DAG-Aware Circuit Compression For Formal Verification

Per Bjesse
Synopsys Inc.

Arne Borälv
Prover Technology Inc.

Abstract— **The choice of representation for circuits and boolean formulae in a formal verification tool is important for two reasons. First of all, representation compactness is necessary in order to keep the memory consumption low. This is witnessed by the importance of maximum processable design size for equivalence checkers. Second, many formal verification algorithms are sensitive to redundancies in the design that is processed. To address these concerns, three different auto-compressing representations for boolean circuit networks and formulas have been suggested in the literature. In this paper, we attempt to find a blend of features from these alternatives that will allow us to remove as much redundancy as possible while not sacrificing runtime. By studying how the network representation size varies when we change parameters, we show that the use of only one operator node is suboptimal, and demonstrate that the most powerful of the proposed reduction rules, two-level minimization, actually can be harmful. We correct the bad behavior of two-level optimization by devising a simple linear simplification algorithm that can remove tens of thousands of nodes on examples where all obvious redundancies already have been removed. The combination of our compactor with the simplest representation outperforms all of the alternatives we have studied, with a theoretical runtime bound that is at least as good as the three studied representations.**

I. Introduction

Binary Decision Diagrams (BDDs) [1] have become a very important representation for boolean functions in the verification domain. However, since this representation is exponential for many functions of interest, it is not a good general purpose representation for a circuit under analysis. In many tools, the netlist itself therefore forms the key representation for a circuit, and when formal models for satisfiability checking or model checking needs to be constructed, this model is massaged and translated into whichever form is necessary.

The netlist representation, however, is non-canonical in the sense that it allows many implementations of a particular boolean network, and many of the netlists generated for formal verification purposes are very redundant [2]. Representation redundancy is bad for two reasons. First of all, a redundant representation will increase the memory foot-print of the formal verification tool. This may cause problems, considering the fact that we not only will rely on our datastructure to represent the system under analysis (which might mean tens of millions of network nodes) but that we also may generate many times more formula operations if we apply SAT-based model checking procedures like McMillans's Interpolation algorithm [3].

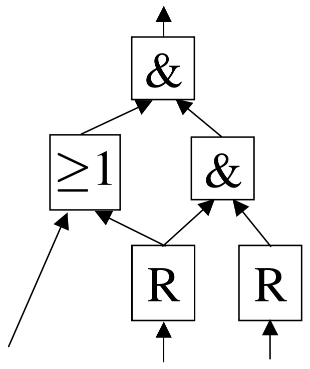

Fig. 1. Retiming example.

Second, the quality of results of many formal verification algorithms are sensitive to redundancies in the representation of the model under analysis. For example, consider the network in Figure 1, where the boxes containing R's are registers, and the inputs to the network are free inputs. Assume that we want to retime this network in the Leiserson-Saxe sense [4]. As it stands the number of registers can not be improved as the OR gate does not have registers on both inputs. However, if we study the combinational logic network, our example in Section II-B shows that this network is redundant, and could be represented using a single AND operator fed by two registers. This implementation could hence be retimed to a network with a single register at the output. This example is not an anomaly. In practice, removing combinational redundancies is crucial to achieving good retiming results [5].

The sensitivity to redundancies is unfortunately not unique to retiming: Consider building a BDD representing a netlist containing a multiplier whose outputs are thrown away due to environment constraints for the current verification run. The sensitivity also extends to CNF-based approaches to satisfiability checking where the removal of redundancies prior to the invocation of a SAT checker can speed up problems with orders of magnitude [6].

Moreover, the redundancy sensitivity problem is not abating: Many of the state-of-the-art approaches to model checking are sensitive to redundancies in the circuit. One example would be the heuristics used to do localization reduction [7] which can be fooled by redundant logic and registers. Moreover, as the underlying SAT solvers are sensitive to redundancies, algorithms that analyze proofs from SAT-solvers will also be sensitive—consider the use of proof cores to do abstraction refinement [8], compute reachability images [3] and to do SAT-based predicate abstraction [9].

While it is important to have a representation for boolean circuit network that is as free from redundancies as possible, it is also important to not make representation construction a large part of the runtime of a formal verification tool—some circuits are free from redundancies and it would be unacceptable to slow down the verification of these circuits with hours of needless attempts at compaction. The representation that is used therefore needs to be built with little overhead in terms of runtime.

