Optimal Trace Compaction with Property Preservation

Yibin Chen, Sean Safarpour, Andreas Veneris
Department of Electrical and Compupter Engineering
University of Toronto, Toronto, Canada
{yibin,sean,veneris}@eecg.toronto.edu

***Abstract*— Debugging design errors is a challenging manual task which requires the analysis of long simulation traces. Trace compaction techniques help engineers analyze the cause of the problem by reducing the length of the trace. This work presents an optimal error trace compaction technique based on incremental SAT. The approach builds a SAT instance from the Iterative Logic Array representation of the circuit and performs a binary search to find the minimum trace length. Since failing properties in the original trace must be maintained in the compacted trace, we enrich our formulation with constraints to guarantee property preservation. Extensive experiments show the effectiveness out SAT based approach as it preserves failing properties with little overhead to the SAT problem while demonstrating on average an order of magnitude in performance improvement.**

I. Introduction

Ensuring that a design operates correctly according to its specifications is one of the major challenges facing today's IC industry. Functional verification tasks can consume up to 70% of the design effort [1]. While adoption of formal verification techniques is on the rise, simulation based functional verification remains the industry's work horse [2]. Simulation based verification methodologies use testbenches composed of stimulus generators and correctness checkers. Stimulus generation is either directed, where specific input vectors are provided to the Design Under Verification (DUV), or constrained-random, where pseudo-random vectors are provided to a subset of the DUV inputs [1]. Correctness checkers identify when functional specifications are violated by the DUV.

When a checker identifies a violation of the specifications, the corresponding stimulus sequence or error trace is used to debug the design. Design debugging is a prominently manual task that can take up to 50% of the verification effort [2]. The complexity of debugging depends in part on the length of the error trace. The longer the trace, the more clock cycles and events must be consider.

To help alleviate the debugging effort, the length of error traces can be reduced using trace compaction. Trace compaction techniques generate an alternative sequence of stimulus events to transition the DUV from an initial state S_i to a final state S_f with fewer events or clock cycles. For simulation traces spanning thousands of clock cycles, the trace compaction problem may be broken up into smaller sub-problems to find shortcuts between intermediate states of the original trace. With debugging being an essential part of verification the performance of trace compaction algorithms can have a direct impact on debugging efficiency.

Many of today's trace compaction algorithms are based on heuristics that may not necessarily reduce the trace enough to ease the manual debugging effort. Thus *optimal* automated trace compaction techniques which can find the minimal number of clock cycles to transition from S_i to S_f are desireable.

Another challenge in trace compaction is that verification engineers may not only seek a shorter trace from S_i to S_f but they may also require that properties observed in the original trace are preserved. In today's verification environment correctness checkers can report failures due to properties, which require that a sequence of events occur. A compacted trace only constrained by the initial and final state may not exhibit the original failure. Unfortunately, existing trace compaction techniques do not guarantee that properties are preserved. Instead, they re-run the simulators and rely on the checkers to confirm the validity of the trace. As a result, trace compaction techniques that can preserve sequences of events are valuable to the industry.

The main contributions of this paper are as follows. Firstly, we focus on performance by introducing an efficient optimal trace compaction algorithm which returns a compacted trace of minimal length. Our trace compaction algorithm first obtains an upper bound for the minimal trace length using SAT and then employs a binary search using incremental SAT to find an optimal trace. Experimental results show on average a 12× improvement in run time over a previous optimal trace compaction algorithm. Secondly, this paper introduces the concept of assertion based error trace compaction for the purpose of *property preservation*. Properties specified using assertions have grown in popularity as they can be used by both simulation and formal verification tools. These properties may be specified by designers and verification engineers in languages such as SystemVerilog Assertions (SVA) [3]. This is a departure from traditional formulations as it guarantees that properties failing in the original simulation trace are preserved in the compacted trace. Failing assertions specified in SVA are directly encoded into the trace compaction formulation with minimal overhead.

