Merging Nodes Under Sequential Observability

Michael L. Case[1,2] Victor N. Kravets[3] Alan Mishchenko[1] Robert K. Brayton[1]

[1] Department of EECS, University of California at Berkeley, CA
[2] IBM Systems and Technology Group, Austin, TX
[3] IBM TJ Watson Research Center, Yorktown, NY

ABSTRACT

This paper presents a new type of sequential technology independent synthesis. Building on the previous notions of combinational observability and sequential equivalence, sequential observability is introduced and discussed. By considering both the sequential nature of the design and observability simultaneously, better results can be obtained than with either algorithm alone. The experimental results show that this method can reduce the technology-independent gate count up to 10% more than the previously best known synthesis techniques.

Categories and Subject Desc. J.6 [Computer-aided design]: CAD
General Terms Algorithms, Design, Performance, Verification
Keywords Sequential, Observable, Merge, Synthesis

1. INTRODUCTION

In verification it is important to reduce the size of the technology independent model before using model checking [1]. In synthesis, technology independent optimizations are used before technology mapping, and there is a strong correlation between circuit sizes before and after mapping [2]. In this work we propose a method to minimize the number of nodes in the and-inverter graph (AIG) representation of a technology independent sequential design.

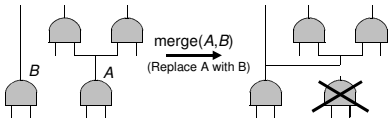

Figure 1: Merging signals A and B by rewiring and removing dangling nodes.

Consider merging signals as a basic synthesis operation. This operation, illustrated in Figure 1, involves rewiring such that the fanouts of a signal are driven by a different signal. This leads to circuit simplification as 1) one of the signals can be removed from the circuit and 2) the merge may cause downstream logic to appear redundant. This merge operation can be applied in multiple contexts to form the basis of many synthesis algorithms.

DAC 2008, June 8–13, 2008, Anaheim, California, USA.

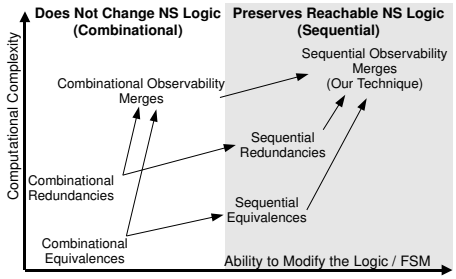

Figure 2: Taxonomy of merge-based optimizations. Arrows show that an algorithm generalizes another.

In this work we consider merging signals under *sequential observability*. We consider the combinational outputs or COs (primary outputs and register inputs) as being more significant than internal signals, and we change the internal signals freely while preserving the CO values on reachable states. We call this method *sequential observability* because it leverages both the sequential nature of the design as well as observability don't cares within the design. We merge signals A and B if in every reachable state either $A = B$ or the merge does not produce a difference at any CO. This guarantees sequential equivalence [3] between the original and optimized design while providing significant flexibility for optimizations in the interior of the design.

In this paper we present a framework for merging nodes under sequential observability. Basic techniques and optimizations are discussed, and the method is examined on a variety of academic and industrial benchmarks.

2. BACKGROUND AND RELATED WORK

Synthesis algorithms based on merging signals have been extensively studied in the past, as examined in Figure 2. Significant circuit optimizations can be realized using merge-based algorithms, but there is more potential for optimization with this simple yet powerful operation. All previous approaches are either limited in scope or do not take advantage of the sequential nature of the design.

SAT and BDD sweeping are two early algorithms based on merging signals [4, 5, 6, 7, 8, 9]. These algorithms attempt to prove that pairs of signals are combinationally equivalent and then merge the pairs successfully proved. These algorithms are used extensively in modern synthesis and verification frameworks [1, 10] because they scale well and provide significant reductions in the number of AIG nodes.

Generalizing on the previous algorithms, [11] finds sequentially equivalent signals by proving that pairs of signals are equivalent in every reachable state, a superset of the

merges found by SAT and BDD sweeping. The success of this method depends on the design style, but in general the method is effective in reducing the AIG node count.

