Applying Logic Synthesis for Speeding Up SAT

Niklas Een, Alan Mishchenko, Niklas Sörensson

Cadence Berkeley Labs, Berkeley, USA.
EECS Department, University of California, Berkeley, USA.
Chalmers University of Technology, Göteborg, Sweden.

Abstract. SAT solvers are often challenged with very hard problems that remain unsolved after hours of CPU time. The research community meets the challenge in two ways: (1) by improving the SAT solver technology, for example, perfecting heuristics for variable ordering, and (2) by inventing new ways of constructing simpler SAT problems, either using domain specific information during the translation from the original problem to CNF, or by applying a more universal CNF simplification procedure after the translation. This paper explores preprocessing of circuit-based SAT problems using recent advances in logic synthesis. Two fast logic synthesis techniques are considered: DAG-aware logic minimization and a novel type of structural technology mapping, which reduces the size of the CNF derived from the circuit. These techniques are experimentally compared to CNF-based preprocessing. The conclusion is that the proposed techniques are complementary to CNF-based preprocessing and speedup SAT solving substantially on industrial examples.

1 Introduction

Many of today's real-world applications of SAT stem from formal verification, test-pattern generation, and post-synthesis optimization. In all these cases, the SAT solver is used as a tool for reasoning on boolean circuits. Traditionally, instances of SAT are represented on conjunctive normal form (CNF), but the many practical applications of SAT in the circuit context motivates the specific study of speeding up SAT solving in this setting.

For tougher SAT problems, applying CNF based transformations as a preprocessing step [6] has been shown to effectively improve SAT run-times by (1) minimizing the size of the CNF representation, and (2) removing superfluous variables. A smaller CNF improves the speed of constraint propagation (BCP), and reducing the number of variables tend to benefit the SAT solver's variable heuristic. In the last decade, advances in logic synthesis has produced powerful and highly scalable algorithms that perform similar tasks on circuits. In this paper, two such techniques are applied to SAT.

The first technique, *DAG-aware circuit compression*, was introduced in [2] and extended in [11]. In this work, it is shown that a circuit can be minimized efficiently and effectively by applying a series of local transformations taking logic sharing into account. Minimizing the number of nodes in a circuit tends to reduce the size of the derived CNFs that are passed to the SAT engine. The

process is similar to CNF preprocessing where a smaller representation is also achieved through a series of local rewrites.

The second technique applied in this paper is technology mapping for lookup-table (LUT) based FPGAs. Technology mapping is the task of partitioning a circuit graph into cells with k inputs and one output that fits the LUTs of the FPGA hardware, while using as little area as possible. Many of the signals present in the unmapped circuit will be hidden inside the LUTs. In this manner, the procedure can be used to decide for which signals variables should be introduced when deriving a CNF, leading to CNF encodings with even fewer variables and clauses than existing techniques [14, 15, 9].

The purpose of this paper is to draw attention to the applicability of these two techniques in the context of SAT solving. The paper makes a two-fold contribution: (1) it proposes a novel CNF generation based on technology mapping, and (2) it experimenally demonstrated the practicality of the logic synthesis techniques for speeding up SAT.

2 Preliminaries

A combinational *boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. Incoming edges of a node are called *fanins* and outgoing edges are called *fanouts*. The *primary inputs* (PIs) of the network are nodes without fanins. The *primary outputs* (POs) are nodes without fanouts. The PIs and POs define the external connections of the network.

A special case of a boolean network is the *and-inverter graph* (AIG), containing four node types: PIs, POs, two-input AND-nodes, and the constant TRUE modelled as a node with one output and no inputs. Inverters are represented as attributes on the edges, dividing them into *unsigned* edges and *signed* (or complemented) edges. An AIG is said to be *reduced and constant-free* if (1) all the fanouts of the constant TRUE, if any, feed into POs; and (2) no AND-node has both of its fanins point to the same node. Furthermore, an AIG is said to be *structurally-hashed* if no two AND-nodes have the same two fanin edges including the sign. By decomposing k-input functions into two-input ANDs and inverters, any logic network can be reduced to an AIG implementing the same boolean function of the POs in terms of the PIs.