The remainder of the paper is structured as follows. In Section II some of the previous advancements in this field are reviewed. The error trace compaction algorithm is given Section III. Section IV illustrates how SVA constraints can be encoded into the formulation to preserve properties. Finally experimental results and conclusions are provided in Sections V and VI.

II. Previous Work

Previous work in the area of trace compaction include an array of algorithms based on formal engines and simulation based approaches. In [4], a collection of simulation and formal techniques are used to iteratively reduce the error trace size. To ensure that the original error is preserved, the trace is re-simulated after each iteration.

Other techniques such as [5] and [6] use image computation to search for shortcuts between circuit states. In [6], BDDs are used to compute all one cycle reachable states from each unique state occurring in simulation. Similarly [5] performs pre-image computation using an all-solution SAT solver and don't cares to efficiently traverse the state space from the final state to the initial state.

Many trace compaction techniques proposed in recent years heavily employ SAT engines. For instance, [7] uses a sequential SAT solver combined with BMC techniques to find shortcuts between intermediate states. In fact the basic BMC technique can be used for trace compaction as it generates a minimal trace [8]. An optimal SAT based trace compaction algorithm using a binary search on the number of time frames in the trace is proposed in [9]. This technique reduces the number of SAT instances solved to $log_2(h)$ (h is the original trace length) whereas BMC needs to solve a linear number of SAT instances. However, [9] requires that at least one of the initial or final states is a self-transition state.

III. Optimal Error Trace Compaction

A. Detecting Reachability using SAT

Given a sequential circuit C with initial and final state constraints $S_i(Q)$ and $S_f(Q)$, we can determine whether S_f can be reached from S_i in exactly k steps by constructing the Iterative Logic Array (ILA) [10] of C for k time frames (or clock cycles). The problem can be formulated as a satisfiability problem in CNF form [11]:

$$\Phi_{=k} = S_i(Q^0) \wedge (\bigwedge_{0 \leq i < k} T_c(Q^i, I^i, Q^{i+1})) \wedge S_f(Q^k) \quad (1)$$

where $T_c(Q, I, D)$ is C's transition relation. Q, I and D denote the current state, input, and next state variables respectively. The above formula can be extended to express the reachability analysis problem for $\leq k$ time frames by adding multiplexers between timeframes:

$$\begin{aligned} \Phi_{\leq k} = & S_i(Q^0) \wedge (\bigwedge_{0 \leq i < k} T_c(Q^i, I^i, D^i) \\ & \wedge MUX(D^i, Q^i, e^i, Q^{i+1})) \wedge S_f(Q^k) \end{aligned} \quad (2)$$

In the above $MUX(D^i, Q^i, e^i, Q^{i+1})$ represents the constraints of time frame multiplexers (T-MUXes), where D^i and Q^i are the inputs and e^i is the select variable. If $e^i = 1$ then $Q^{i+1} = Q^i$; otherwise

$Q^{i+1} = D^i$. A satisfiable solution to $\Phi_{\leq k}$ contains assignments to the T-MUX select variables effectively determining which transition relations can be skipped. The resulting solution trace of length $\leq k$ thus consists of the sequence of input vectors I^j for which $e^j = 0$, for all $0 \leq j < k$. Note that $\Phi_{\leq k}$ is satisfiable if $\Phi_{=k}$ is satisfiable but the converse is not necessarily true unless the initial and/or the final state are self-transition states.

Consider the formulation for $\Phi_{\leq 4}$ as given in Figure 1. Suppose that the minimal trace length is 3 but S_f cannot be reached from S_i in 4 time frames (i.e. $\Phi_{=4}$ is not satisfiable). Then by setting exactly one of e^j, $0 \leq j \leq 3$ to 1 and the rest to 0 the problem is satisfied.

