Resolution Proofs as a Data Structure For Logic Synthesis

John D. Backes and Marc D. Riedel

Department of Electrical and Computer Engineering
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455
{back0145, mriedel}@umn.edu

***Abstract*—The results of modern synthesis tools are heavily influenced by the initial structure of the designs they are given. Many synthesis tools use And-Inverter Graphs (AIGs) as the underlying representation in the technology-independent phase of synthesis. Generally, AIGs can be compacted through a series of operations that merge equivalent nodes. Proving nodes to be equivalent can be a done with efficient SAT-based algorithms. However, when an AIG consists of very sparse logic, compaction is ineffective; few equivalent nodes are found.**

Recently, techniques for generating functional dependencies via Craig Interpolation have been proposed. Such methods first generate a resolution proof from a SAT instance. Interpolation is performed on the resolution proof to generate the dependency function. Unlike nodes in an AIG, determining whether an equivalent resolution node can be generated from other resolution nodes is a simple task. In this work, we study the use of resolution proofs as an underlying data structure for performing technology-independent synthesis, as opposed to just the front-end step. In order to represent multiple functions, we merge separate resolution proofs into a single, monolithic resolution proof and then perform restructuring operations. We analyze the effectiveness of our method on standard benchmark circuits. The results suggest that resolution proofs could be a very effective data structure for representing multiple target functions.

I. Introduction

Many problems in logic synthesis and verification can be naturally translated to Boolean Satisfiability (SAT). Modern SAT solvers are remarkably efficient computational engines, solving problems with thousands of variables with ease. When a SAT instance is satisfiable, the solver returns a satisfying assignment of the instance's variables. When a problem is unsatisfiable, the solver returns a *resolution proof* of unsatisfiability.

Algorithms based on Craig Interpolation exploit these resolution proofs to generate Boolean functions that satisfy certain properties [6], [10], [13]. These Boolean functions are referred to as *Craig Interpolants* or just *interpolants*. Such interpolants can be used to generate functional dependencies [1], [9]. With this approach, a SAT instance is created that asks whether or not a target function can be implemented with a specified support set. If the instance is unsatisfiable, the answer to this question is affirmative. The interpolant generated from the resolution proof of unsatisfiability is the target function; its support set contains variables in the target support set. The structure of this interpolant is heavily influenced by the structure of the resolution proof. The structure of the resolution proof, in turn, is heavily influenced by the order of decision variables used by the SAT solver.

When interpolation is used to generate functional dependencies, often the goal is to generate dependencies for multiple target functions. In this case, target functions that are able to share gates in their transitive fanin are ideal. However, large differences in the resolution proofs can lead to little logic sharing between interpolants. Consider the example illustrated in Figures 1 and 2. Both resolution proofs have many of their root clauses in common. However, the resolution proofs in Figure 1 have few of their intermediate clauses in common. As a result, after the interpolants are generated for each proof, none of the gates compute the same Boolean function. In contrast, the resolution proofs in Figure 2 share many of the same intermediate clauses. The interpolants generated for these proofs share more logic compared to those in Figure 1.

Many modern synthesis tools use And-Inverter Graphs (AIGs) as their underlying data structure. With AIGs, equivalence between nodes can be asserted with SAT-based algorithms, combined with structural hashing [14]. When two nodes are proved to be equivalent, one node can be substituted in place of the other, and any dangling logic can be removed from the netlist.

A related but more difficult problem is determining whether or not certain nodes can be expressed as a function of other nodes (sometimes called *substitution* or *resubstitution* [4]). For this task, the application of SAT-based methods is not so straightforward. In contrast, it can be easily determined if nodes in a resolution proof can be resolved from a subset of other nodes. Accordingly, Craig Interpolation provides a new approach to performing substitution in logical synthesis.