A *cut* C of node n is a set of nodes of the AIG, called *leaves*, such that any path from a PI to n passes through at least one leaf. A *trivial cut* of a node is the cut composed of the node itself. A cut is *k-feasible* if the number of nodes in it does not exceed k. A cut C is *subsumed* by C' of the same node if $C' \subset C$.

3 Cut Enumeration

Here we review the standard procedure for enumerating all k-feasible cuts of an AIG. Let Δ_1 and Δ_2 be two sets of cuts, and the merge operator $\otimes_k$ be defined as follows:

$$\Delta_1 \otimes_k \Delta_2 \;=\; \{\; C_1 \cup C_2 \;\mid\; C_1 \in \Delta_1,\; C_2 \in \Delta_2,\; |C_1 \cup C_2| \leq k \;\}$$

Further, let n_1, n_2 be the first and second fanin of node n, and let $\Phi(n)$ denote all k-feasible cuts of n, recursively computed as follows:

$$\Phi(n) \;=\; \begin{cases} \Phi(n_1) & ,n \in \text{PO} \\ \{\{n\}\} & ,n \in \text{PI} \\ \{\{n\}\} \;\cup\; \Phi(n_1) \otimes_k \Phi(n_2) & ,n \in \text{AND} \end{cases}$$

This formula gives a simple procedure for computing all k-feasible cuts in a single topological pass from the PIs to the POs. Informally, the cut set of an AND node is the trivial cut plus the pair-wise unions of cuts belonging to the fanins, excluding those cuts whose size exceeds k. Reconvergent paths in the AIG lead to generating subsumed cuts, which may be filtered out for most applications.

In practice, all cuts can be computed for $k \leq 4$. A partial enumeration, when working with larger k, can be achieved by introducing an *order* on the cuts and keeping only the L best cuts at each node. Formally: substitute Φ for Φ_L where $\Phi_L(n)$ is defined as the trivial cut plus the L best cuts of $\Delta_1 \otimes_k \Delta_2$.

4 DAG-Aware Minimization

The concept of DAG-aware minimization was introduced by Bjesse et. al. in [2], and further developed by Mishchenko et. al. in [11]. The method works by making a series of local modifications to the AIG, called rewrites, such that each rewrite reduces the *total* number of AIG nodes. To accurately compute the effect of a rewrite on the total number of nodes, *logic sharing* is taken into account. Two equally-sized implementations of a logical function may have different impact on the total node count if one of them contains a subgraph that is already present in the AIG (see *Figure 1*).

In [11] the authors propose to limit the rewrites to 4-input functions. There exists $2^{16} = 65536$ such functions. By normalizing the order and polarity of input

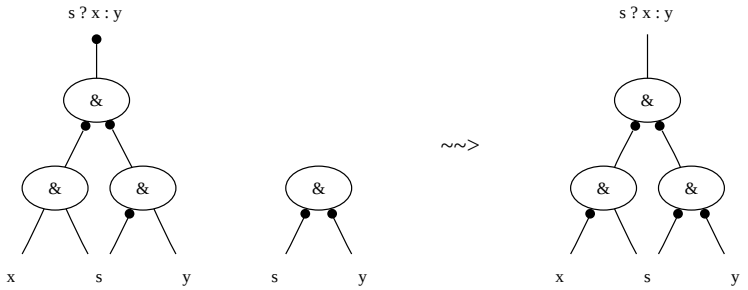

Fig. 1. Given a netlist containing the two fragments on the left, one node can be saved by rewriting the MUX "s ? x : y" to the form on the right, reusing the already present node "$\neg s \wedge \neg y$".

and output variables, these functions are divided into 222 equivalence classes.[1] Good AIG structures, or *candidate implementations*, for these 222 classes can be precomputed and stored in a table. The algorithm of [11] is reviewed below:

> ***DAG-Aware Minimization.*** Perform a 4-feasible cut enumeration, as described in the previous section, proceeding topologically from the PIs to the POs. During the cut enumeration, after computing the cuts for the current node n, try to improve its implementation as follows: For every cut C of n, let f be the function of n in terms of the leaves of C. Consider all the candidate implementations of f and choose the one that reduces the total number of AIG nodes the most. If no reduction is possible, leave the AIG unchanged; otherwise recompute the cuts for the new implementation of node n and continue the topological traversal.