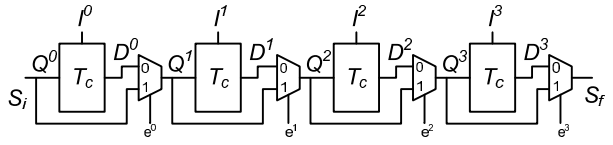


Fig. 1. ILA Representation of $\Phi_{\leq 4}$

B. Optimal Trace Compaction Algorithm

A trace from S_i to S_f of length m is a minimal length trace *iff* $\Phi_{\leq m}$ is SAT and $\Phi_{\leq m-1}$ is UNSAT. One way to solve for m given an original trace length h is to iteratively construct ILAs of different lengths until the minimal length is achieved [9]. Alternatively, using $\Phi_{<h}$, m can be solved by finding the maximum number of T-MUX select variables that can be set to 1.

The *incremental_trace_compaction* algorithm (Algorithm 1) finds m by incrementally solving $\Phi_{\leq h}$ using different unit constraints. Algorithm 1 takes the problem formulation $\Phi_{<h}$ as its single argument. The function *gather_tmux_select* on line 3 gathers and returns all the the T-MUX select variables. $solver.reset(\Phi_{<h})$ initializes the problem. A binary search using incremental SAT is performed by the **while** loop on lines 6-25. Notice that the solver state is never reset in the **while** loop thus allowing learned clauses to be retained between iterations. Unit clause constraints which may vary between iterations are passed to the solver using *unit_constraints*. The **for** loop on lines 10-12 ensures that at least tm transition relations are skipped via the T-MUXes. If the problem is satisfiable, *extract_trace* on line 15 extracts the satisfying assignment for the relevant input variables. Lines 17-21, re-adjusts the upper bound, $high$, according to the number of T-MUX select variables set to 1 in the satisfying assignment.

Due to the reuse of learned clauses between iterations, Algorithm 1 can find the minimal trace length very efficiently. Another powerful way to improve performance is to derive an initial upper bound for m. This can be effective for cases where m is much smaller than the original trace length h. The *derive_upper_bound* algorithm shown in Algorithm 2 performs this as a pre-processing step to *incremental_trace_compaction*.

Algorithm 2 takes the circuit C, the original trace length h, and a reduction factor $0 < f < 1$ as its arguments. The function

Algorithm 1 The incremental_trace_compaction algorithm

incremental_trace_compaction($\Phi_{\leq h}$)
1: $low \leftarrow 0$
2: $high \leftarrow h$
3: $\{e^0 \ldots e^h\} \leftarrow gather\_tmux\_select(\Phi_{\leq h})$
4: $compacted\_trace \leftarrow \emptyset$
5: $solver.reset(\Phi_{\leq h})$
6: **while** $high - low \neq 1$ **do**
7: $tm \leftarrow \lceil (high + low)/2 \rceil$
8: $unit\_constraints \leftarrow \emptyset$
9: // Disable the last tm time frames
10: **for** $i = 0$ to $tm - 1$ **do**
11: $unit\_constraints \leftarrow unit\_constraints \cup (e^{h-i})$
12: **end for**
13: $issat \leftarrow solver.solve(unit\_constraints)$
14: **if** $issat$ **then**
15: $compacted\_trace \leftarrow extract\_trace(solver)$
16: $high \leftarrow tm$
17: **for** $j = 0$ to tm **do**
18: **if** $solver.value(e^j) = 1$ **then**
19: $high \leftarrow high - 1$
20: **end if**
21: **end for**
22: **else**
23: $low \leftarrow tm$
24: **end if**
25: **end while**
26: **return** $compacted\_trace$

$construct\_ila(\Phi, n)$ constructs the CNF representation of the ILA obtained by replicating C for n time frames with all T-MUXes added. A SAT solver then tests for reachability within $\leq n$ time frames using progressively smaller n. The reduction factor f determines how fast the pre-processing step reduces the trace length and how close $upper$ is to the minimum.

