge_{b=1}^{B} \left[\left(\sum_{p=1}^{k} bwe_{p,b} \; + \; \sum_{i=1}^{I} bwe^i_{0,b} \right) = 1 \right] \tag{7}$$

This ensures that each bus is written to either by the initial state of memory or by a time-frame in the trace.

Every address in memory does not need to be included in $M_{initial}$, but rather only values that are known to be valid. This information is easily obtained by a standard simulator. It should be noted that using similar types of constraints as described above, the crossbar model in [9] can be extended to deal with initial memory configurations as well.

Example 2 *Figure 3 extends Example 1 to handle initial memory states. The trace length has been cut in half by taking the last two time-frames from Example 1 (the time-frames are re-indexed). The memory state after the second time-frame in Example 1 now becomes $M_{initial}$, which has one entry, (IA_1, ID_1). (IA_1, ID_1) is connected to time-frame 1 and enabled via se_1 because during simulation this time-frame would have read from $M_{initial}$. In addition, we must also connect (IA_1, ID_1) to the bus variables to allow it to forward data to any future time-frame. This is enabled using $bwe^1_{0,1}$, which is constrained in Equation 7 (not shown in the figure).*

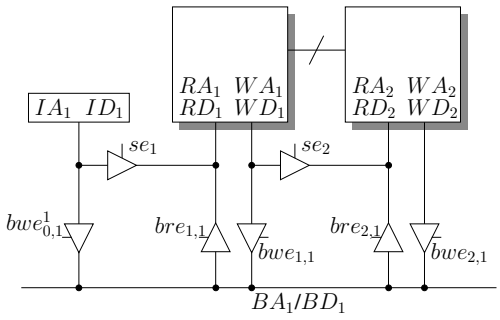

Fig. 3. Modeling an initial memory configuration

B. Non-Cycle Accurate Design Debugging with Final Memory Configurations

In certain situations, a cycle accurate expected output trace is not available. Instead, only an expected final state of memory can be used for comparison. For example, in verification environments today, it is common for RTL-level implementations to be compared to non-cycle, pin accurate high-level behavioral models, e.g. a C program. If an error is found during verification, the trace ends when the state of memory differs between the design and the high-level model and no intermediate pin mapping values are available. This results in the need to constrain the final state of memory to the expected memory configuration given by the high-level model.

The presented memory model for design debugging allows a natural extension to handle these types of non-cycle accurate output traces. In this problem, we are given an erroneous design, an input trace, as well as the *expected* final state of memory M_{final}. M_{final} is a set of F pairs (FA_f, FD_f), where FD_f is the data at address FA_f, for $1 \le f \le F$. The idea behind this formulation is to ensure that every pair $(FA_f, FD_f) \in M_{final}$ has at least one source which forwards data to it, coming from either a simulation forwarding or a bus forwarding.

Let $h(f)$ denote the most recent time-frame p such that there exists a forwarding path between time-frame p and (FA_f, FD_f) in the simulation of the design in the erroneous trace. We extend the notation for se_q to let se^f_{k+1} denote the enable signal allowing a forwarding path from time-frame $h(f)$ to the final memory location (FA_f, FD_f). Now, we can add the following constraints to Equation 2 to include data and addresses written to the final memory state:

$$\bigwedge_{1 \le f \le F | \exists h(f)} [se^f_{k+1} \rightarrow (WA_{h(f)} = FA_f) \wedge (WD_{h(f)} = FD_f)] \tag{8}$$

Setting $se^f_{k+1} = 1$ creates a forwarding path from time-frame $h(f)$ to (FA_f, FD_f). Note that $h(f)$ may not exist for every f because the expected final memory state might have more entries than the simulated final memory state. Also, the definition of $h(f)$ along with Equation 8 can be trivially extended to allow paths from initial memory state entries to final memory state entries. Such a forwarding path indicates that there have been no memory writes to that address during the trace.

The other possible data forwarding sources to the final state are the buses. For each pair (FA_f, FD_f), we extend the notation for $bre_{q,b}$ to let the enable signal $bre^f_{k+1,b}$ allow a connection to each set of bus variables, BA_b and BD_b. This enable signal is constrained in the same way as in Equation 4:

$$\bigwedge_{f=1}^{F} \bigwedge_{b=1}^{B} [bre^f_{k+1,b} \rightarrow (FA_f = BA_b) \wedge (FD_f = BD_b)]$$

When $bre^f_{k+1,b} = 1$, data is forwarded from the bus to (FA_f, FD_f).