Several components are necessary to implement this procedure:

- A cut enumeration procedure, as described in the previous section.
- A bottom-up topological iterator over the AIG nodes that can handle rewrites during the iteration.
- An incremental procedure for structural hashing. In order to efficiently search for the best substitution candidate, the AIG must be kept structurally-hashed, reduced and constant-free. After a rewrite, these properties may be violated and must be restored efficiently.
- A pre-computed table of good implementations for 4-input functions. We propose to enumerate all structurally-hashed, reduced and constant-free AIGs with 7 nodes or less, discarding candidates not meeting the following property: For each node n, there should be no node m in the subgraph rooted in n, such that replacing n with m leads to the same boolean function. Example: "$(a \wedge b) \wedge (a \wedge c)$" would be discarded since replacing the node "$(a \wedge b)$" with its subnode "b" does not change the function.
- An efficient procedure to evaluate the effect of replacing the current implementation of a node with a candidate implementation.

The implementation of the above components is straight-forward, albeit tedious. We observe that in principle, the topological iterator can be modified to revisit nodes as their fanouts change. When this happens, new opportunities for DAG-aware minimization may be exposed. Modifying the iterator in this way yields an idempotent procedure, meaning that nothing will change if it is run a second time. In practice, we found it hard to make such a procedure efficient.

A simpler and more useful modification to the above procedure is to run it several times with a *perturbation phase* in between. By changing the structure of the AIG, without increasing its size, new cuts can conservatively be introduced with the potential of revealing further node saving rewrites. One way of

[1] Often referred to as the NPN-classes, for Negation (of inputs), Permutation (of inputs), Negation (of the output).

perturbing the AIG structure is to visit all multi-input conjunctions and modify their decomposition into two-input AND-nodes. Another way is to perform the above minimization algorithm, but allow for zero-gain rewrites.

5 CNF through the Tseitin Transformation

Many applications rely on a some version of the Tseitin transformation [14] for producing CNFs from circuits. For completeness, we state the exact version compared against in our experiments. When the transformation is applied to AIGs, two improvements are often used: (1) multi-input ANDs are recognized in the AIG structure and translated into clauses as one gate, and (2) if-then-else expressions (MUXes) are detected in the AIG through simple pattern matching and given a specialized CNF translation. The clauses generated for these two cases are:

x $\leftrightarrow$ And(a_1, a_2, ..., a_n). Clause representation:

$$a_1 \wedge a_2 \wedge \ldots \wedge a_n \rightarrow x$$
$$\overline{a}_1 \rightarrow \overline{x},\ \overline{a}_2 \rightarrow \overline{x},\ \ldots\ , \overline{a}_n \rightarrow \overline{x}$$

x $\leftrightarrow$ ITE(s,t,f). If-then-else with selector s, true-branch t, false-branch f. Clause representation:

$$s \wedge t \rightarrow x \qquad \overline{s} \wedge f \rightarrow x \qquad \text{(red)}\ \ t \wedge f \rightarrow x$$
$$s \wedge \overline{t} \rightarrow \overline{x} \qquad \overline{s} \wedge \overline{f} \rightarrow \overline{x} \qquad \text{(red)}\ \ \overline{t} \wedge \overline{f} \rightarrow \overline{x}$$

The two clauses labeled "red" are redundant, but including them increases the strength of unit propagation. It should be noted that a two-input XOR is handled as a special case of a MUX with t and f pointing to the same node in opposite polarity. This results in representing each XOR with four three-literal clauses (the redundant clauses are trivially satisfied). In the experiments presented in section 7, the following precise translation was used:

- The *roots* are defined as (1) AND-nodes with multiple fanouts; (2) AND-nodes with a single fanout that is either complemented or leads to a PO; (3) AND-nodes that, together with its two fanin nodes, define an if-then-else.
- If a root node defines an if-then-else, the above translation with 6 clauses, including redundant clauses, is used.
- The remaining root nodes are encoded as multi-input ANDs. The scope of the conjunction rooted at n is computed as follows: Let S be the set of the two fanins of n. While S contains a non-root node, repeatedly replace that node by its two fanins. The above clause translation for multi-input ANDs is then used, unless the conjunction collected in this manner contains both x and $\neg x$, in which case, a unit clause coding for $x \leftrightarrow$ FALSE is used.
- Unlike some other work [7, 9], there is no special treatment of nodes that occur only *positively* or *negatively*.

6 CNF through Technology Mapping

Technology mapping is the process of expressing an AIG in the form representative of an implementation technology, such as standard cells or FPGAs. In particular, *lookup-table (LUT) mapping* for FPGAs consists in grouping AND-nodes of the AIG into logic nodes with no more than k inputs, each of which can be implemented by a single LUT.

Normally, technology mapping procedures optimize the area of the mapped circuit under delay constraints. Optimal delay mapping can be achieved efficiently [3], but is not desirable for SAT where size matters more than logic depth. Therefore we propose to map for area *only*, in such a way that a small CNF can be derived from the mapped circuit. In the next subsections, we review an improved algorithm for structural technology mapping [12].

6.1 Definitions

A *mapping* $\mathbf{M}$ of an AIG is a partial function that takes a non-PI (i.e. AND or PO) node to a k-feasible non-trivial cut of that node. Nodes for which mapping $\mathbf{M}$ is defined are called *active* (or mapped), the remaining nodes are called *inactive* (or unmapped). A *proper mapping* of an AIG meets the following three criteria: (1) all POs are active, (2) if node n is active, every leaf of cut $\mathbf{M}(n)$ is active, and (3) for every active AND-node m, there is at least one active node n such that m is a leaf of cut $\mathbf{M}(n)$. The *trivial mapping* (or mapping induced by the AIG) is the proper mapping which takes every non-PI node to the cut composed of its immediate fanins.

An *ordered cut-set* Φ_L is a total function that takes a non-PI node to a non-empty ordered sequence of L or less k-feasible cuts. In the next section, $\mathbf{M}$ and Φ_L as will be viewed as updateable objects and treated imperatively with two operations: For an inactive node n, procedure $\mathit{activate}(\mathbf{M}, \Phi_L, n)$ sets $\mathbf{M}(n)$ to the first cut in the sequence $\Phi_L(n)$, and then recursively activates inactive leaves of $\mathbf{M}(n)$. Similarly, for an active node n, procedure $\mathit{inactivate}(\mathbf{M}, n)$, makes node n inactive, and then recursively inactivates any leaf of the former cut $\mathbf{M}(n)$ that is violating condition (3) of a proper mapping.

Furthermore, $\mathit{nFanouts}(\mathbf{M}, n)$ denotes the number of fanouts of n in the subgraph induced by the mapping. The *average fanout* of a cut C is the sum of the number of fanouts of its leaves, divided by the number of leaves. Finally, the *maximally fanout-free cone* (MFFC) of node n, denoted $\mathit{mffc}(\mathbf{M}, n)$, is the set of nodes used exclusively by n. More formally, a node m is part of n's MFFC iff every path in the current mapping $\mathbf{M}$ from m to a PO passes through n. For an *inactive* node, $\mathit{mffc}(\mathbf{M}, \Phi_L, n)$ is defined as the nodes that would belong to the MFFC of node n if it was first activated.