Algorithm 2 The derive_upper_bound algorithm

derive_upper_bound(C, h, f)
1: $n \leftarrow h$
2: **repeat**
3: $upper \leftarrow n$
4: $n \leftarrow \lceil upper * f \rceil$
5: $\Phi_n \leftarrow construct\_ila(C, n)$
6: $solver.reset(\Phi_{\leq n})$
7: **until** $!solver.solve()$
8: **return** $upper$

C. Reducing the Search Space for T-MUX Select Variables

In the previous sections, formulations and algorithms were introduced to solve the optimal trace compaction problem. However, since T-MUXes are added after every transition relation, there are many possible assignments to the T-MUX select variables that effectively result in the same number of skips. For example, consider $\Phi_{\leq 4}$ again as shown in Figure 1, this time with a minimal trace length of 4. Here, there are no solutions of length 1, 2, or 3. In order to determine this fact, the solver must explore a state space of size $\binom{4}{1} = 4$, $\binom{4}{2}/2 = 6$, and $\binom{4}{3} = 4$, respectively. To reduce the search space, adjacent T-MUX select variables can be tied together through *connection* variables c^i, according to successive powers of two similar [12]. Using $2^0 + 2^1 \cdots + 2^{x-1} = 2^x - 1$, the additional constraints that need to be added to $\Phi_{\leq k}$ are given as follows:

$$\Phi_{\leq k} \wedge \Big(\bigwedge_{0<i<k} c^i \rightarrow (e^{i-1} = e^i) \Big) \wedge \Big(\bigwedge_{\substack{0<i<k \\ i \neq 2^x - 1}} c^i \Big) \wedge \Big(\bigwedge_{\substack{0<i<k \\ i = 2^x - 1}} \overline{c^i} \Big) \quad (3)$$

where x is any integer. Consequently, there is only one unique assignment to the e^i's for a given trace length.

For incremental SAT the implication relation in the above expression needs to be added to the $\Phi_{\leq k}$ during construction. Adjacent T-MUXes can thus be grouped together dynamically by adding their respective connection variables as a unit clause to $unit\_constraints$ in Algorithm 1. We can also take into consideration the lower bound low to further reduce the search space. An example using a lower bound length of $low = 1$, an upper bound length of $high = 9$, and an original length of $h = 10$ is given in Figure 2. The connectors which are depicted in grey are set to 0 indicating that the adjacent T-MUX select variables are not connected. An x on a T-MUX select variable indicates that the variable is unconstrained in the SAT problem. Since we are testing whether S_f can be reached from S_i in ≤ 9 clock cycles the last transition relation is skipped ($e^{10} = 1$). Similarly $e^0 = 0$ and $e^1 = 0$ since $low = 1$ and traces must be of length ≥ 2. The remaining T-MUX select variables can be grouped together as described above. In this case, the groupings are: {3}, {4,5}, and {6,7,8,9}. The SAT solver is allowed to return any solution trace whose length is within the range from 2 to 9.

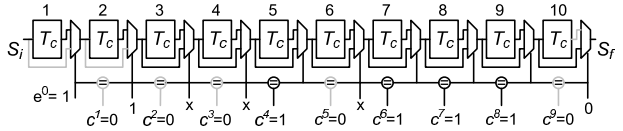
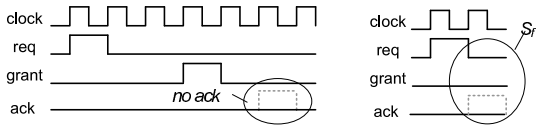


Fig. 2. Example encoding for $l = 2$, $h = 9$,and $k = 10$

IV. Assertion based Error Trace Compaction

In the previous section a SAT formulation is presented with the objective of deriving an error trace of minimal length from an initial state to a final state. The discussion did not address how to preserve properties that occur in the original error trace in the resulting compacted trace. Since circuit properties can define sequences of events to be maintained, using only the initial and final states to constrain a trace compaction problem is often not sufficient to ensure that properties are preserved.