Techniques based on redundancies represent another class of merge-based algorithms. These techniques prove that a signal, not necessarily constant, can be safely merged with a constant without producing a difference at the COs, improving the AIG node count by enabling constant propagation. Combinational redundancy methods are used also in industrial flows to enhance stuck-at-testability [12].

Recently there has been a renewed interest in combinational observability merges [13, 14]. This class of algorithms finds ordered pairs of signals that may not be strictly equivalent yet will not produce a difference at the COs after merging. This takes advantage of logic reconvergence that masks out the signal differences as the values propagate toward the outputs. The merges found by this method are a superset of those found by SAT/BDD sweeping but are in general a different superset than sequentially equivalent signals. These merges are also a superset of those found by combinational redundancy methods.

In this paper we propose a method to find merges under sequential observability. We find ordered pairs of signals that are either equivalent in every reachable state or do not result in an observable difference at a circuit CO in a reachable state after merging. In this way, our work can be viewed as a combination of sequentially equivalent signals and combinational observability merges, hence the term *sequential observability*. The merges found by our technique a superset of all merges found by every previous algorithm. In this paper we demonstrate that introducing sequential analysis into an observability-based algorithm can greatly improve its potential to optimize a logic network.

This work is limited in the sense that it will always preserve the values of the next state functions on the reachable states. Viewed from a state transition graph (STG) perspective, our method would preserve the reachable part of the STG exactly, unlike some sequential synthesis methods (eg. retiming [15]). In this work we limit ourselves for computational reasons and cannot claim to utilize full sequential observability, just a limited form of it. This restriction may hurt our results but introduces several nice synthesis properties. Namely, the scan chain is preserved and post-silicon debugging is not complicated by a change in latch functions.

3. MERGING NODES UNDER SEQUENTIAL OBSERVABILITY

The merge-based synthesis methods discussed in Section share a common algorithmic flow. They are all possible to implement by following this simple outline:

Derive Candidate Merges → Prove that Merges are Valid → Simplify Circuit

In this section we will discuss how to specialize these blocks for sequential observability.

3.1 Finding Merge Candidates

In the first step of our algorithm we would like to quickly find a set of candidate merges that is a superset of the CO-preserving merges. That is, we would like to find ordered pairs of signals (A, B) such that it is likely that for all reachable states S that one of the following is true:

- $A = B$ in state S.
- Merging A and B will not produce a difference at a CO in S.

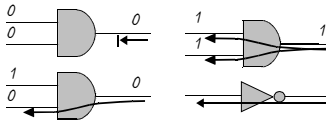

Figure 3: Propagating the controlling path backward through an AIG.

In the language of [13, 14] this is: for all reachable states, either $A = B$ or B is not observable. The merge operation is free to replace A with B because B being unobservable implies that the difference will not be visible at a CO.

To quickly find a set of candidate merges we use a slightly weaker notion than observability: whether or not a signal lies on a CO's controlling path. If an AIG has been simulated for a single concrete input then the controlling path can be propagated backwards through the AIG as shown in Figure 3. Note that in the $0 \cdot 0 = 0$ case we choose to conservatively conclude that neither of the inputs are observable, making the marked controlling paths smaller but guaranteeing that all marked signals have the ability to influence the value at a CO. Toggling any marked controlling path signal will cause at least one CO to toggle. The controlling signals are therefore observable at the COs.

Algorithm 1 Extracting candidate merges from simulation.

```
1: // Given "design" and a number of reachable input "states"
2: sim = simulateDesign(design, states);
3: controlling = extractControllingPaths(design, sim);
4: for all pairs of signals (A, B) do
5:    if (sim.A == sim.B) || ! controlling.B then
6:       (A, B) is a candidate merge;
7: end for
```

Our method to find merge candidates is shown in the pseudocode of Algorithm 1. We simulate the circuit with a number of known-reachable input vectors and extract the controlling paths. Then we use the simulation and controlling path information to check for each ordered pair of signals (A, B) if $A = B$ or B is not observable. For simplicity, an $O(n^2)$ algorithm is presented here, but an improvement is discussed in Section 4.3.