6.2 A Single Mapping Phase

Technology mapping performs a sequence of refinement phases, each updating the current mapping $\mathbf{M}$ in an attempt to reduce the *total cost*. The cost of a

single cut, $cost(C)$, is given as a parameter to the refinement procedure. The total cost is defined as sum of $cost(\mathbf{M}(n_{act}))$ over all active nodes n_{act}.

Let $\mathbf{M}$ and Φ_L be the proper mapping and the ordered cut-set from the previous phase. A refinement is performed by a bottom-up topological traversal of the AIG, modifying $\mathbf{M}$ and Φ_L for each AND-node n as follows:

- All k-feasible cuts of node n (with fanins n_1 and n_2) are computed, given the sets of cuts for the children: $\Delta = \{\{n\}\} \cup \Phi_L(n_1) \otimes_k \Phi_L(n_2)$
- If the first element of $\Phi_L(n)$ is not in Δ, it is added. This way, the previously best cut is always eligible for selection in the current phase, which is a sufficient condition to ensure global monotonicity for certain cost functions.
- $\Phi_L(n)$ is set to be the L best cuts from Δ, where smaller cost, higher average fanout, and smaller cut size is better. The best element is put first.
- If n is active in the current mapping $\mathbf{M}$, and if the first cut of $\Phi_L(n)$ has changed, the mapping is updated to reflect the change by calling *inactivate*($\mathbf{M}$, n) followed by calling *activate*($\mathbf{M}$, Φ_L, n). After this, $\mathbf{M}$ is guaranteed to be a proper mapping.

6.3 The Cost of Cuts

This subsection defines two complementary heuristic cost function for cuts:

Area Flow. This heuristic estimates the *global* cost of selecting a cut C by recursively approximating the cost of other cuts that have to be introduced in order to accommodate cut C:

$$cost_{AF}(C) = area(C) + \sum_{n \in C} \frac{cost_{AF}(first(\Phi_L(n)))}{\max(1, nFanouts(\mathbf{M}, n))}$$

Exact Local Area. For nodes currently not mapped, this heuristic computes the total cost-increase incurred by activating n with cut C. For mapped nodes, the computations is the same but n is first deactivated. Formally:

$$mffc(C) = \bigcup_{n \in C} mffc(\mathbf{M}, \Phi_L, n)$$

$$cost_{ELA}(C) = \sum_{n \in mffc(C)} area(first(\Phi_L(n))$$

In standard FPGA mapping, each cut is given an area of 1 because it takes one LUT to represent it. A small but important adjustment for CNF generation is to define area in terms of the number of *clauses* introduced by that cut. Doing so affects both the area flow and the exact local area heuristic, making them prefer cuts with a small representation.

The boolean function of a cut is translated into clauses by deriving its *irredundant sum-of-products (ISOP)* using Minato-Morreale's algorithm [10] (reviewed

```
cover isop(boolfunc L, boolfunc U)
{
    if (L == FALSE) return ∅
    if (U == TRUE) return {∅}

    x = topVariable(L, U)
    (L0, L1) = cofactors(L, x)
    (U0, U1) = cofactors(U, x)

    c0    = isop(L0 ∧ ¬U1, U0)
    c1    = isop(L1 ∧ ¬U0, U1)
    Lnew  = (L0 ∧ ¬func(c0)) ∨ (L1 ∧ ¬func(c1))
    c*    = isop(Lnew, U0 ∧ U1)

    return ({x} ×  c0) ∪ ({¬x} ×  c1) ∪ c*
}
```

Fig. 2. *Irredundant sum-of-product generation.* A ***cover*** (= SOP = DNF) is a set, representing a disjunction, of *cubes* (= product = conjunction of literals). A cover c induces a boolean function $func(c)$. An *irredundant SOP* is a cover c where no cube can be removed without changing $func(c)$. In the code, ***boolfunc*** denotes a boolean function of a fixed number of variables $x_1, x_2, \ldots, x_n$ (in our case, the width of a LUT). L and U denotes the lower and upper bound on the cover to be returned. At top-level, the procedure is called with $L = U$. Furthermore, *topVariable(L, U)* selects the first variable, from a fixed variable order, which L or U depends on. Finally, $cofactors(F, x)$ returns the pair $(F[x = 0], F[x = 1])$.