For example, consider a simple sequence where the initial state `req` followed by `grant` must be followed by `ack` after two clock cycles. An error trace, can include the transition from `req`, to `grant` without observing `ack` thus violating the property (Figure 3(a)). A traditional trace compaction technique can inadvertently find a



(a) Original Error Trace (b) Compacted Trace

Fig. 3. Trace Compaction Example using only Initial and Final State

shorter trace without observing `grant` (Figure 3(b)). In this case, the resulting trace is of no value to the verification engineer since the failing property used for debugging is not violated.

Motivated by the above example, in this section, we add to the objective of trace compaction the requirement of property preservation. In particular we show how properties encoded in SVA [3] can be converted directly into CNF and added to $\Phi_{\leq k}$. While the use of properties to constrain problems is common in model checking [8], [13], to the best of our knowledge, this is the first trace compaction technique that considers property preservation. In the following sections we show how to encode some common SVA properties to enrich the proposed formulation. To ease discussion we will only consider the encoding of properties for the problem formulation given in Section III-B.

A. Properties with only Boolean Expressions

Consider the assertion property specified in SVA:

@(**posedge** clk) (a && b == d);

This assertions states that at every positive edge of the clock the bitwise AND of the variables a and b must be equal to d. The combinational expression can be directly converted to CNF and added to each time frame in the ILA. If this property is violated during the original error trace we need to ensure that the compacted trace also violates this property at least once. $\Phi_{\leq k}$ thus needs to be appended with the following expression:

$$(\bigwedge_{0\leq i<k} \overline{p^i} \leftrightarrow (\overline{e^i} \wedge \overline{(a^i \&\& b^i == d^i)})) \wedge (\bigvee_{0\leq i<k} \overline{p^i}) \tag{4}$$

where a^i and d^i are the corresponding variables for a and d in time frame i of the expanded CNF representation.

In simulation, each property is verified by creating a new instance of the property at at each clock cycle and evaluating the instance starting from each time frame. Similarly we specify a property identifier p^i for each time frame i in CNF indicating whether the property evaluation starting from that time frame leads to a failure. It is set to 0 if the property is violated and the T-MUX select line $e^i = 0$ (i.e. the transition relation is not skipped). At least one of the p^i, $0 \leq i < k$ must be set to 0 for the property to fail.

B. Properties with Timing Relationships

One of the most common usages of properties is to specify timing relationships between signals. For properties with timing windows we need to encode all possible clock cycle delays into CNF. As in the previous section property identifiers are used to evaluate the property instance starting at each timeframe.

Consider an assertion property which states that if $a = 1$ is followed within one to three cycles by $b = 1$, then c must be 1 in the next clock cycle:

@(**posedge** clk) (a ##[1:3] b |=> c);

The property identifier can be encoded as:

$$\begin{aligned}\overline{p^i} \leftrightarrow &((a^i \wedge b^{i+1} \rightarrow c^{i+2}) \vee (\overline{e^{i+3}} \wedge \overline{(a^i \wedge b^{i+2} \rightarrow c^{i+3})}) \\ &\vee (\overline{e^{i+3}} \wedge \overline{e^{i+4}} \wedge \overline{(a^i \wedge b^{i+3} \rightarrow c^{i+4})})) \wedge \overline{e^i} \wedge \overline{e^{i+1}} \wedge \overline{e^{i+2}}\end{aligned} \tag{5}$$

In order to ensure that a specific delay is met, none of the intermediate transition relations in the sequence can be skipped. Since at least 3 time frames are needed to determine whether the property fails, $e^i = 0$, $e^{i+1} = 0$, and $e^{i+2} = 0$. If additional clock cycles are used, the T-MUX select variables are constrained appropriately.