An example of this method for extracting candidate merges is illustrated in Figure 4. It can be proved that if the initial state is $BC = 00$ then it is possible to merge such that AB is replaced with $\overline{A \oplus C}$:

- $BC = 10$ is unreachable.
- $\overline{A \oplus C} + AB = \overline{A \oplus C}$ for every reachable states
- It is safe to replace AB with $\overline{A \oplus C}$ at Z's OR gate.

This merge cannot be an equivalence because the signals have differing support. It is not a combinationally observable merge because we need to know that $BC = 10$ is not reachable. This merge can only be found using sequential observability.

We can algorithmically discover this candidate merge by simulating the initial state $BC = 00$ along with a randomly selected value for the input A. Then highlight the controlling paths that lead up to each CO. According to the simulation information $AB \neq \overline{A \oplus C}$, but AB is not on the controlling path to any CO. Therefore the ordered pair of signals $(AB, \overline{A \oplus C})$ becomes a candidate for merging. Altering the value of the AB signal will not produce a difference at any CO.[1]

3.2 Proving Candidates

[1]The sequentially observable merge $(Z, \overline{A \oplus C})$ is equivalent.

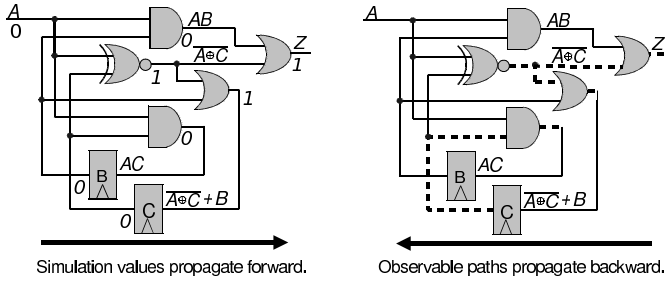

Figure 4: Discovering sequential observable merge candidates. (While we use AIGs, rich gate types are shown for compactness.)

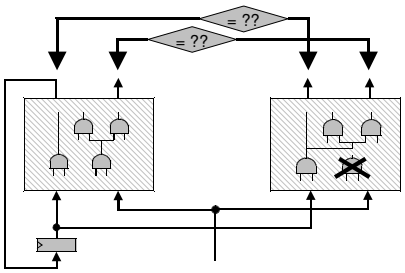

Figure 5: Proving that a set of merges does not change any CO.

Algorithm 2 Proving merges valid.

```
1: // Given "candidates" merges on "design"
2: while (1) do
3:    mod = mergeCandidates(design, candidates);
4:    miter = compareOutputs(design, mod);
5:    if (checkInduction(miter)) then
6:       break ;
7:    else
8:       pruneCandidates(design, candidates, getCex());
9:    end if
10: end while
```

In order for our synthesis algorithm to be sound, we must prove that for all reachable states, modifying the circuit with the candidate merges will not produce a difference at a CO. This will be formulated as a set of properties that are then proved by induction on the state space.

The correctness and compatibility of a set of candidate merges can be checked by building a miter circuit similar to Figure 5. Two copies of the circuit are compared: the original circuit and a copy that has been simplified with the candidate merges. These two circuit copies are driven by the same inputs and state, and logic is synthesized to check that the COs are the same.