A popular approach to the solution of the low-overhead compact representation problem is *auto-compacting DAG circuit* representations. Examples of such representations include Boolean Expression Diagrams (BEDs) [10], And-Inverter graphs [2], and Reduced Boolean Circuits (RBCs) [11]. These three representation are all variations on the theme of a data structure that uses hashing to share all structurally isomorphic parts of a circuit, and that applies a set of fast rewriting rules transparently during network construction.

Since BEDS, And-Inverter graphs, and RBCS all have been devised to solve the representation problem with very small time overhead, they are all in principle good choices for representations. However, since the differences between the three approaches mainly consists of variations on what boolean operators are allowed as nodes in the representation, and what minimization operations are performed, the question is where the sweet spot is in terms of features for the kind of circuit networks found in industrial applications.

In this paper, we introduce the three different representations and study the impact of varying the different parameters when building the representation for seven industrial examples. Led by the experimental results, we diagnose a problem with the most powerful reduction rule used, two-level recursive rewriting, and propose a new compacting algorithm that remedies this problem.

Our contribution has three parts. First of all, we introduce and contrast the different approaches to auto-compacting circuit representations that have appeared in the literature. Second, we present the first experimental study of the impact of different choices of operations and rewriting rules in boolean circuit compaction for verification. Third, we present a new compaction algorithm that is powerful enough to be able to remove tens of thousands of nodes in negligible time on large designs even after all the standard minimization tricks utilized by BEDs, RBCs and And-Inverter graphs have been performed.

The paper is organized as follows. In Section II, we present an overview of the three different representations. In Section III, we study the impact of the number of operator nodes in the representation and present experimental evidence that demonstrate the two-level rewriting rule that is a fundamental part of the BED and And-Inverter representations actually can be harmful. We then present an analysis in Section IV of why this happens, and propose an improved two-level minimization algorithm that remedies the problem. In Section V we present the related work, and in Section VI we conclude.

II. Overview of Boolean Circuit Representations

In this section we introduce the three different function representations that we will study, and demonstrate how they are all variations on a common concept. We use RBCS as our base line representation, as it conceptually is the simplest representation.

A. *Reduced Boolean Circuits (*RBCS*)*

The easiest way to understand Reduced Boolean Circuits (RBCS) is as a restricted class of directed acyclic graphs constructed from nodes and edges. RBCS have two different sorts of nodes: Operator nodes and variable nodes. An operator node has two children and an operator attribute ($\wedge$, $\vee$, or $\leftrightarrow$), whereas a variable node has no children and a variable name attribute. There is one special variable node, $\top$, which is taken to represent the true formula. Nodes are connected with edges that have a negation attribute. Every edge in a RBC correspond to some formula through the obvious semantics of operators and negation markers—the top edge of the RBC in Figure 2 represents $(a \vee \neg b) \wedge (b \leftrightarrow c)$.

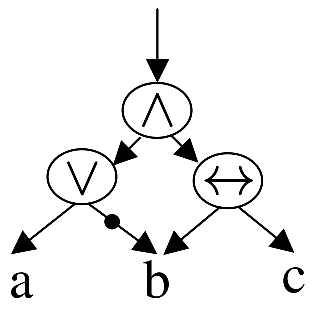

Fig. 2. A simple DAGed circuit.

RBCs are constructed in such a way that they fulfill the following obligations:

1) All common subformulas are shared. This is achieved by hashing during construction.
2) Operator nodes do not have identical children.
3) Operator nodes do not have constant children.
4) $\leftrightarrow$ nodes do not have negated children.
5) The left child is smaller than the right child of an operator node according to some total order $<$ on nodes.

The algorithms for constructing a new RBC from two operand RBCS, MKRBC, do all these minimizations in constant time (assuming constant time hashing), so construction is very fast. Although these transformations do not seem that strong, large savings can be achieved compared to an unoptimized netlist representation. For example, in the presence of environmental constraints, constant propagation alone may remove large parts of a circuit.

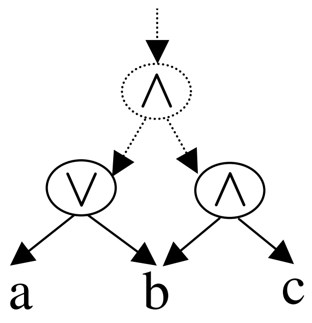

Fig. 3. BED two-level reduction example.

B. Boolean Expression Diagrams (BEDs)

BEDs are datastructures similar to RBCs, but with some variations and strengthenings. First of all, an operator node in a BED is either a standard operator node or an BDD ITE-node (see [1]), and any two-input boolean operator is a legal BED operator. Second, there are no variable vertices; variable vertices are implemented by ITE nodes with constant children. Third, there are no negation markers, as negation is a (redundant) two-operator function.