C. Properties with Repetition Specifications

Consider an assertion which checks that, if $a = 1$ then $b = 1$ will repeat three times (consecutively or intermittently) followed by $c = 1$ one clock cycle after:

@(**posedge** clk) (a |-> b [-> 3] ##1 c);

Since repetitions can occur intermittently there is no upper bound on the length of the sequence. Given a property identifier p^i which marks the beginning of the sequence we define a set of variables q_i^j marking all possible endings of the sequence. The property fails if one q_i^j is able to demonstrate the failure. Thus p^i is given as:

$$\overline{p^i} \leftrightarrow (a^i \wedge \overline{e^i}) \wedge (\bigvee_{i+3\leq j\leq k} q_i^j) \tag{6}$$

where

$$\begin{aligned}q_i^j \leftrightarrow &SORT(\{e_b^i, e_b^{i+1}, \ldots, e_b^{j-2}\}, \{s_2, s_1, s_0\}) \\ &\wedge (\overline{s_2} \wedge s_1 \wedge s_0) \wedge e_b^{j-1} \wedge (\overline{c^j} \wedge \overline{e^j})\end{aligned} \tag{7}$$

$$e_b^i \leftrightarrow (b^i \wedge \overline{e^i}) \tag{8}$$

The $SORT$ function constructs a sorter with three outputs $\{s_2, s_1, s_0\}$ and inputs $\{e_b^i, e_b^{i+1}, \ldots, e_b^{j-1}\}$. The sorter propagates all 1's at its input to the least significant bits of its output and all 0's to the most significant bits. The output is constrained to $\{s_2, s_1, s_0\} = 011$ for our example since the first two repetitions of $b = 1$ can occur anytime before the last repetition. The property fails if the last repetition of $b = 1$ is followed by $c = 0$ within the next clock cycle.

V. Experiments

In this section we demonstrate the effectiveness of our error trace compaction technique. The techniques described in this paper are implemented in C++ and use MiniSat2 as the underlying SAT solver [14]. All experiments are run on a 64-bit Quad Core Intel CPU @ 2.00GHz with 4GB of memory. In total three industrial circuits and four circuits obtained from OpenCores.org [15] are evaluated. Of these `mem2wire`, `wb_4m8s` and `spi` are communications cores. The circuits `rsdecoder`, `sudoku`, `pipeline`, and `ctrl` are a Reed-Solomon decoder, a sudoku solver, a data pipeline, and a traffic light controller respectively. The structural details of these circuits are shown in Table I.

TABLE I
Circuit Information for sample Designs

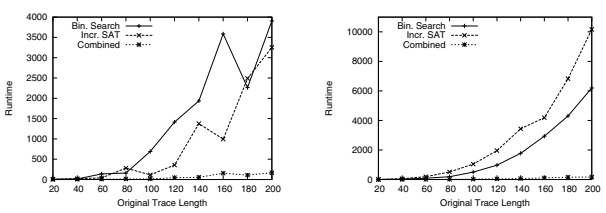
circuit	# gates	# inputs	# outputs	# state elements
mem2wire	14758	41	59	1338
rsdecoder	11353	9	10	521
sudoku	45340	15	1	356
wb_4m8s	19273	567	708	250
pipeline	4225	22	22	228
spi	2071	17	12	132
ctrl	2152	15	4	12

A comparison of the run time results of the various compaction techniques presented in this paper is given in Table II. The algorithm of [9] is provided as comparison as it represents the most recent optimal trace compaction technique developed. Column 1 in Table II gives the problem instance in the format:

<circuit_name>_<orig_trace_length>-<min_trace_length>

For instance, `mem2wire_20-6` states that we are compacting a trace for `mem2wire` from an initial trace length of 20 to a minimal trace length of 6. The *Binary Search* columns give the number of SAT instances solved and the total run time for the binary search technique from [9]. The *Incremental SAT* column presents the number of SAT iterations and run times using the formulation given in Section III-C without using the pre-processing step. The speedup of the proposed Incremental SAT technique over the basic binary search is given in column 6. The *Combined* column provides the run time information for using both the heuristics from Section III-C as well as the pre-processing step from Section III-B. The speedup over the basic binary search is given in column 9.