Finally, we must ensure that only one time-frame (or the initial state of memory) forwards data to each pair (FA_f, FD_f). A similar constraint to Equation 5 is used to ensure this, as follows:

$$\bigwedge_{f=1}^{F} \left[\sum_{b=1}^{B} bre^f_{k+1,b} + se^f_{k+1} = 1 \right]$$

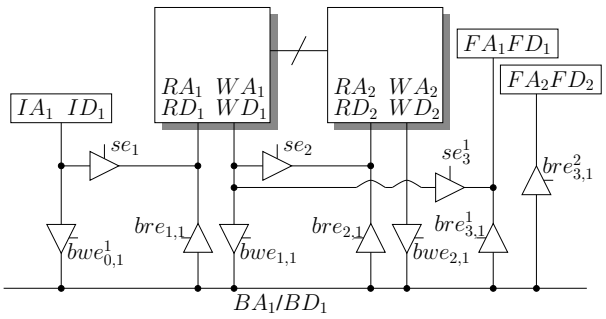

Fig. 4. Non-cycle accurate design debugging with a final memory configuration

TABLE I

CIRCUIT STATISTICS

Circuit Name	# gates	# DFFs	# addr	# data	# extra gates	# extra DFFs
fifo	847	70	8	8	12738	2101
vga	20254	1834	7	24	18548	4874
mrisc	5542	349	7	8	6340	2381

Example 3 *Consider Figure 4, an extension of Example 2, using the final state of memory M_{final} to constrain the problem. M_{final} has two entries (FA_1, FD_1) and (FA_2, FD_2). During simulation, FA_1 is written to by time-frame* 1 *and therefore a connection is made between them, which is enabled using se_3^1. A connection is also made to the bus variables in a similar manner with $bre_{3,1}^1$. Notice that the entry (FA_2, FD_2) has no corresponding simulated forwarding path. This implies that during simulation no time-frame writes a value to this memory location. The only way to satisfy this entry is for a time-frame to write to the bus and for $bre_{3,1}^2$ to be enabled.*

IV. EXPERIMENTS

This section presents experimental results for the proposed memory modeling technique in a SAT-based automated design debugging framework. In addition to our own memory model, we implement the memory model in [9] and extend it to handle arbitrary initial memory configurations in a design debugging context. We also include results for the explicit memory representation which models memory as an array of flip-flops. A SAT-based design debugging framework similar to [14] is used to produce the SAT instances. The SAT-solver MINISAT-V2.0 [13] is deployed on a Pentium Core 2, 2.4 GHz workstation with 4GB of memory.

We show the effectiveness of the proposed memory model for multiple debugging instances on three designs with large dual-port embedded memories from OpenCores.org [16]. Instances are generated using either a real or random bug inserted in the RTL design. The error cardinality is set to 1 or 2 in all the debugging instances. The number of bus variables B is initialized to 1 and is incremented until the error is located.

Table I shows the relevant statistics for each of the three designs. The first three columns show the circuit name followed by the gate and flip-flop (DFF) count in the absence of embedded memories. Columns four and five show the address and data widths of the embedded memory, while the next two columns show the additional gates and flip-flops generated due to the explicit memory representation.

Table II shows cycle accurate design debugging results for two circuits: `fifo` and `vga`. Five different instances of varying trace lengths starting from a reset state are created for `fifo` (`fifo1` to `fifo5`). For each of the four instances with larger traces, results are shown using a suffix of the trace and making use of the initial memory configuration model described in Section III-A (`fifo2-init` to `fifo5-init`). Seven instances of `vga` (`vga1-init` to `vga7-init`) are also generated using the initial memory configuration.

The first three columns in Table II show the instance name, error cardinality and trace length. Column *base* shows the memory usage of each instance excluding the memory model size. The next two columns show the run-time and size of the memory model for the explicit memory representation. Columns 7 to 12 show the number of clauses in thousands, followed by the run-time in seconds and size of the memory model in MBs for both the crossbar model and the proposed memory model for design debugging (which we refer to as the *bus model*). The last column shows the B value required to solve each instance using our method. The run-times for the bus model are the sum of the run-times of all runs starting from $B = 1$ until the listed value of B.

For `fifo`, the proposed method produces an average run-time reduction of 26% and memory model size reduction of 95% compared to the crossbar model. For `vga`, the proposed memory model gives an average run-time reduction of 13% and size reduction of 98% compared to the crossbar model. On average 6% more solutions are produced by the proposed method which does not significantly harm the effectiveness of our debugging memory model. Both of these methods significantly outperform the explicit memory model both in terms of performance and memory model size. In fact, the explicit memory representation runs out of memory for many of the instances.