Algorithm 3 Discarding bad merge candidates.

```
1: // Given "counterexample", "candidates", and "design"
2:
3: // fast but incomplete method (checks correctness)
4: Simulate counterexample and extract controlling paths;
5: for all (A, B) ∈ candidates do
6:    if (sim.A ≠ sim.B) && controlling.B) then
7:       discard (A, B);
8: end for
9: if (∃ discarded candidate) then return ; // Skip slow step.
10:
11: // slow but complete method (checks compatibility)
12: stack = (candidates);
13: goodCands = nil;
14: while (!stack.empty) do
15:    currCands = stack.pop();
16:    mod = design with merged goodCands and currCands;
17:    Simulate design and mod with counterexample;
18:    if (∃ output o s.t. sim.design.o ≠ sim.mod.o) then
19:       if (currCands.size() == 1) then
20:          discard currCands; // Drop single culprit.
21:       else
22:          cand1, cand2 = divideInHalf(currCands);
23:          stack.push(cand1, cand2); // Divide and conquer.
24:       end if
25:    else
26:       goodCands += currCands;
27:    end if
28: end while
```

Note that constraining the merges such that the next state is not altered greatly simplifies the miter circuit. Without this constraint each circuit in Figure 5 would have an independent state, doubling the number of latches in the miter. Here we choose to simplify our miter and subsequent proof, possibly at the expense of the optimization results. This constraint also implies that the design's reachable STG will not be altered.

The miter circuit can be viewed as a single sequential machine with a set of properties to be checked, one for each CO equivalence. We prove these properties using 1-step induction [16]. This is an incomplete unbounded verification technique that can prove some of the properties true for all reachable states.

Algorithm 2 illustrates how induction is used to find a subset of CO-preserving candidate merges. Counterexample states are produced where the two circuits in the miter are not equivalent. By removing the merges that caused the miscompare and trying induction again, a greatest fixed point algorithm is developed that will produce a set of merges that when applied will yield an simplified equivalent circuit.

One significant challenge is to determine which candidate merges are faulty given the fact that the simplified circuit produces a differing output. This task is performed by the `pruneCandidates()` function, outlined in Algorithm 3. Two methods are utilized to find the merges responsible for the output mismatch.

The first (lines 3-9) is a fast method that only checks that each merge is individually correct, similar to Section 3.1. This method is fast but incomplete because merges can interact with each other, and a set of merges being individually correct is no guarantee they are compatible as a group. This method used as a fast filter before calling the second, more expensive, method.

The next method (lines 11-28) will check the compatibility. It does this by performing a binary search over subsets of the candidates until it finds individual candidate merges that cause a local reduced model to have a differing circuit output. It is able to check compatibility but is slow because for each subset it must simulate a reduced circuit model. The fast method is used whenever possible, and the more expensive method is only used when the first method fails to find a guilty candidate.

These two methods together will greedily isolate the first compatible set of merges that is preserves the COs under the given counterexample. There are often multiple compatible subsets which can be exploited by applying our proposed synthesis algorithm multiple times.

3.3 Using the Candidates

We often need to simplify a circuit using a set of candidate merges. This is done both in checking the merges as in Figure 5 and also in constructing the final simplified circuit.

Table 1: Candidate Merge Statistics

Design	Merges	Redund. Nodes	Choices / Redund.	Avg. Level For Red.	Avg. Level For Rep.
ibm4	3215	1498	616	5.1	7%	21%
ibm5	2469	715	409	4.9	11%	27%
ibm6	2167	673	381	4.7	12%	17%
ibm7	4323	2668	705	3.7	12%	37%
ibm9	8688	3478	1593	4.1	11%	29%
			0.45	4.5	10%	26%

Producing a simplified circuit from a set of candidate merges is not simple. Table 1 gives statistics on the set of candidate merges as examined periodically throughout a run of our tool. Each candidate merge is an ordered pair of signals (*redundant*, *replacement*) where *redundant* will be replaced with *replacement*. On average, the candidate merges suggest replacements for approximately 45% of the design nodes, and for each redundant node there are an average of 4.5 candidate replacements. Selection of the replacement to use has a large impact on ability of our algorithm to minimize the AIG size, and heuristics that select the replacement to use are important.

In selecting a replacement for a redundant node, it is very important to not introduce a combinational cycle. Table 1 shows that on average for a circuit with N levels, the redundant node is at level $10\% \cdot N$, and the replacement is node is at level $26\% \cdot N$. Since a node will often be replaced with one of a higher level, we can expect combinational cycles to abound. This also means that the upper 74% of the circuit usually has few candidate merges, a potential basis for future heuristics research.