BEDs use reduction rules corresponding to rule 1–3 for RBCs, but in addition the following optimizations are performed:

- There are no negation operators below a non-negation, non-ITE operator. This is possible to achieve since every binary operator is allowed in a BED.
- When an operator node with two children operator nodes is to be created, this two-level window is rewritten to a canonical two-level window with as few nodes as possible. This rule, which we will refer to as the *two-level minimization rule*, will play an important part in the remainder of this paper.

The presence of ITE nodes in BEDs allows a BED to be incrementally translated to a BDD: It is not hard to define a transformation that bubbles up a ITE vertex to the top of a BED. This transformation can be performed for all occurrences of ITE nodes in one variable, and by doing this for all variables in order, we will be left with a BDD. In this paper, we will not make use of this incremental translation facility and we will hence not concern ourselves with ITE nodes further.

As in the case of RBCs, the BED reductions are applied transparently whenever a new BED is created from two operands. The two-level minimization rule is implemented by tabulating all boolean functions of three nodes or less, and choosing a minimal number node implementation for each such function. Once a new node is going to be created from two operands, the top two levels of the new function is inspected, and the tabulated implementation is used as a template to generate the resulting node.

As an example of an application of two-level minimization, consider the situation in Figure 3, where we are going to construct the BED MKBED(AND, n_1, n_2) where n_1 is a BED containing a top disjunction operator, and n_2 is a BED with a top conjunction operator. Table lookup on the top two-levels shows that this window can be reimplemented as MKBED(AND, b, c), which is evaluated and returned to the user.

Note that in contrast to the case of RBCs, the creation of a BED node may take more than constant time: The previous example shows that a single call to MKBED may spark one or more additional recursive calls to MKBED. Each one of these calls may in turn trigger a number of new calls, and so on.[1]

C. And-Inverter graphs

And-Inverter graphs can conceptually be seen as a graph structure that is similar to RBCs in that it contains operator nodes, variable nodes and a mechanism for representing terminals. Moreover, just like RBCs, And-Inverter graphs represent negations with markers on edges. A crucial difference, however, is that And-Inverter graphs only allow the AND operator. Compared to BEDs, And-Inverter graphs have no ITE nodes. In terms of reductions, And-Inverter graphs combines rules 1–3 from Section II-A with the two-level minimization rule from Section II-B. Since two-level minimization is done before every node creation, a single operator creation is not guaranteed to run in constant time for the same reason as in the case of BEDs.

III. Comparing Representations

In order to find an optimal representation for boolean functions and networks we want to meet two important criteria: On one hand, we want the representation to be as *compact* as possible so that it is free from redundancies. On the other, we want the construction of the representation of a boolean network to be *fast* so that our model construction does not slow down the overall algorithms.

The three representations that we presented in Section II, RBCs, BEDs, and And-Inverter graphs, all seem like good candidates and they have a lot in common. However, there are some crucial differences between them.

For one, And-Inverter graphs only allow one operator, whereas RBCs use three and BEDs allows the use of every two-input logic operator. In the presence of negation markers on edges, the use of more binary operators than the set ($\wedge$, $\vee$, or $\leftrightarrow$) used in RBCs does not improve the strength of a representation as every binary operator has a one node RBC representation (with possible negation markers on edges). However, [2] states that the restriction to one operator in And-Inverter graphs is motivated by the assumption that the use of a single type of operator node will allow more things to get hashed to each other. Clearly, there is a penalty for doing this—for example, binary XOR operators now take three nodes to represent whereas it would be represented as a single RBC and BED node. Will the these factors offset each other?

[1]In the original BED paper [10], two-level rewriting is not part of node creation, but it is done in all known BED implementations

Circuit	RBC	And op. only	And op. only 2-level rewr.	And op. only 2-level red. only	RBC 2-level red. only	RBC DAG-aware rewr.	RBC DAG-aware rewr.*
1	2724	+99	-46	-324	-892	-848	-910
2	2882	+40	+249	+24	-200	-30	-198
3	7678	+448	+1168	+448	0	-315	-580
4	7705	+433	+1277	+415	-28	-309	-616
5	8961	+163	+557	+14	-184	-699	-1471
6	77181	+81	+46761	+554	-507	-908	-3664
7	100709	+2872	+8656	+1779	-1156	-1890	-11401

TABLE I

CIRCUIT SIZES IN NODES FOR DIFFERENT BUILD STRATEGIES

Secondly, And-Inverter graphs and BEDS use two-level minimization to compress the representation while it is being built. Intuitively this seems like a really good idea, and [10], [2] maintain that this is a key reason for efficiency of And-Inverter graphs and BEDS.