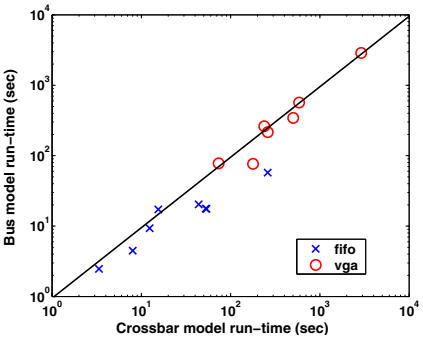

Fig. 5. Cycle accurate performance results

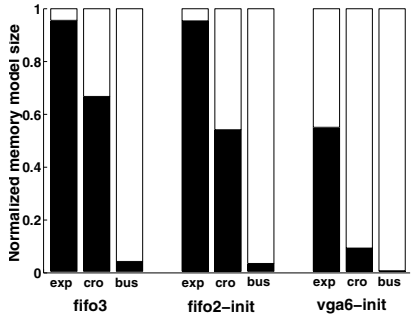

Fig. 6. Normalized memory size for `fifo3`, `fifo2-init` and `vga6-init`

Looking across the second and sixth rows of Table II, one can notice that the explicit memory representation runs out of memory when the full trace length is used but is able to solve the instance using a suffix of the trace. Comparing `fifo2` and `fifo2-init`, reducing the trace length results in a reduction in memory usage from 478MB to 150MB for the crossbar model, and from 7MB to 3MB for the bus model.

Figure 5 plots the performance results on a log-log scale for `fifo` and `vga` instances for the crossbar and bus model. Notice that most of the points lie below the 45 degree line indicating that the run-times of the proposed method are generally better. On the points that lie above the line, the performance is marginally better showing that we can achieve similar performance results with a large reduction in the memory model size.

Figure 6 shows the size of each memory model (in black) relative to the size of the entire instance for `fifo3`, `fifo2-init` and `vga6-init`. Here, *exp*, *cro* and *bus* respectively stand for explicit memory model, crossbar model and bus model. It can be seen that the explicit model takes a large portion of the problem size for all three instances, while the crossbar model size is more dependent on the trace length. The bus model consistently shows a small relative memory footprint.

From Table II, we see that in most instances $B = 1$. This can be explained in two ways. The first is that the error has very little effect on the address lines so only one time-frame is affected by the change, thus only one set of bus variables is needed. The second is that an error that affects an address line propagates to the outputs relatively quickly. As a result, it does not manifest itself across multiple time frames because the trace would end at that point. This is also the reason why we do not see a large increase in the number of solutions for the bus model. In many cases, the exact same number of solutions are found.

Figure 7 shows how the memory usage of the crossbar model and the debugging memory model vary with the number of unrollings k for the `fifo` design. Note that the data used in this figure is not shown in Table II. As predicted, the crossbar model grows quadratically with k, while the bus model is linear for a fixed value of B. It can also be seen that the memory usage grows linearly when increasing B.

Table III shows non-cycle accurate design debugging results on `mrisc`. It compares the proposed method to the explicit memory representation. `mrisc` is a microprocessor design whose outputs are

TABLE II

CYCLE ACCURATE DESIGN DEBUGGING RESULTS FOR FIFO AND VGA

Instance Info				Explicit		Crossbar			Bus Model			
instance	N	# clks	base (MB)	time (sec)	mem (MB)	# cls (K)	time (sec)	mem (MB)	# cls (K)	time (sec)	mem (MB)	B
fifo1	1	28	21	11.3	386	22	0.5	4	8	1.1	1	3
fifo2	1	307	167	MEMOUT		2725	53.7	478	40	17.5	7	1
fifo3	1	227	128	242.1	2772	1488	12.4	255	30	9.3	5	1
fifo4	1	595	330	MEMOUT		10251	260.4	1770	76	57.5	11	1
fifo5	2	83	47	687.6	953	197.3	43.9	82	18.1	20.3	2	2
fifo2-init	1	155	87	142.6	1913	879	8	150	19	4.5	3	1
fifo3-init	1	110	65	69.9	1335	448	3.35	76	13	2.5	2	1
fifo4-init	1	308	173	MEMOUT		3452	52.6	593	37	17.7	7	1
fifo5-init	2	47	29	335.2	574	75.2	15.5	13	10	17.2	1	2
vga1-init	1	148	2478	MEMOUT		1902	260.6	327	34	216.5	5	1
vga2-init	1	70	1216	810	1424	665	238.6	111	20	262.3	2	1
vga3-init	1	130	2210	MEMOUT		1570	586.6	262	31	568.8	4	1
vga4-init	1	58	1010	496.4	1235	526	178.7	94	18	76.7	2	1
vga5-init	1	103	1746	MEMOUT		792	502.2	152	18	344	3	1
vga6-init	1	82	1401	163.7	1701	821	73.5	136	22	77.7	3	1
vga7-init	2	49	776	6082.9	924	425	2901.7	57	14.7	2870.8	2	1