Experimentally we found that the best performance could be obtained not by replacing the entire signal *redundant* with *replacement* but by selectively replacing each of the fanouts of *redundant*. This introduces the flexibility to replace each fanout with a different signal, thereby improving the optimization potential. It also provides an easy way to handle combinational cycles; if doing replacing *redundant* with *replacement* would introduce a cycle and *redundant* is a multi-fanout net then the replacement can still occur on the subset of the fanouts not involved in the cycle. This generalization greatly improved the quality of our synthesized designs.

The heuristic used in this work is shown in Algorithm 4. Each candidate merge is passed to this routine to incrementally simplify the network. The general strategy used is to 1) avoid cycles, 2) enable constant propagation, 3) remove fanouts from low-fanout nodes in the hope that they can be removed after simplification, 4) greedily enable the maximum amount of structural hashing benefits, 5) as a tie breaker, remove fanouts from nodes that have a low ID.

Algorithm 4 Using a merge to simplify the logic network.

```
1: // Given a merge with signals (redundant, replacement)
2: for all fanouts out of redundant do
3:   if (replacement ∈ trans_fanout_cone(out)) then
4:     Drive out with redundant    // Avoid a cyclic circuit
5:   else if (redundant or replacement is constant) then
6:     Drive out with (constant)    // Propagate constants
7:   else if (redundant or replacement has ≤ 2 fanouts) then
8:     Drive out with (higher fanout) // Remove low-fanouts
9:   else if (struct_hash_gain(replacement → out) ≠
           struct_hash_gain(redundant → out)) then
10:    Drive out with (better gain)  // Maximize hash gain
11:  else if (num_fanouts(replacement) ≠
           num_fanouts(redundant)) then
12:    Drive out with (higher fanout) // Remove low-fanouts
13:  else
14:    Drive out with (higher node ID)   // Remove low-IDs
15:  end if
16: end for
```

4. ENHANCING SCALABILITY

The method outlined above works well for small benchmarks, but in order for it to scale to larger benchmarks several tricks were employed. In this section we document the ways in which our method was made significantly more scalable.

4.1 Simulation Quality

In Section 3.2, candidate merges are extracted from a the simulation of a set of known-reachable states. Choosing a high quality set of simulation inputs is key to the success of this method.

A naïve implementation might do a random walk through the state space starting from an initial state, collecting reachable states to serve as input stimulus. When considering observability, we require simulation inputs that allow many internal signals to propagate to the COs. The vectors must sensitize interesting controlling paths, and vectors from a random walk do not do this well.

Table 2: Candidates In The IBM Benchmarks

Design	Sig. Pairs Tested (Using BK-Trees)	Number of Candidate Merges Random Walk	Number of Candidate Merges Redund. Cex's
ibm4	19.29%	1,274,272	16,217
ibm5	42.03%	6,577	8,460
ibm6	27.40%	4,556	4,721
ibm7	24.25%	1,766,864	101,409
ibm9	28.50%	20,429	25,648

What is needed is a set of vectors much like what one would get from a sequential stuck-at test generator. Unfortunately, this problem is intractable for large sequential designs [17], but in this context there is a simpler way to get meaningful vectors.

Suppose we attempt to find all sequential redundancies in a circuit, a special case of sequential observability merges where we only try to merge signals with constants. Such merges are usually falsified, but in attempting to prove these merges a set of reachable counterexamples will be generated. Each counterexample will set up a value on an internal signal and allow that value to propagate to a CO, exactly what is needed to set up interesting controlling paths.

Table 2 shows the number of candidate merges after execution of Algorithm 1. One column shows the results from simulating states encountered on a random walk from an initial state. Another column shows the results from simulating counterexample states encountered during a sequential redundancy elimination pass. In some cases, the counterexamples much more effectively sensitize interesting controlling paths and can give several orders of magnitude fewer candidate merges. Without the counterexamples, the high number of candidate merges would cause our algorithm to exhaust both the available time and memory.