In this paper, we propose an algorithm for merging resolution proofs and then restructuring them. The structure in the resulting interpolant is then less sparse. This interpolant can be further minimized with traditional minimization algorithms. Trials on benchmarks suggest that there is significant potential for restructuring in the proofs of target function sets encountered in practice.

This research has been funded by NSF CAREER award #0845650.

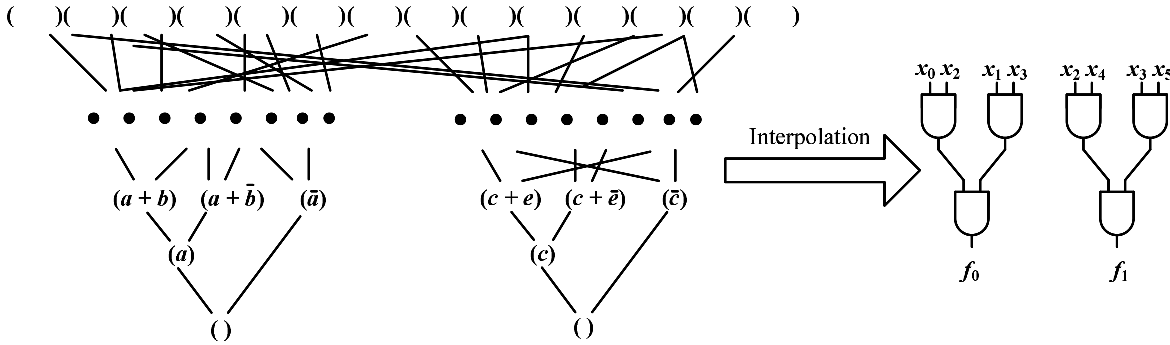

Fig. 1. A conceptual example of two resolution proofs with very few shared clauses. The resulting interpolants do not contain any shared logic.

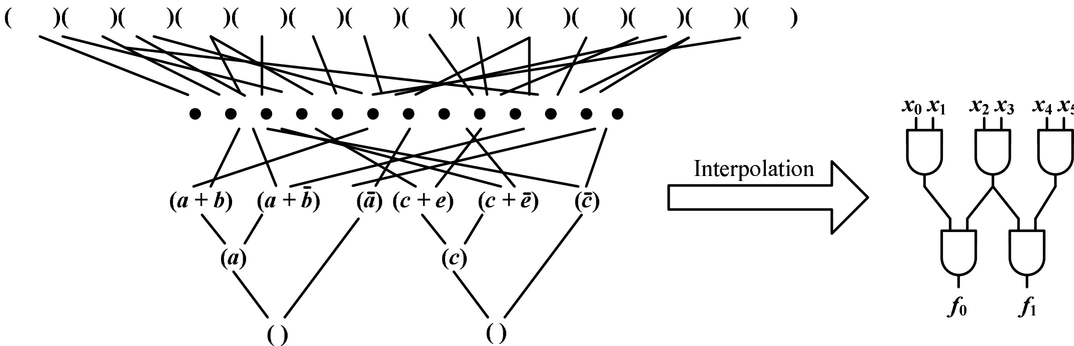

Fig. 2. An conceptual example of two resolution proofs that share many of the same clauses. The resulting interpolants share some of same logic.

II. Related Work and Context

In [9], a method for generating functional dependencies based on Craig Interpolation was proposed. This method was shown to scale much better with circuit size than previous methods based on binary decision diagrams (BDDs) [8]. While the process of finding and generating the dependencies with this method is efficient, in many cases, the resulting logic is poor. This is because the interpolant that implements the dependency function is often large and redundant. The structure of the interpolant is heavily dependent on the structure of the proof of unsatisfiability generated by the SAT solver. Generally, solvers strive for speed without regard to the proof structure. Some methods for reducing the size of interpolants in specific application domains have been proposed [5]. Methods for adjusting and reducing the size of resolution proofs have been discussed [2], [7]. Unfortunately, it is difficult to predict or measure how these algorithms affect the structure of the interpolants that are generated from the proofs.