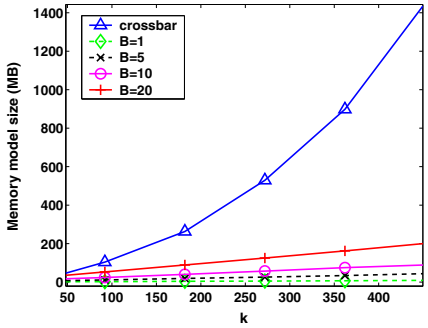

Fig. 7. Memory model size vs. trace length for fifo

not explicitly generated at each clock cycle. Instead, to verify the correctness of the design, the final state of the embedded memory (a register file) is compared to a high-level model. The first four columns of Table III indicate the instance name, error cardinality, trace length and memory usage excluding the memory model size. The next four columns show the run-time in seconds and size of memory model in MBs for both the explicit memory representation and the proposed memory model. The last column shows the value of B needed to find the design bug using our method. Again, the run-times for the debugging memory model are a sum of each run from $B = 1$ to the value of B shown in the table.

For mrisc1, the bus model results in a significant reduction in the memory model size from 199MB in the explicit model to 6MB. The run-time is about 17 seconds for both models in this instance. The explicit state model performs better on some instances such as mrisc3, but our method is faster in others. This is an acceptable trade-off in run-time for a 98% reduction in memory usage. Furthermore, for designs with larger memories, the size of the explicit memory representation will affect the run-time to a larger degree, degrading its performance.

Notice in Table III, the values of B to find the bug are overall higher than in those in Table II. The reason is that several memory locations in the final state of memory are not written to during simulation of the erroneous design. Thus, more bus variables are needed to ensure that each location in the final state of memory has a valid forwarding.

V. CONCLUSION

In this work, a novel memory model for design debugging is presented. The proposed memory model results in a parameterizable SAT encoding that grows linearly with the erroneous trace length. In addition, the described method is extended to handle initial memory configurations as well as non-cycle accurate design debugging with an expected final memory state. Experiments show that this model reduces memory requirements significantly in both cycle accurate and non-cycle accurate debugging.

TABLE III

NON-CYCLE ACCURATE DESIGN DEBUGGING RESULTS ON MRISC

Instance Info				Explicit		Bus Model		
instance	N	# clks	base (MB)	time (sec)	mem (MB)	time (sec)	mem (MB)	B
mrisc1	1	19	87	17.8	199	17.3	6	5
mrisc2	1	22	96	18.6	224	27.7	9	6
mrisc3	1	19	87	14.3	199	88.2	6	5
mrisc4	1	16	74	17.0	178	11.7	2	3
mrisc5	1	31	124	47.3	290	15.8	3	3

REFERENCES

[1] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publisher, 2000.

[2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment." in *CHARME*, 2005, pp. 254–268.

[4] M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir, "Debugging sequential circuits using Boolean satisfiability," in *Int'l Conf. on CAD*, 2004, pp. 204–209.

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

[7] N. K. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2002.

[8] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Design, Automation and Test in Europe*, 2007, pp. 1182–1187.

[9] M. K. Ganai, A. Gupta, and P. Ashar, "Verification of embedded memory systems using efficient memory modeling," in *Design, Automation and Test in Europe*, 2005, pp. 1096–1101.

[10] P. Manolios, S. K. Srinivasan, and D. Vroon, "Automatic memory reductions for rtl model verification," in *Int'l Conf. on CAD*, 2006, pp. 786–793.

[11] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 11, pp. 4–15, 1992.

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conf.*, 2001, pp. 530–535.

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

[14] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.

[15] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," in *JSAT*, vol. 2, 2006, pp. 1–26.

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