The baseline for our investigations will be RBCS as (1) this is the simplest representation of the three alternatives as it does not use two-level tabulation, (2) unlike BEDS and And-Inverter graphs (for which the worst-case complexity of network construction is unknown) we know that we are guaranteed to be able to compute the RBC representation of a network in linear time, (3) it is the data structure that already is used to represent formulas in the NUSMV 2 and FIXIT model checkers.

In order to find a maximal strength representation, we will now investigate what impact a restriction in the number of operators will have, and what impact the use of two-level minimization will have. We hope the answers to these questions will lead us to some way to incorporate the best features of And-Inverter graphs and BEDS into the RBC data structure in such a way that we arrive at an optimum representation.

Our experimental setup is as follows. We have implemented RBCS as described in [11], and made some extensions to the datastructure so that we can set a flag that will restrict the set of generated nodes to contain AND operators only instead of the full RBC set. We have also constructed a two-level minimizer that can handle windows with arbitrary operators. We will study the result of applying this package to the construction of the representation of seven industrial designs ranging in size from a hundred registers to four thousand registers. Before these designs reach the formal modeling, they have been synthesized from Verilog, and gone through many optimization steps including restructuring of sequential logic by retiming and identification of equivalent registers.

We present our results in Table I. The entries in each columns shows the number of nodes generated by different strategies. We do not present the runtime, as the time for constructing the representation for all these problems and all strategies is less than a second.

The baseline for the experiments is RBCS in column 2—the size of the RBC representations of the logic range between a few thousand nodes and a hundred thousand nodes. The other columns show the difference in size relative to the RBC representation (a negative number means fewer nodes was used, and a positive number means that the representation increased in size compared to the RBC representation). Note that during RBC construction all constants are removed and all isomorphic networks are shared, so all subsequent savings are relative to a representation where all simple redundancies already have been removed.

A. Restricting the number of operators without two-level minimization

In our first experiment, we modify the number of operators that are allowed in the RBCS so that we only allow AND nodes. As can be seen in column 3 in Table I, this means that we incur a penalty ranging from about a hundred nodes up to three thousand nodes for the largest example. The reason for this is that some gates now in the worst case has to be represented using more than one node. That this would generate a size increase is not necessarily obvious, as this worst case might now happen very often due to sharing, and simplification. Moreover, if the thesis put forward in [2] holds, then the use of just one operator will allow more things to be shared, and this may offset the three node gates. However, for our examples the sharing gain is not enough to offset the cost of representing some binary gates with many operators even in the case of large networks.

B. Two-level minimization

The decision to use just one operator node in the representation may well incur less blowup if stronger minimization is used. To check whether this thesis holds for our test cases, we next measure the size of the representations when we just use AND operators but do two-level minimization during node creation (see column 4 in Table I).

This generates some interesting results. With the exception of the smallest circuit, the use of two-level minimization in the presence of just one operator make the results degrade severely even compared to the negative results generated by just allowing one operator. In the case of circuit 6, the two-level minimized circuit increases in size with almost 75% (a 50 000 operator node increase for a 80 000 node circuit).

The very strong negative results generated when turning on two-level rewriting seems extremely counter-intuitive as at a first glance the use of stronger minimization rules should make the representation smaller, rather than larger.

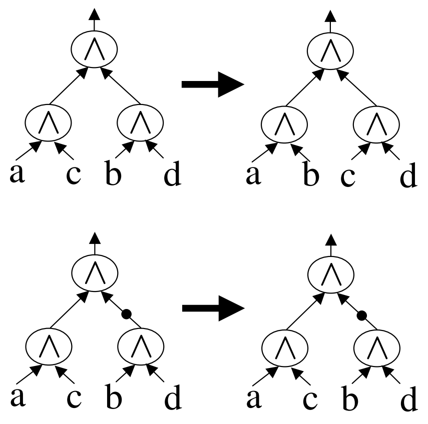

Fig. 4. A pair of two-level rewrite rules.

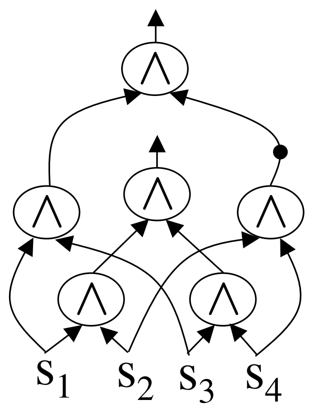

Fig. 6. A suboptimal rewrite result.