In [1], an algorithm for augmenting a resolution proof with the goal of reducing the size of the proof's interpolant was proposed. The authors showed that changing the order in which clauses were resolved could greatly reduce the size of the corresponding interpolant. In this paper, we expand on this idea by showing how multiple resolution proofs can be merged into a single, monolithic proof. We show how this monolithic resolution proof can be restructured in order to increase the amount of shared logic in the resulting interpolant.

III. Background and Definitions

A Boolean formula maps an assignment of Boolean variables to a Boolean value ($\mathbf{B^m} \rightarrow \mathbf{B}$). We use the convention that addition denotes an OR operation, multiplication or a "$\wedge$" denotes the AND operation, a "$\rightarrow$" denotes implication, and an over-bar (e.g., $\bar{x}$) denotes negation. An occurrence of a Boolean variable in a Boolean formula, either negated or non-negated, is referred to as a *literal*. A disjunction of literals is referred to as a *clause*. A Boolean formula is in conjunctive normal form (CNF) if it is represented by a conjunction of clauses. Boolean Satisfiability (SAT) is the problem of determining whether or not a CNF formula can evaluate to *true* for some assignment of its variables. If there is some variable assignment that causes the formula to evaluate to *true*, then the formula is said to be *satisfiable*. If there is no variable assignment that causes the formula to be *true*, then the formula is said to be *unsatisfiable*. We will sometimes use the term *SAT instance* when referring to a CNF formula whose satisfiability we are trying to solve.

Given the conjunction of two clauses that share a literal that is negated in one but not the other, a third clause that is a disjunction of their remaining literals is implied. This identity is known as Boolean *resolution*. The common literal that is negated in one clause but not the other is called the *pivot variable* and the resulting clause is called the *resolvent*. Consider the identity

$$(z_0 + x_1 + \cdots + x_n)(\bar{z_0} + y_1 + \cdots + y_m) \rightarrow$$
$$(x_1 + \cdots + x_n + y_1 + \cdots + y_m)$$

Here the pivot variable is z_0. A set of clauses can be proved to be unsatisfiable through a series of resolutions that lead to an empty clause. This results in a directed acyclic graph (DAG): the *roots* are the original clauses, the *intermediate* nodes are clauses proved by resolution, and the single *leaf* is the empty clause. This structure is called a *resolution proof*. We will sometimes use the words "node" and "clause" interchangeably when we are discussing resolution proofs.

When two clauses c_1 and c_2 resolve a clause c_3, c_1 and c_2 are said to be the *parents* of c_3; c_3 is said to be a *child* of c_1 and c_2. Clauses that were used to resolve c_1 or c_2 are said to be *ancestors* of c_3. When we say that a node is towards the *beginning* of a proof, we are declaring that there were few resolutions steps taken from the leaves of the proof to reach this node. When we say that a node is towards the *end* of a proof, we are declaring that there are few resolution steps that need to be taken to reach the empty clause from this node.

Given an unsatisfiable instance of SAT and a bi-partition of its clauses, set $\boldsymbol{A}$ and set $\boldsymbol{B}$, Craig's Interpolation theorem states that there exists an intermediate formula $\boldsymbol{I}$, called an *interpolant*, such that $\boldsymbol{A} \rightarrow \boldsymbol{I}$ and $\boldsymbol{I} \rightarrow \overline{\boldsymbol{B}}$. A variable in the SAT instance is said to be a *global variable* if it is present in both clause sets $\boldsymbol{A}$ and $\boldsymbol{B}$. Likewise, a variable is said to be *local* to a clause partition if it is only present in that clause partition. An interpolant only contains variables that are global to $\boldsymbol{A}$ and $\boldsymbol{B}$. We say that a set of clauses is *satisfied* for some assignment of the set's variables if every clause in the set evaluates to true.