in *Figure 2*). ISOPs are computed for both f and $\neg f$ to generate clauses for both sides of the bi-implication $t \leftrightarrow f(x_1, \ldots, x_k)$. For the sizes of k used in the experiments, boolean functions are efficiently represented using truth-tables. In practice, it is useful to impose a bound on the number of products generated and abort the procedure if it is exceeded, giving the cut an infinitly high cost.

6.4 The Complete Mapping Procedure

Depending on the time budget, technology mapping may involve different number of refinement passes. For SAT, only a very few passes seem to pay off. In our experiments, the following two passes were used, starting from the trivial mapping induced by the AIG:

- An initial pass, using the area-flow heuristic, $cost_{AF}$, which captures the global characteristics of the AIG.
- A final pass with the exact local area heuristic, $cost_{ELA}$. From the definition of local area, this pass cannot increase the total cost of the mapping.

Finally, there is a trade-off between the quality of the result and the speed of the mapper, controlled by the cut size k and the maximum number of cuts stored at each node L. To limit the scope of the experimental evaluation, these parameters were fixed to $k = 8$ and $L = 5$ for all benchmarks. From a limited testing, these values seemed to be a good trade-off. It is likely that better results could be achieved by setting the parameters in a problem-dependent fashion.

7 Experimental Results

To measure the effect of the proposed CNF reduction methods, 30 hard SAT problems represented as AIGs were collected from three different sources. The first suite, "Cadence BMC", consists of internal Cadence verification problems, each of which took more than one minute to solve using SMV's BMC engine. Each of the selected problem contains a bug and has been unrolled upto the length k, which reveals this bug (yielding a satisfiable instance) as well as upto length $k-1$ (yielding an unsatisfiable instance).

The second suite, "IBM BMC", is created from publically available IBM BMC problems [16]. Again, problems containing a bug were selected and unrolled to length k and $k-1$. Problems that MiniSat could not solve in 60 minutes were removed, as were problems solved in under 5 seconds.

Finally, the third suite, "SAT Race", was derived from problems of *SAT-Race 2006*. Armin Biere's tool "cnf2aig", part of the AIGER package [1], was applied to convert the CNFs to AIGs. Among the problems that could be completely converted to AIGs, the "manol-pipe" class were the richest source. As before, very hard and very easy problems were not considered.

For the experiments, we used the publically available synthesis and verification tool ABC [8] and the SAT solver MiniSat2. The exact version of ABC used in these experiments, as well as other information useful for reproducing the experimental results presented in this paper, can be found at [5].

Clause Reduction. In *Table 1* we compare the difference between generating CNFs using only the Tseitin encoding (section 5) and generating CNFs by applying different combinations of the presented techniques, as well as CNF preprocessing [6] (as implemented in MiniSat2). Reductions are measured against the Tseitin encoding. For example, a reduction of 62% means that, on average, the transformed problem contains 0.38 times the original number of clauses.

We see a consistent reduction in the CNF size, especially in the case where the CNF was derived using technology mapping. The preprocessing scales well, although its runtime, in our current implementation, is not negligible.

For space reasons, we do not present the total number of literals. However, we note that: (1) the speed of BCP depends on the number of clauses, not literals; (2) deriving CNFs from technology mapping produces clauses of at most size $k+1$, which is 9 literals in our case; and (3) in [6] it was shown that CNF preprocessing in general does not increase the number of literals significantly.

SAT Runtime. In *Table 2* we compare the SAT runtimes of the differently preprocessed problems. Runtimes do *not* include preprocessing times. At this stage, when the preprocessing has not been fully optimized for the SAT context, it is arguably more interesting to see the potential speedup. If the preprocessing is too slow, its application can be controlled by modifying one of the parameters (such as the number or width of cuts computed), or preprocessing may be delayed until plain SAT solving has been tried for some time without solving the problem. Furthermore, for BMC problems, the techniques can be applied before unrolling the circuit, which is significantly faster (see *Incremental BMC* below).