4.2 Use of SemiFormal Analysis

In our implementation, each counterexample is precious and is stored. If the synthesis algorithm is based on induction there is no guarantee that a counterexample will be a reachable state. Non-reachable states will cause Algorithm 1 to miss candidate merges, so counterexamples mot guaranteed to be reachable must be discarded. Therefore it is important to utilize techniques that guarantee that all coun-

terexamples are reachable.

Semiformal analysis, first described in [18], is a way of interleaving random simulation with bounded model checking to form a sound but incomplete bug-finding technique that has better coverage than simulation alone and better depth than BMC alone. All counterexamples found by semiformal analysis are guaranteed to be reachable. In our implementation of all synthesis algorithms, semiformal analysis is always used before induction. Calling induction alone is often much faster but will not provide the same quality of counterexamples. As described in Section 4.1, counterexample quality is key to the scalability of our method.

4.3 Avoiding $O(n^2)$ Candidate Merge Checks

In the method to find candidate merges given in Section 3.1, each pair of signals in the design is examined. The resulting $O(n^2)$ complexity is a major problem on designs with a large number of signals n.

Papers discussing combinational observability merges [13, 14] have proposed heuristics to reduce the average complexity of this step, but these heuristics are inadequate because they fail to handle a large number of simulation vectors. In this section, we discuss a method to reduce this $O(n^2)$ complexity while handling large amounts of simulation data resulting from our semiformal analysis.

Burkhard-Keller (BK) Trees are an algorithm and datastructure used to quickly find points in a space that are "close" to a given reference point [19]. More formally, given a set of points, a metric $d(\cdot,\cdot)$, a point A, and a constant ϕ, BK-Trees can be used to find points B such that $d(A,B) \leq \phi$. BK-Trees exploit the triangle inequality to bound the search for feasible B's and only need to touch a small part of the search space to answer the query.

BK-Trees can be utilized to speed up the search for candidate merges. Given design signals A and B with simulation vectors $sim.A$ and $sim.B$, define the pseudometric $d(A,B)$ as $d(A,B) = ||sim.A \oplus sim.B||$ where $||\cdot||$ denotes the number of ones in a vector. This is not a true metric because it is not positive definite (referred to as a pseudometric [20]), but BK-Trees do not rely on positive definiteness and this pseudometric can be safely used. With formalism, we fix node B and utilize $controlling.B$, a mask expressing for which simulation vectors B is controlling, to find the A's such that $d(A,B) \leq ||!controlling.B||$. The set of satisfying A's is a superset of the set of A's satisfying $||(sim.A \oplus sim.B) \cdot controlling.B|| = 0$ and therefore a superset of the A's that can be safely merged with B.

In this way, BK-Trees are used to narrow the search for B's that can be merged with a given node A. The effectiveness of this approach is shown in Table 2 where the percent of the search space that was examined is shown in the column "Signal Pairs Tested." While this doesn't reduce the search space exponentially, it is effective in improving the runtime in practice. Furthermore, BK-Trees are easy to implement and a query over a BK-Tree takes almost constant time.

4.4 Selection of Proof Technique

In Section 3.2, the candidate merges are checked with a miter circuit and a series of sequential properties that can be proved using any unbounded technique. In this work we have chosen to use induction because it scales well, but it is well known that induction is an incomplete technique that is not able to prove all true properties [21]. In a verification domain this behavior is not desirable because of the requirement that all properties be proved. In a synthesis domain, this incomplete behavior is an acceptable trade-off for the scalability of induction. Merges disproved by induction are dropped, and because of the incompleteness some correct merges might be dropped along with the incorrect ones, reducing synthesis results.

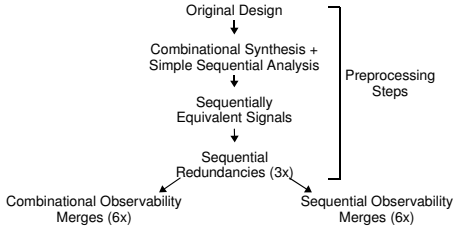

Figure 6: Algorithmic flow used in our experiments.