The algorithm in Figure 3, presented in [10], is a procedure for generating a circuit that implements an interpolant from a resolution proof and a clause partition. It was adapted from a procedure presented in [13] to find the Boolean value for an interpolant given a variable assignment.

```
p(c):
     if c is a leaf clause
        if c is in A
           return g(c)
        else
           return 1
     else let v be the pivot variable
        if v is local to A
           return p(c1) + p(c2)
        else
           return (p(c1))(p(c2))
```

Fig. 3. The algorithm proposed in [10] to produce a circuit that implements an interpolant of a given clause partition, via a proof of unsatisfiability.

The procedure g(c) takes a clause c as input and returns clause c with only its global literals present. Let c_1 and c_2 refer to c's parent clauses. Procedure p(c) is defined in Figure 3. Calling p(c) on the empty clause of a resolution proof will return a DAG whose nodes represent Boolean functions. In this DAG, the node with no fanout, corresponding to the empty clause in the resolution proof, computes a Boolean function in terms of the global variables of $\boldsymbol{A}$ and $\boldsymbol{B}$. This Boolean function is an interpolant of the given clause partition. When we refer to the *size* of an interpolant, we mean the number of gates that are needed to represent it. It should be clear that the size of the interpolant is bounded by the number of nodes in the resolution proof.

A. Functional Dependencies with Craig Interpolation

A method for synthesizing functional dependencies based on *Craig interpolation* was proposed in [9]. The formulation of our algorithm relies on this construction so we provide a brief review of it here.

The method constructs a miter, as shown Figure 4. Here f_0 is the *target function*. The satisfiability of the primary output of this circuit indicates whether or not there exists a dependency function $h(f_1,f_2,f_3)$ that can be used to represent f_0 for some network. Here f_0 *Left* and f_0 *Right* are two copies of the same network. The primary inputs $x_0, x_1, \ldots, x_n$ (referred to as X) are the primary inputs to f_0 *Left*. The primary inputs x_0*, x_1*, $\ldots$, x_n* (referred to as $X*$) are the primary inputs to f_0 *Right*; these are distinct sets of variables, but in direct correspondence with one another: $f_i(X)$ is equivalent to $f_i*(X*)$ where the assignment of X is equal to the assignment of $X*$.

If the primary output of this circuit is satisfied, then this indicates that f_0 evaluates to a different value from f_0* while functions f_1, f_2, and f_3 evaluate to the same values of f_1*, f_2*, f_3*, respectively, on each side of the circuit for some assignment of X and $X*$. Clearly this indicates that the ON set $f_0(f_1,f_2,f_3)^1$ is not disjoint from the OFF set $f_0(f_1,f_2,f_3)^0$. Accordingly, there is no function $h(f_1,f_2,f_3)$ that is equivalent to $f_0(X)$ (or to $f_0*(X*)$).

If the primary output of the circuit is unsatisfiable for all assignments of X and $X*$, this indicates that either f_0 (or f_0*) is a constant 1 or 0, or that the ON set $f_0(f_1,f_2,f_3)^1$ is disjoint from the OFF set $f_0(f_1,f_2,f_3)^0$. This indicates that there is some function $h(f_1,f_2,f_3)$ that is functionally equivalent to $f_0(X)$.

If the clauses representing the logic surrounded by the box in Figure 4 are partitioned into a set $\boldsymbol{A}$ and the rest of the clauses are partitioned into a set $\boldsymbol{B}$ then the resulting interpolant of this clause partition will be a valid implementation of the function $f_0(f_1,f_2,f_3)$.

IV. General Method

Consider using the approach described in the previous section for generating functional dependencies in the case where many of functions contain the same support variables. Let functions $g_0, g_1, \ldots, g_n$ be the target functions and functions $x_0, x_1, \ldots, x_n$ be the variables in the potential support sets for these functions. Then the SAT instance to verify the existence of the ith functional dependency takes the following form:

$$\begin{array}{l} A = (g_i) \wedge (CNF_{\text{left}}) \\ B = (x_0 \equiv x_0^*) \wedge (x_1 \equiv x_1^*) \wedge \ldots \\ \wedge (x_n \equiv x_n^*) \wedge (CNF_{\text{right}}) \wedge (\bar{g}_i{}^*) \end{array} \quad (1)$$

Eq. 1: A SAT instance that checks the existence of target function g_i with support set $x_0, x_1, \ldots, x_n$.