Speedup is given both as a total speedup (the sum total of all SAT runtimes) and as arithmetic and harmonic average of the individual speedups. For BMC, we see a clear gain in the proposed methods, most notably for the Cadence BMC problems where a total speedup of 6.9x was achieved not using SATELITE-style preprocessing, and 5.3x with SATELITE-style preprocessing (for a total of 22.3x speedup compared to plain SAT on Tseitin). However, the problems from the SAT-Race benchmark exhibit a different behavior resulting in an increased runtime. It is hard to explain this behavior without knowing the details of the benchmarks. For example, equivalence checking problems are easier to solve if the equivalent points in the modified and golden circuit are kept. The proposed methods may remove such pairs, making the problems harder for the SAT solver.

CNF Generation based on Technology Mapping. Here we measure the effect of using the number of CNF clauses as the size estimator of a LUT, rather than a unit area as in standard technology mapping. In both cases, we map using LUTs of size 8, keeping the 5 best cuts at each node during cut enumeration. The results are presented in *Table 5*. As expected, the proposed technique lead to fewer *clauses* but more *variables*. In these experiments, the clause reduction almost consistently resulted in shorter runtimes of the SAT solver.

Incremental BMC. An alternative and cheaper use of the proposed techniques in the context of BMC, is to minimize the AIG before unrolling. This prevents simplification across different time frames, but is much faster (in our benchmarks, the runtime was negligible). The clause reduction and the SAT runtime using DAG-aware minimization are given in *Table 4*. In this particular experiment, ABC was not used, but an in-house Cadence implementation of DAG-aware minimization and incremental BMC. Ideally, we would like to test the CNF generation based on technology mapping as well, but this is currently not available in the Cadence tool. For licence reasons, IBM benchmarks could not be used in this experiment. Instead, 5 problems from the TIP-suite [1] were used, but they suffer from being too easy to solve.

8 Conclusions

The paper explores logic synthesis as a way to speedup the solving of circuit-based SAT problems. Two logic synthesis techniques are considered and experimentally evaluated. The first technique applies recent work on DAG-aware circuit compression to preprocess a circuit before converting it to CNF. In spirit, the approach is similar to [4]. The second technique directly produces a compact CNF through a novel adaptation of area-oriented technology mapping, measuring area in terms of CNF clauses.

Experimental results on several sets of benchmarks have shown that the proposed techniques tend to substantially reduce the runtime of SAT solving. The net result of applying both techniques is a 5x speedup in solving for hard industrial problems. At the same time, some slow-downs were observed on benchmarks from the previous year's SAT Race. This indicates that more work is needed for understanding the interaction between the circuit structure and the heuristics of a modern SAT-solver.

9 Acknowledgements

The authors acknowledge helpful discussions with Satrajit Chatterjee on technology mapping and, in particular, his suggestion to use the average number of fanins' fanouts as a tie-breaking heuristic in sorting cuts.