The heart of the problem is the following. Since all rewriting rules are applied as a node is constructed, there is no allowance for the effect of node sharing (nor can it be, as at creation time, the final number of references to a particular node is unknown). A rewrite that does not consider sharing can thus be globally destructive, even though it seems like a locally good thing to do. To see this, assume that we are using the two-level rewrite rules in Figure 4. The first rule rewrites the four input AND network so that the input networks occur in order. The second rule is an identity transformation.

As both the rules do not add new network nodes, it might seem like their use never should hurt. However, to see that this is not true in the presence of sharing, consider the rewriting of the simple circuit in Figure 5. When the left hand side

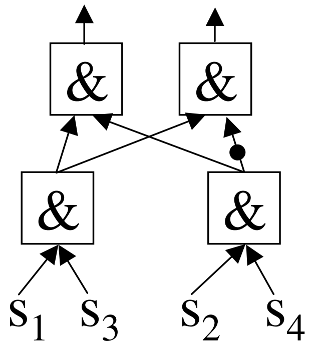

Fig. 5. An example circuit for non-optimal two-level rewriting.

logic function is built, the two-level rewriting now restructures the function so that the inputs are reordered according to the upper rule in Figure 4. However, in the case of the right hand function, the representation gets implemented as an isomorphic network to the original implementation. As a consequence, we end up with the representation in Figure 6, where 50% more nodes are needed compared to the representation size if we had not used two-level rewriting. As demonstrated in Table I, this kind of blowup is not only a theoretical possibility.

One response to the representation blowup in the example in Figure 5 is that this situation would not have occurred if only the rules would have ordered the inputs in a consistent way. However, there will always be situations where any given set of two-level rewrite rules will generate an increase in representation size, if sharing is not taken into account.

Another view would be that the problem is that we are permuting networks without necessarily removing nodes. However, rewriting that locally seems to save nodes still has the possibility to generate suboptimal results when sharing is being considered; the restriction to just perform a rewrite when a window gets reimplemented with fewer nodes for AND-only RBCs is almost always worse than the use of plain RBCs, and still sometimes worse than avoiding a two-level analysis for AND-only RBCs (see column 5 in Table I).

As a final experiment, we investigate the result of building full RBCs while doing two-level reductions only (see column 6 in Table I). With this build strategy, we have the first consistent improvement over plain RBCs with node savings with up to two thousand operators over the baseline. This is not bad considering that the runtime penalty for applying the two-level minimization is negligible. For our purposes, it hence seems clear that the combination of a richer set of operator nodes with reductions-only two-level rewriting works the best.

The reason that two-level minimization does fairly well in the context of full RBCs but can be harmful in the context of AND operators only is likely to be the following: A two-level window for a full RBC representation contains more logic to restructure than a two-level window when the only nodes are AND nodes. The potential savings from the rewritings are therefore more powerful, and this ends up offsetting costs incurred by destroyed sharing.

IV. Sharing-Aware Rewriting

In the previous section we have seen that we can gain thousands of nodes in representation size by combining RBCs with eager two-level rewriting. However, when we studied the impact of two-level based rewriting on RBCs that only contained AND operators, it was clear that two-level minimization actually may destroy a lot of the sharing that already is present in the graph and that positive results in terms of net operator gain comes at the expense of a potential for destructive rewrites. The following question therefore presents itself: How good would our results be for full RBCS if we could avoid performing the bad rewrites?

Moreover, destructive rewrites aside, network redundancy removal by combining RBC construction with standard two-level minimization has additional undesirable properties: For one, even if the rewriter could accurately analyze the impact of restructurings, restricting the choice to a single possible rewrite is likely to be suboptimal. No one reimplementation will always be the one that leads to the largest overall savings. Secondly, since only one possible reimplementation is possible for a particular two-level window and the restructuring is applied recursively, the standard minimization approach is *non-iterable* in the sense that once the full representation for a network has been built, a new rebuild will not result in additional savings.

If we were to construct an improved two-level minimizer, the following properties of the new algorithm would hence be desirable:

1) *Sharing-awareness.* The minimizer should take the impact of sharing into consideration when deciding whether to apply a rewrite or not.
2) *Versatility.* The minimizer should not be restrict itself to evaluating the consequences of a single possible reimplementation.
3) *Iterability.*

A. An improved minimization algorithm

Let us now focus on how to construct a minimizer that fulfills properties 1–3. In order to be sharing aware, we need to do delay our two-level rewriting until after we have constructed the representation for the whole network. In a first pass, our compactor will thus construct normal RBCS. Once that is done, we pass the two-level minimizer a list of the roots of the DAG that we want to compress.