Here CNF_{left} and CNF_{right} represent the clauses for the circuit elements in the left and right halves of the circuit, respectively, as shown in Figure 4. The common variables between sets $\boldsymbol{A}$ and $\boldsymbol{B}$ $(x_0, x_1, \ldots, x_n)$ remain the same

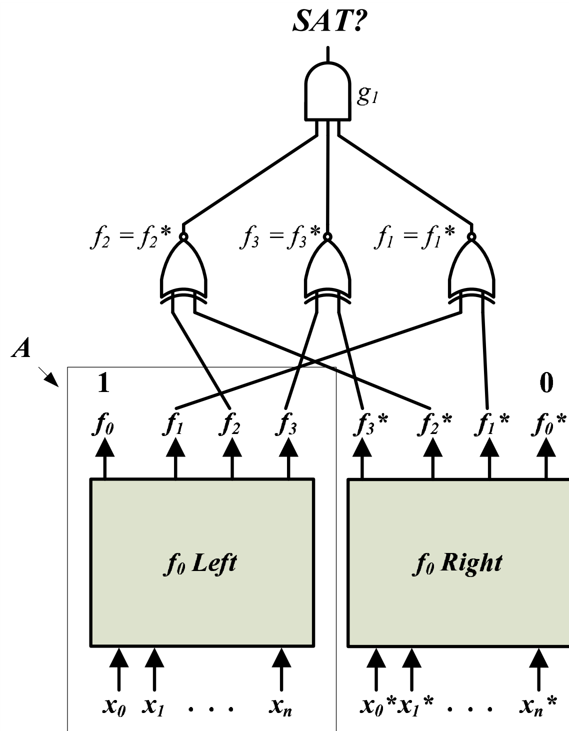

Fig. 4. A miter that checks to see if f_0 can be specified in terms of f_1, f_2, and f_3.

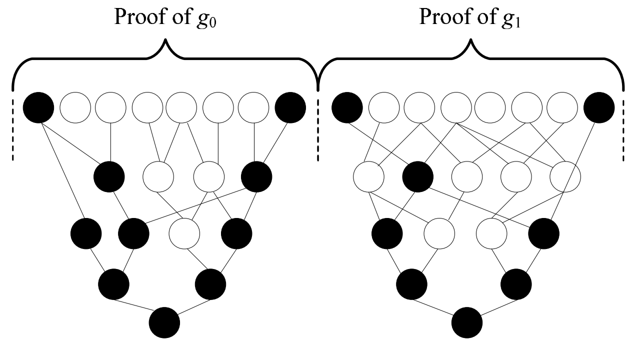

Fig. 5. Two resolution proofs without any shared nodes.

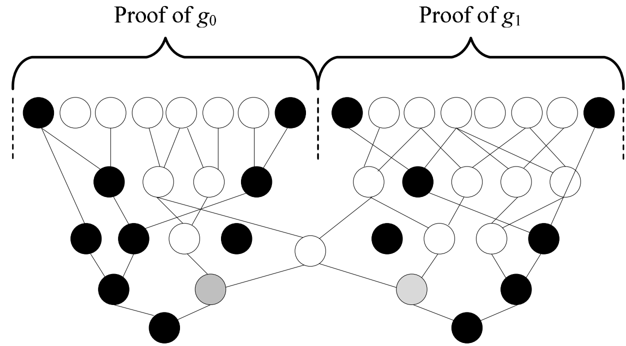

Fig. 6. Two resolution proofs with a shared node. An additional white node is added to the proof, but two black nodes can then be removed. The gray nodes are nodes that were black but can now be colored white.

for each SAT instance. Also, the only clauses that differ among each of the individual SAT instances are the (g_i) and $(\bar{g_i}*)$ terms. We refer to these terms as the "on" and "off" *assertion clauses*, respectively. Such large similarity between SAT instances can be leveraged to create similarities in the proofs of unsatisfiability. Structural similarities in the resolution proofs can then lead to structural similarities in the interpolants. Using these properties, we propose the following general method for creating a circuit structure that contains shared logic.