On average the incremental SAT technique achieves a run time improvement of 3×. However for circuits such as `rsdecoder`, `wb_4m8s`, and `pipeline` we see a consistent drop in performance



(a) mem2wire (b) wb_4m8s

Fig. 4. Run Time vs Original Trace Length for sample circuits

TABLE II
RUN TIME COMPARISON OF DIFFERENT COMPACTION TECHNIQUES

problem	Binary Search [9]		Incremental SAT			Combined		
	iterations	run time(s)	iterations	run time(s)	speedup	iterations	run time(s)	speedup
mem2wire_20-6	4	5.25	3	5.86	0.9	5	6.68	0.79
mem2wire_80-6	6	158.95	6	281.56	0.56	6	11.94	13.31
mem2wire_140-6	7	1935.15	7	1376.35	1.41	6	54.67	35.4
mem2wire_200-6	8	3907.14	8	3254.44	1.2	6	165.16	23.66
rsdecoder_20-14	5	11.9	4	6.71	1.77	5	7.3	1.63
rsdecoder_80-14	7	67.35	7	72.18	0.93	6	8.9	7.57
rsdecoder_140-14	7	249.56	7	520.85	0.48	6	37.14	6.72
rsdecoder_200-14	8	587	6	1274.52	0.46	7	68.26	8.6
sudoku_20-14	5	42.16	4	7.95	5.3	5	9.99	4.22
sudoku_80-14	7	224.44	7	23.22	9.67	6	86.88	2.58
sudoku_140-14	7	414.82	7	50.78	8.17	6	33.71	12.31
sudoku_200-14	8	793.79	7	52.9	15.01	7	142.33	5.58
wb_4m8s_20-4	5	5.22	4	5.43	0.96	4	1.61	3.24
wb_4m8s_80-4	7	188.86	6	514.81	0.37	5	12.56	15.04
wb_4m8s_140-4	7	1775.23	7	3434.96	0.52	5	62.46	28.42
wb_4m8s_200-4	7	6198.33	8	10171.02	0.61	6	168.64	36.75
pipeline_50-43	6	178.62	6	25.63	6.97	6	26.07	6.85
pipeline_200-43	8	486.23	8	234.41	2.07	7	278.73	1.74
pipeline_350-43	9	1758.61	9	986.46	1.78	7	129.89	13.54
pipeline_500-43	9	3735.8	9	2023	1.85	8	321.98	11.6
spi_50-47	6	37.4	6	9.85	3.8	6	10.4	3.6
spi_200-47	8	157.57	7	111.76	1.41	8	120.72	1.31
spi_350-47	9	658.9	8	445.02	1.48	8	46.46	14.18
spi_500-47	9	2402.99	9	1140.54	2.11	8	132.67	18.11
ctrl_100-16	6	2.13	7	1.16	1.84	6	0.25	8.52
ctrl_400-16	8	88.26	9	21.84	4.04	7	5.2	16.97
ctrl_700-16	9	418.64	10	78.22	5.35	8	21.92	19.1
ctrl_1000-16	10	988.33	10	127.97	7.72	8	79.25	12.47
				Average:	**3.17**		Average:	**11.92**

relative to [9] as the initial trace length is increased. This can be attributed to the smaller size of the SAT problems solved by the basic binary search. For instance while Incremental SAT is solving an ILA expanded over 100 time frames, the Binary Search technique is solving an ILA with only 50 time frames.

The pre-processing step from Section III-B compensates for this disadvantage. On average we were able to observe a $12\times$ speedup in run time using $f = 0.2$ as the reduction factor. Especially as h is increased relative to the minimum trace length, the pre-processing step helps reduce the number of SAT instances solved while also decreasing the size of the SAT problems. A graphical representation of the changes in run time as the initial trace length increases for the circuits `mem2wire` and `wb_4m8s` is given in Figure 4.

Table III compares the Incremental SAT compaction technique with and without using the connectors described in Section III-C. Only a representative sample of the data is given. In almost all cases the heuristic allowed for a significant speedup in run time with an average improvement of almost $3\times$.