Given the roots of the DAG, we can compute the *network reference count* for every node: If a node is pointed to by N nodes accessible from the roots of the formula DAG, then it is assigned a reference count of N. Nodes that are not present in the representation of the DAG have a reference count of zero.

We will make use of a hash function TWOLEVELHASH that maps an RBC window containing at most three operator nodes to a 16-bit number. Our rewrite candidates for a two-level window is stored in a precomputed table of size $2^{16} = 65536$ that is indexed by hash values. Every table entry contains a linked list of minimum-node reimplementations of the hashed two-level window (this means that if a given window can be implemented with two operator nodes, no three operator implementations are in the linked list). We will also use a rewriting function REWRITE that, given a node whose two-level window is to be transformed and a new two level implementation, returns the result of rearranging the window. For details of the construction of these supporting functions, see Section IV-B.

Algorithm 1 Compress the network rooted at $node$

```
TWOLEVELCOMPRESS($node$) :=
  if $node$ has been processed, and result was $res$ then
    return $res$
  else if $node$ is a constant or a variable then
    $res \leftarrow node$
  else
    $res_l \leftarrow$ TWOLEVELCOMPRESS(LEFTCHILD($node$))
    $res_r \leftarrow$ TWOLEVELCOMPRESS(RIGHTCHILD($node$))
    $currNode \leftarrow$ MKRBC(GETOP($node$), $res_l, res_r$)
    $currNode \leftarrow$ SETSIGN($currNode$, GETSIGN($node$))
    $hash \leftarrow$ TWOLEVELHASH($currNode$)
    $bestScore \leftarrow 0$
    for all $reImpl \in reimplTable[hash]$ do
      $score \leftarrow$ SCORE($reImpl, currNode, refcountTable$)
      $bestReImpl \leftarrow NULL$
      if $score \leq bestScore$ then
        $bestScore \leftarrow score$
        $bestReImpl \leftarrow reImpl$
      end if
    end for
    if $bestReImpl$ then
      $res$ = REWRITE($currNode, bestReImpl, refcountTable$)
    end if
  end if
  mark $node$ as processed with result $res$
  return $res$
```

Given the list of roots of the network, we apply the minimization algorithm presented as Algorithm 1 to each list entry. The algorithm traverses the nodes in the DAGed network in depth-first post-order (children of a node are processed before a node). For each node, every reimplementation of a given two-level window rooted at this node is scored using a function SCORE as follows. First the *node decrease score* is calculated by counting how many nodes in the current two-level window have a reference count of one. Then the *node increase score* is calculated by figuring out how many nodes that would result from the reimplementation currently either (1) are present in the current two-level implementation and have a reference count of one, or (2) are not in the current two-level implementation and have a reference count of zero. The total score for a reimplementation is the difference between the node increase score and the node decrease score. A positive (negative) score hence indicates how many new nodes would be created (destroyed) by the reimplementation. If there exists a reimplementation that does not increase the size of the network, then the window is rewritten and the affected reference counts are adjusted using the REWRITE function.

Our overall compaction algorithm runs in optimal time:

Theorem 1: Assuming constant time hashing, the complexity of one compression traversal over the whole network runs in time $O(N)$ for a network with N nodes.

Proof: To see this, consider that (1) reference counting is linear in the size of the network, (2) TWOLEVELCOMPRESS is called precisely once per node in the original network due to the function memoization, and (3) the number of possible reimplementations for a node is a bounded by a small fixed constant. ■

By construction, Algorithm 1 clearly has the desirable properties 1 and 2. However, is it iterable? Interestingly it is: Consider the fact that when we do a pass over the roots of the network we always rewrite if we do not gain in node count from the permutation. As a result, after one iteration of restructuring, it is very possible that we end up with a network that we can restructure again.

B. Implementation details

The tabulation of the rewrite rules could be done by generating code that perform the reimplementation of the two-level window in the same way as done in the described implementation of And-Inverter graphs in [2] and BEDS in [10]. However, this would be cumbersome as we need to return a list of alternatives rather than perform a set pattern of rewrites in each situation. We have therefore chosen the alternative route of representing a reimplementation of a two-level window as an RBC over at most four variables $v_1 \dots v_4$, and precomputing the table of rewrite list templates whenever the RBC package is initialized. This is simple to implement cleanly and ends up requiring negligible start-up time.