1) For each primary output, create a SAT instance verifying that the primary output can be expressed in terms of the circuit's primary inputs (Equation 1).
2) Generate a proof of unsatisfiability for each SAT instance created in the previous step.
3) For each node in each proof, color the node black if it has an assertion clause as an ancestor; otherwise color the node white.
4) Check to see if any black node in any proof can be resolved from any set of white nodes. If it can, color the node white.
5) Restructure the proofs so that the black nodes that were re-colored in step 4 are only resolved from white nodes.
6) Generate the interpolant for each proof.

Step 4 of the algorithm is illustrated graphically in Figures 5 and 6. Figure 5 contains two proofs whose interpolants are an implementation of functions g_0 and g_1, respectively. Figure 6 shows that two black nodes can be resolved from only white nodes present in both of the proofs.

V. CORRECTNESS

In this section, we argue the correctness of our method and elucidate it with examples. First we discuss our mechanism for deciding whether or not a node in a resolution proof can be resolved from other nodes. This mechanism was discussed by Gershman in [7] and was used to reduce the size of interpolants in [1].

Proposition 1
Let c be some clause and W be some set of clauses. Then c can be resolved from W if $W \wedge \bar{c}$ is unsatisfiable

Proof: The statement c can be resolved from W iff $W \rightarrow c$ is a tautology. Therefore, if there is no assignment of the variables present in W and c such that every clause in W evaluates to *true* and c evaluates to *false*, then c can be resolved from the clauses of W. ■

To perform Step 4 of our algorithm, we can simply repeatedly solve the SAT instance $W \wedge \bar{c}$, where W is the set of all the white clauses present in all of the resolution proofs and c is a black clause whose color we wish to change. If $W \wedge \bar{c}$ is unsatisfiable, a resolution proof of unsatisfiability will be returned by the SAT solver. This proof can then be modified by a procedure known as *bubble transformation*, described in [7], to show how c can be resolved by the clauses of W. The bubble transformation can then used to implement Step 5 of our algorithm.

Example 1
Figure 7 shows a portion of a resolution proof. Let clauses $(a +$

$b)$, $(a+b+\bar{c})$, and $(a+b+c)$ be colored black and the remaining clauses be colored white. In Step 3 of the algorithm we want to determine if clause $(a+b)$ can be colored white. Clause $(a+b)$ can be resolved from clauses $(a+\bar{e}+\bar{d})$, $(a+b+d)$ and $(a+b+\bar{d}+e)$ iff $(a+\bar{e}+\bar{d})(a+b+d)(a+b+\bar{d}+e) \rightarrow (a+b)$ is a tautology. Solving the SAT instance: $(a+\bar{e}+\bar{d})(a+b+d)(a+b+\bar{d}+e)(\bar{a})(\bar{b})$ verifies that $(a+b)$ can indeed be resolved from only white clauses.*

The bubble transformation method described in [7] can then be used to show the resolution steps needed to resolve $(a+b)$ from the white nodes. These resolution steps can then be used to restructure the proof as shown in Figure 8.