We experimented with stronger induction formulations, namely K-step induction with unique state constraints [16]. Increasing K rarely improved our optimized AIG node count, and increasing K beyond 1 significantly hurt the runtime of our method. Because of the near-independence of the results from the complexity of the induction formulation, in our implementation we always use simple or $K = 1$ induction.

5. EXPERIMENTAL RESULTS

Sequential observability and all other algorithms in Figure 2 were implemented in C++ inside of a synthesis and verification environment. The environment uses ABC [10] for combinational synthesis and MiniSat [22] for SAT solving.

Combinational observability merges as presented in [13, 14] is the best merge-based synthesis algorithm in the literature to date. We will compare our work to combinational observability on 3 benchmark suites: a set of processor blocks from IBM Corporation, the ISCAS '89 benchmarks, and a selection of blocks from the Sun PicoJava processor [23].

To fairly compare combinational and sequential observability, each design was first heavily synthesized as shown in Figure 6. Combinational synthesis, including SAT sweeping and rewriting [24] along with simple techniques to find structurally equivalent latches and sequentially constant latches [25] was first applied. This was followed by processing of sequential equivalences and sequential redundancies. The sequential redundancy algorithm was run 3 times in order to build a high quality set of simulation vectors as discussed in Section 4.1. The designs at this point are labeled as "Preprocessed" in Tables 4 and 3. Finally, on two separate runs the preprocessed design was optimized using either combinational observable merges or sequentially observable merges. The observability algorithms were run 6 times to allow iterative circuit gain to saturate so that the maximum cumulative gain could be measured.

Results on the two sets of industrial benchmarks and the one set of academic benchmarks are shown in Table 3. For the columns labeled "Preprocessing" the numbers of AIG nodes and latches are given relative to the original design. In the sequential observability and combinational observability columns, the AIG nodes and latches are given relative to the preprocessed design. Sequential observability is able to reduce the node counts of the preprocessed designs by about 6% and the latch counts by about 3%. The node count reduction is dramatically more for sequential observability

Table 3: Performance Across Benchmark Suites

Benchmark Suite				Preprocessing		Additional Sequential Observability Benefits		Additional Combinational Observability Benefits	
Suite	# Designs	Avg. Latches	Avg. Ands	Latches	Ands	Latches	Ands	Latches	Ands
IBM	5	248	2384	0.98	0.82	0.99	0.96	1.00	1.00
ISCAS89	28	109.39	751.64	0.95	0.72	1.00	0.90	1.00	1.00
PicoJava	64	627.83	2943.55	0.87	0.69	0.92	0.95	0.92	1.00

Table 4: Performance On The IBM Benchmarks

	Original		Preprocessing			Additional Sequential Observability Benefits			Additional Combinational Observability Benefits		
Benchmark	Latches	Ands	Time	Latches	Ands	Time	Latches	Ands	Time	Latches	Ands
ibm4	255	1845	49.36	253	1539	136.2	252	1453	23.64	253	1539
ibm5	93	925	7.72	93	738	42.51	93	679	5.4	93	738
ibm6	151	811	8.24	142	657	30.35	140	651	7.49	142	657
ibm7	428	3173	57.34	426	2684	302.15	422	2628	169.13	426	2684
ibm9	354	3896	21.9	353	3482	167.49	352	3447	103.35	353	3470
				0.98	0.82		0.99	0.96		1.00	1.00

than it is for combinational observability, and this indicates that reachable states are a very important degree of freedom to consider in an observability-based algorithm. It is also interesting that while this technique does not directly target latch reductions, occasionally all fanouts are removed from a latch, causing the latch to be removed.

Detailed runtimes for the IBM designs are shown in Table 4. Sequential observability is slower than combinational observability, but this slowdown is expected because we must check the property inductively across two time frames instead of combinationally in just one time frame. The slowdown is not severe, and the sequential case is still scalable and not likely to substantially increase the total runtime of industrial tools.