TABLE III
COMPACTION WITH AND WITHOUT CONNECTORS

problem	Without Connectors		With Connectors		
	iterations	run time(s)	iterations	run time(s)	speedup
mem2wire_100-6	7	600.75	7	115.08	5.22
rsdecoder_100-11	6	248.37	6	141.82	1.75
sudoku_100-14	6	48.57	6	30.41	1.6
wb_4m8s_100-4	7	1274.35	7	1044.56	1.22
pipeline_250-43	6	842.3	8	411.91	2.04
spi_250-47	8	320.15	8	142.43	2.25
ctrl_500-16	8	199.23	9	30.47	6.54
				Average:	**2.95**

We also implemented our trace compaction technique from Section III-B using SVA properties instead of final state constraints. Table IV provides the size of the assertions relative to some of the problems. The number of assertion clauses and literals is given in columns 4 and 5 with the % overhead over the total problem size given in parentheses. Note that the problems in this table are different from the problems in Table II since the circuits are modified to ensure that different assertions fail. All circuits presented in Table II use at least one implication operator along with other operators presented in Section IV in their SVA expressions. Compared to the total size of the CNF problem, the CNF for the assertions is relatively small and do not exceed 1.5% of the total problem size.

VI. CONCLUSIONS

In this paper, we describe an efficient trace compaction technique using incremental SAT which guarantees an optimal compacted trace. The proposed technique introduces a formulation, algorithms and heuristics to improve the average run time performance by $12\times$ on average compared to a previous optimal compaction technique.

TABLE IV
TRACE COMPACTION WITH SVA

problem	# problem clauses	# problem literals	# assertion clauses	# assertion literals	run time(s)
spi-sva1_50-21	343832	884264	342 (0.10%)	802 (0.09%)	3.44
spi-sva2_100-82	679036	1752772	1382 (0.20%)	3253 (0.19%)	76.23
pipeline-sva1_50-30	664842	1683984	5664 (0.84%)	13240 (0.78%)	18.62
pipeline-sva2_100-22	1355844	3421588	692 (0.05%)	1627 (0.05%)	30.78
ctrl-sva1_50-9	129224	298698	1832 (1.40%)	4288 (1.42%)	0.38
ctrl-sva2_100-9	219178	516956	692 (0.31%)	1627 (0.31%)	1.37

This work also introduces the concept of assertion based error trace compaction which uses SVA to ensure that properties are preserved from the original trace. Encodings for common property types are given and experiments demonstrate that the overhead for encoding assertions in CNF is relatively small ($< 1.5\%$).

REFERENCES

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

[2] International Techonology Roadmap for Semiconductors, "ITRS 2006 Update," 2008, http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm.

[3] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

[4] K. Chang, V. Bertacco, and I. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *Int'l Conf. on CAD*, 2005, pp. 1045–1051.

[5] S. Safarpour, A. Veneris, and H. Mangassarian, "Trace compaction using sat-based reachability analysis," *ASP Design Automation Conf.*, pp. 932–937, 2007.

[6] Y. Chen and F. Chen, "Algorithms for compacting error traces," in *ASP Design Automation Conf.*, 2003, pp. 99–103.

[7] S.-J. Pan, K.-T. Cheng, J. Moondanos, and Z. Hanna, "Generation of shorter sequences for high resolution error diagnosis using sequential sat," in *ASP Design Automation Conf.*, 2006, pp. 25–29.

[8] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," in *Advances In Computers*, 2003.

[9] Y. Chia-Chih, "An optimum algorithm for compacting error traces for efficient design error debugging," *IEEE Trans. Comput.*, vol. 55, no. 11, pp. 1356–1366, 2006.

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

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

[12] F. Fallah, "Binary time-frame expansion," in *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002, pp. 458–464.

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

[14] N. S. N. Een, "An Extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 333–336.

[15] OpenCores.org, "http://www.opencores.org," 2008.