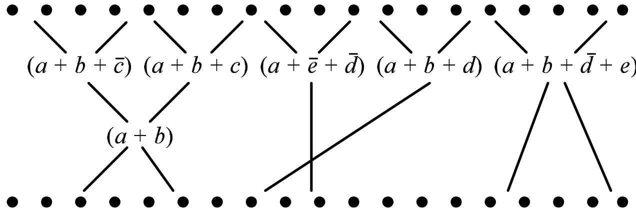

Fig. 7. A portion of a resolution proof

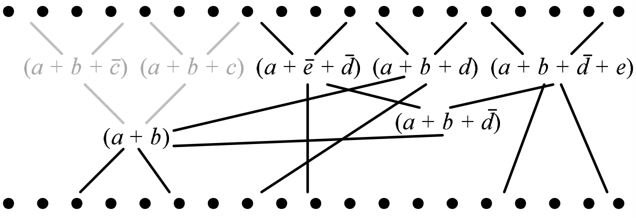

Fig. 8. A portion of a resolution proof that has been restructured. Clause $(a+b)$ can be resolved from clauses $(a+\bar{e}+\bar{d})$, $(a+b+d)$ and $(a+b+\bar{d}+e)$. Restructuring the proof this way adds one clause and removes two others.

After the proofs are restructured, the interpolants are generated by calling the recursive procedure described in Figure 3 on each of the empty clauses present in the resolution proofs. We assert that these interpolants are *valid* in the sense that they still fulfill the same properties as interpolants generated from the non-modified resolution proofs.

Proposition 2
The interpolants generated from the restructured proofs are still valid.

Proof: As described in the previous section, the only root clauses that differ among each resolution proof are the "on" and "off" assertion clauses. Therefore all the white root clauses are shared between the proofs. Step 5 of our algorithm only restructures the proof such that a previously black node is resolved from only white clauses. Therefore any white node present in the restructured proof is resolved from only root clauses that are common between each of the original proofs. Because the set of root clauses remains the same between each proof, the clauses present in $\boldsymbol{A}$ and $\boldsymbol{B}$ remain the same. Also, for each resolution proof, the common variables of $\boldsymbol{A}$ and $\boldsymbol{B}$ remain the same; therefore the interpolants generated in Step 6 are valid. ■

VI. Implementation and Results

We implemented steps 1 through 4 of our method in Berkeley ABC [12]. To test the potential savings of our algorithm, we performed trials on benchmarks in the IWLS benchmark collection [3]. Trials were run on a AMD Phenom™ II X6 1090T machine with 3GB of RAM running 32-bit Linux. Only one core was utilized. In Table I, we present numbers on how many of the black nodes can be re-expressed in terms of only white nodes present in the proofs. These numbers allow us to gauge the potential for restructuring resolution proofs. The column "Orig. Num. White" lists the number of white nodes originally present in the proofs. If a white node has an identical set of literals as another white node, these nodes are merged and counted as only one white node. The column "Orig. Num. Black" lists the number of black nodes originally present in the proofs. The column "Num. Checked" lists the number of black nodes that we checked to see if they could be considered white. The column "Num. Fixable" lists the number of nodes that could be colored white out of the number of black nodes that we checked. The column "Percent Fixable" lists the percentage of the ratios of the "Num. Fixable" to "Num. Checked" columns. The "Time" column lists the time spent checking to see if black nodes could be colored white.

The time spent on each benchmark was limited to 200 seconds. In this experiment, all white nodes present in the proofs were considered when checking to see if a black node could be colored white (Step 4 of the algorithm). The more white nodes considered, the larger the SAT instance that needs to be solved in Step 4. Reducing this number to be a subset of the white nodes present in the proof will reduce the runtime; however this may also reduce the number of black nodes that can be colored white. This is a tradeoff that we plan to investigate in the future.

For most of the benchmarks, the percentage of black nodes that could be colored white was around 20–30%. The `table3` and `table5` benchmarks were two exceptions to this trend. Since trials on both of these benchmarks reached the timeout, only a subset of nodes were checked. The percentage was calculated as the number of fixable nodes divided by the number of nodes that were checked. It is likely that if a longer timeout had been used, then these percentages would be similar to the other benchmarks.