The hash function that maps the two-level window under a node to a variable slot number, TWOLEVELHASH, is implemented as follows: Assume that the variables $v_1 \dots v_4$ are ordered $v_1 < v_2 < v_3 < v_4$. If the two level window has n nodes in the support, assign variables $v_1 \dots v_n$ to the input nodes in such a way that the variable ordering respects the node ordering. The hash value for this two-level window is now the integer corresponding to the truth table row formed by evaluating this function for the 16 combinations of values for $v_1 \dots v_4$. Unlike the hash function for And-Inverter graphs [2] which relies on setting bits according to a number of properties of the window, this function has the benefit of being easy to implement and prove collision-free.

Given a reimplementation RBC template and a top node to process, the function REWRITE first identifies the correspondence between the nodes feeding the inputs of the current window and the template variables by studying the node ordering. Given the resulting node–variable correspondence, the window is next reimplemented by creating the appropriate RBC nodes, after which the network reference counts for the involved nodes are updated.

C. Experimental results

In column 7 of Table I we show the results of applying a single pass of our minimizer to the RBC representation of the circuit networks. As can be seen, this approach represents an improvement over all the other approaches for five out of the seven test cases with between $1.5x$ and $10x$ larger savings in nodes compared to the best alternative. However, the real difference compared to the other approaches becomes apparent when we iterate the minimization at most five times or until no big change occurs in size (see column 8 in Table I). When this is done, we win uniformly and increase the savings with a factor around $2x$ to $5x$ on all examples except the smallest.

The runtime for doing the repeated minimization is still negligible for all examples including the ones with on the order of a hundred thousand nodes. This is not a coincidence:

Theorem 2: The worst case time complexity for creating and minimizing a network using RBCS and our compression strategy is linear in the size of the network.

Proof: To see this consider that (1) the complexity for constructing the RBC representation of the original network is linear in the size of the network, (2) each minimization pass is linear in the size of the original RBC as shown by Theorem 1, (3) we do at most a constant number of passes over the network, and (4) the network never increases in size over the original RBC size during the minimization iterations. ■

Note that Theorem 2 shows that our approach to efficient representation construction and compaction is asymptotically at least as fast as the construction of And-Inverter graphs or BEDS since the worst case time complexity for constructing these representations for a network is at least linear and possibly worse (see Sections II-A and II-B).

V. RELATED WORK

The first of the three representations that we have discussed to appear in the literature was the BED datastructure [10]. Interestingly, the original aim of BEDS was not primarily to serve as a circuit representation, but as a way to generalize BDDs by allowing them to contain explicit operator nodes and support the incremental conversion of such a representation into canonical form. This side of BEDS have not been relevant to our presentation. BEDS have been used to do equivalence checking [12], SAT-based model checking [13], and fault tree analysis [14].

RBCS was developed independently from BEDS as a general representation for formulas and circuit networks in the SAT-based model checking framework FIXIT [11]. One way to look at RBCS is as a weaker variant of the BED datastructure that have done away with peculiarities necessary to allow incremental BDD construction. RBCS are used as a formula representation in the open source model checker NUSMV 2 [15], and has been used to implement a variety of analyses such as SAT-based approximate model checking [16] and Symbolic Trajectory Evaluation [17].

And-Inverter graphs was developed as a representation for circuits by Kuehlmann and coworkers [18], [2] and has been used as a key component in several approaches to combinational and sequential circuit verification. Noteworthy examples is the retimer of Baumgartner et.al. which interleaves logic restructuring with retiming [5], Ganai and coworkers's structural SAT solver [19], and Kuehlmann and coworkers's framework for boolean reasoning [20]. One reason for why the two-level blowup we are seeing may not have been as much of an issue in these frameworks compared to the results that we have shown may be that And-Inverter graphs primarily seem to have been applied to equivalence checking problems and the processing of high-performance netlists. In both of these applications, the logic is likely to contain many redundancies—in [2] the designs that are considered contain 30-50% redundant gates. The gains are therefore likely to offset any two-level losses.

In this paper, we have been interested in compacting a boolean network for verification purposes. This is clearly a similar problem to the problem of optimizing a netlist during synthesis. There exists a rich literature on how this can be done, see for example [21], [22]. Our compression algorithm, and to a lesser degree standard two-level minimization, can be seen as a particular way to do *rule-based optimization*, where we have enumerated all rules applicable to a two-level window. An important difference between our approach and the standard rule-based optimization is that we have geared our rewriting in such a way that we are sure that we will be using negligible runtime.