In our experiments, the combinational observability method did very little on average. We are starting from a heavily synthesized design point, and combinational observability is not able to improve upon this design point further. This indicates that the types of optimizations done by combinational observability are contained in all of the preprocessing that has been done (and the preprocessing was much cheaper). This is not true of sequential observability. Our method was able to improve upon the preprocessed designs, sometimes significantly. In this way, sequential observability is a powerful optimization whose design simplifications fall outside the scope of all other synthesis algorithms examined in this paper.

6. CONCLUSION

In this work we explored merging signals under a form of sequential observability. Pairs of signals are merged such that either: 1) each pair is equivalent in every reachable state, or 2) the merge does not produce a difference at a circuit CO. In this way, the internal signals in a design are free to change so long as the COs are preserved on the reachable states. This flexibility allows for synthesis optimizations, enabling lower gate counts than previous synthesis methods.

Experimental results show that the number of AIG nodes can be reduced by up to 10% in heavily optimized designs, and the method is scalable to industrial designs.

7. REFERENCES

[1] J. Baumgartner, "Integrating FV Into Main-Stream Verification: The IBM Experience," Tutorial Given at *FMCAD* 2006.

[2] S. Chatterjee, A. Mishchenko, R.K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *ICCAD* 2005.

[3] M.N. Mneimneh and K.A. Sakallah, "Principles of sequential-equivalence verification," in *IEEE Design and Test Comp*, 2005.

[4] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *DAC 1997*.

[5] D. Brand, "Verification of large synthesized designs," in *ICCAD 1993*.

[6] W. Kunz, "HANNIBAL: An efficient tool for logic verification based on recursive learning," in *ICCAD 1993*.

[7] W. Kunz, D. Stoffel, and P. Menon, "Logic optimization and equivalence checking by implication analysis," in *ICCAD 1997*.

[8] E. Goldberg, M.R. Prasad, and R.K. Brayton, "Using SAT in combinational equivalence checking," in *DATE 2001*.

[9] F. Lu, L.C. Wang, K.T. Cheng, and R.C.Y. Huang, "A circuit SAT solver with signal correlation guided learning," in *DATE 2003*.

[10] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/

[11] C. van Eijk, "Sequential equivalence checking based on structural similarities," in *IEEE Trans. Computer-Aided Design*, July 2000.

[12] S. Devadas, A. Ghosh, and K. Keutzer, "Logic Synthesis," McGraw-Hill 1994.

[13] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't cares," in *DAC 2006*.

[14] S. Plaza, K.H. Chang, I. Markov, and V. Bertacco, "Node Mergers in the Presence of Don't Cares," in *ASPDAC 2007*.

[15] C.E. Leiserson and J.B. Saxe, "Retiming synchronous circuitry," in *Algorithmica* 1991.

[16] P. Bjesse and K. Claessen, "SAT-based Verification without State Space Traversal," in *FMCAD* 2000.

[17] H.K. Ma, S. Devadas, A.R. Newton, and A. Sangiovanni-Vincentelli, "Test generation for sequential circuits," in *CAD of Integrated Circuits and Systems* 1988.

[18] P.H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *ICCAD 2000*.

[19] W.A. Burkhard and R.M. Keller, "Some approaches to best-match file searching," in *Communications of the ACM* 1973.

[20] L.A. Steen and J.A. Seebach, "Counterexamples in Topology," Courier Dover Publications 1970, Page 34.

[21] M.R. Prasad, A. Biere, and A. Gupta, "A Survey of Recent Advances in SAT-Based Formal Verification," in *Software Tools for Technology Transfer* 2005.

[22] Niklas Een, Niklas Sorensson, MiniSat. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

[23] Sun Microsystems, "Processor Technology Resources - picoJava," http://www.sun.com/software/communitysource/processors/picojava.xml

[24] A. Mishchenko, S. Chatterjee, and R.K. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC* 2006.

[25] P. Bjesse and J. Kukula, "Automatic Phase Abstraction for Formal Verification," in *ICCAD* 2005.