Perhaps the most salient result is that our algorithm scales well with the number of nodes in the resolution proof. Note that, unlike previous implementations, we are not applying interpolation to individual functions; rather we are applying it to each circuit in its entirety. For cases where we reach the computational limit, runtimes could be improved by applying the algorithm to smaller windows of logic within the circuit.

VII. Conclusions

Given a resolution proof, determining whether or not a given node in the proof can be resolved from a set of other nodes is an easy task. Given an AIG, determining whether a given node in the graph can be expressed in terms of other nodes is difficult task. Accordingly, algorithms like AIG Rewriting and SAT-Sweeping can only make incremental improvements to small windows within an AIG [11], [14]. In contrast, the

Number of Black Nodes That Can Be Colored White						
Benchmark	Orig. Num. White	Orig. Num. Black	Num. Checked	Num. Fixable	Percent Fixable	Time (s)
dk15	1743	581	581	175	30.12	0.04
5xp1	3203	1636	1636	275	16.81	0.18
sse	3848	2650	2650	563	21.25	0.28
ex6	4055	2731	2731	588	21.53	0.29
s641	6002	5148	5148	2269	44.08	0.46
s510	7851	5092	5092	1155	22.68	0.74
s832	15359	14826	14826	3358	22.65	3.67
planet	40516	43387	43387	10640	24.52	26.39
styr	44079	54128	54128	16578	30.63	33.88
s953	49642	46239	46239	12252	26.50	31.99
bcd	96385	109167	103514	34349	33.18	200.00
table5	137607	288461	69070	27848	40.32	200.00
table3	177410	283066	47279	24454	51.72	200.00

TABLE I
RESULTS OF STEPS 1 THROUGH 4 OF OUR ALGORITHM APPLIED TO A SET OF BENCHMARKS. THE TIMEOUT WAS SET TO 200 SECONDS.

properties of resolution proofs allow for large transformations of the initial structure. The results presented in this paper suggest that there is significant potential for clause sharing among resolution proofs for multiple target functions. Given an abundance of shared clauses, we expect interpolants with significant structural similarities. Even if these structures lead to logic that suboptimal, it will be much less *sparse* than traditional AIG representations; accordingly, such logic might be a promising starting point for the application of traditional logic synthesis.

REFERENCES

[1] J. Backes and M. D. Riedel. Reduction of interpolants for logic synthesis. In *International Conference on Computer-Aided Design*, 2010.

[2] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, pages 114–128, 2009.

[3] Benchmarks from the 2005 International Workshop on Logic Synthesis available at http://iwls.org/iwls2005/benchmarks.html.

[4] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990.

[5] M. L. Case, A. Mishchenko, R. K. Brayton, J. Baumgartner, and H. Mony. Invariant-strengthened elimination of dependent state elements. In *Formal Methods in Computer-Aided Design*, pages 9–17, 2008.

[6] W. Craig. Linear Reasoning: A New Form of the Herbrand-Gentzen Theorem. *Symbolic Logic*, 22(3):250–268, 1957.

[7] R. Gershman, M. Koifman, and O. Strichman. An approach for extracting a small unsatisfiable core. *Formal Methods in System Design*, 33(1-3):1–27, 2008.

[8] Jie hong R. Jiang and Robert K. Brayton. Functional dependency for verification reduction. In *in Proc. CAV*, pages 268–280, 2004.

[9] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *International Conference on Computer-Aided Design*, pages 227–233, 2007.

[10] K. L. McMillan. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13, 2003.

[11] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Design Automation Conference*, pages 532–536, 2006.

[12] A. Mishchenko et al. ABC: A system for sequential synthesis and verification, 2007.

[13] P. Pudlak. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[14] Qi Zhu, Nathan Kitchen, Andreas Kuehlmann, and Alberto Sangiovanni-Vincentelli. Sat sweeping with local observability don't-cares. In *Design Automation Conference*, pages 229–234, 2006.