The powerful logic optimization techniques that have been developed in the synthesis community are put to use in a verification context in [6], where the effect of optimizing a netlist before the invocation of a SAT solver is studied. The results demonstrate that this can lead to order of magnitude speedups, even when relatively few nodes are simplified away. The logic optimizer used in this case is the synthesis system SIS [23]. A significant difference to the work presented here is that the reduction algorithms that are used are of much higher complexity than the combination of our minimizer with RBCs, so they are only applicable to doing compaction of relatively small circuits—in the paper 20-bit integer factorization problems are considered.

VI. Conclusions

In this paper, we have experimentally evaluated the impact of varying the number of operators and the use of two-level minimization in a representation for boolean formulas and circuits. The studied problem is very important, and is becoming more so every day as more and more formal verification methods rely on algorithms that are sensitive to redundancies in the circuit.

Our results have shown that the thesis that the use of a single operator in the representation will allow greater sharing does not seem to hold in the realm of representing industrial circuits where all obvious combinational and sequential redundancies already have been removed. Moreover, we have shown that while two-level rewriting is a powerful concept, it may cause circuit blowup when sharing that already is present in the circuit gets destroyed. Finally, we have presented a simple minimization algorithm that can remove tens of thousands of nodes even after all standard simplifications such as constant propagation, operand reordering and sharing of isomorphic parts of the DAG have been done. This conceptually simple algorithm, paired with the uncomplicated RBC representation, outperforms all of the alternatives that we have evaluated, in some cases saving an order of magnitude more nodes compared to the best competitor. Moreover, the algorithm is not only easier to implement than standard two-level minimization, it is also guaranteed to run in provable $O(N)$ time.

Acknowledgments

This work was done while Per Bjesse was employed by Prover Technology. The authors would like to thank Poul F. Williams for answering questions about BEDs.

References

[1] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[2] M. K. Ganai and A. Kuehlmann, "On-the-fly compression of logical circuits," in *Proc. IEEE/ACM Int. Workshop on Logic Synthesis*, 2000.

[3] K. McMillan, "Interpolation and SAT-based model checking," in *Proc.* 13^{th} *Int. Conf. on Computer Aided Verification*, 2001.

[4] C. Leiserson and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1990.

[5] J. Baumgartner and A. Kuehlmann, "Min-area retiming on flexible circuit structures," in *Proc. Int. Conf. on Computer Aided Design*, 2001.

[6] R. Drechsler, "Using synthesis techniques in SAT-solvers," in *GI/ITG/GMM-Workshop, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2004.

[7] R. Kurshan, *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[8] P. Chauhan, E. M. Clarke, S. Sapra, J. Kukula, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *Formal Methods in Computer Aided Design*, 2002.

[9] E. Clarke, M. Talupur, H. Veith, and D. Wang, "SAT-based predicate abstraction for hardware verification," in *Proc.* 6^{th} *Int. Conf. on Theory and Practice of Satisfiability Testing*, 2003.

[10] H. R. Andersen and H. Huulgaard, "Boolean expression diagrams," in *Proc.* 12^{th} *IEEE Symp. on Logic in Computer Science*, 1997.

[11] P. A. Abdullah, P. Bjesse, and N. Eén, "Symbolic reachability analysis based on SAT-solvers," in *Proc. TACAS '00,* 9^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[12] H. Hulgaard, P. F. Williams, and H. R. Andersen, "Equivalence checking of combinational circuits using boolean expression diagrams," in *IEEE Trans. Computer-Aided Design*, 1999.

[13] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining decision diagrams and SAT procedures for efficient symbolic model checking," in *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, 2000.

[14] P. F. Williams, M. Nikolskaia, and A. Rauzy, "Bypassing BDD construction for reliability analysis," in *Information Processing Letters*, 2000.

[15] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource tool for symbolic model checking," in *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, 2002.

[16] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer Aided Design*, 2000.

[17] P. Bjesse, T. Leonard, and A. Mokkedem, "Finding bugs in an Alpha microprocessor using satisfiability solvers," in *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, 2000.

[18] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc.* 35^{th} *Design Automation Conference*, 1997.

[19] M. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver," in *Proc.* 40^{th} *Design Automation Conference*, 2002.

[20] A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based boolean reasoning," in *Proc.* 39^{th} *Design Automation Conference*, 2001.

[21] G. D. Micheli, *Synthesis and Optimization of Boolean Circuits*. Kluwer Academic Publishers, 1994.

[22] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. McGraw-Hill, 1996.

[23] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgat, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," University of Berkeley, Tech. Rep., 1992.