	Clause Reduction (k clauses)								**Preprocessing Time** (sec)						
Problem	(orig)	S	D	DS	T	TS	DT	DTS	S	D	DS	T	TS	DT	DTS
Cdn1-70u	160	113	69	43	54	41	36	29	1	6	7	14	15	11	12
Cdn1-71s	166	117	71	44	55	43	37	30	1	6	6	14	15	12	12
Cdn2-154u	682	452	467	310	312	257	282	254	6	31	35	48	51	66	68
Cdn2-155s	693	459	475	316	318	262	287	259	7	32	36	49	52	67	69
Cdn3.1-18u	1563	813	952	511	905	529	506	306	12	91	99	151	159	189	193
Cdn3.1-19s	1686	898	1028	559	977	593	547	336	12	98	107	162	170	208	212
Cdn3.2-19u	1684	899	1027	561	977	578	547	337	12	98	106	163	171	206	210
Cdn3.2-20s	1807	979	1102	611	1049	612	588	368	13	104	114	175	184	219	224
Cdn3.3-19u	1686	897	1027	560	977	578	547	338	12	100	109	163	171	204	208
Cdn3.3-20s	1809	974	1103	611	1049	647	588	368	14	104	113	174	183	224	229
ibm18-28u	151	95	72	55	67	54	50	48	1	5	6	11	11	11	12
ibm18-29s	158	99	75	57	70	56	53	50	1	5	6	11	12	12	12
ibm20-43u	253	156	127	97	120	99	89	85	2	10	11	19	20	20	21
ibm20-44s	259	161	131	100	123	101	91	88	2	10	11	19	20	21	21
ibm22-51u	415	269	211	160	201	174	149	143	4	16	17	31	33	33	34
ibm22-52s	425	275	216	164	205	178	153	147	4	16	18	32	33	34	34
ibm23-35u	231	147	116	86	100	85	80	76	2	9	9	17	18	18	19
ibm23-36s	239	152	120	89	103	89	83	78	2	9	10	17	18	19	19
ibm29-25u	53	35	28	21	22	20	18	17	0	2	2	4	4	5	5
ibm29-26s	55	36	29	22	24	21	19	18	0	2	3	5	5	5	5
c10id-s	293	273	280	258	177	159	167	151	2	20	21	31	33	46	48
c10nidw-s	643	593	612	563	416	380	394	363	4	47	52	77	84	119	126
c6nidw-i	154	142	147	134	97	89	93	87	1	10	11	18	19	26	27
c7b	41	36	39	33	27	26	26	25	0	3	3	5	6	7	8
c7b-i	40	36	38	33	27	26	26	25	0	3	4	5	5	7	8
c9	23	20	20	17	15	14	13	12	0	2	2	3	3	4	4
c9nidw-s	535	489	507	465	340	312	326	300	4	39	42	66	71	96	101
g10b	128	116	127	111	87	82	83	76	1	9	10	15	16	23	24
g10id	258	240	254	234	161	147	156	143	2	20	21	30	32	47	49
g7nidw	119	110	118	107	78	72	75	70	1	8	8	13	14	20	21
Avg. red.	–	29%	32%	47%	46%	56%	57%	62%							

Table 1. *CNF generation with different preprocessing.* "**(orig)**" denotes the original Tseitin encoding; "**D**" DAG-Aware minimization; "**T**" CNF generation through Technology Mapping; "**S**" SATELITE style CNF preprocessing. On the left, the number of clauses in the CNF formulation is given, in thousands. On the right, the runtimes of applied preprocessing are summed up. No column for the time of generating CNFs through Tseitin encoding is given, as they are all less than a second. The "Cdn" problems are internal Cadence BMC problems; the "ibm" problems are IBM BMC problems from [16]; the remaining ten problems are the "manol-pipe" problems from SAT-Race 2006 [13] back-converted by "cnf2aig" into the AIG form.

Problem	**SAT Runtime** (sec) – Cadence BMC (orig)	S	D	DS	T	TS	DT	DTS
Cdn1-70u	21.9	12.3	3.6	3.1	2.5	4.1	**1.2**	1.3
Cdn1-71s	15.2	8.8	7.7	3.9	**2.1**	3.1	4.0	2.7
Cdn2-154u	116.4	48.3	41.1	37.7	11.6	34.4	15.6	**9.3**
Cdn2-155s	101.8	22.9	12.9	16.2	18.2	50.6	13.4	**6.9**
Cdn3.1-18u	1516.0	139.4	361.9	119.4	196.3	78.8	78.8	**39.0**
Cdn3.1-19s	1788.2	276.7	535.0	154.8	317.8	137.1	131.9	**42.5**
Cdn3.2-19u	403.8	214.4	239.8	169.7	140.9	**73.7**	114.8	78.1
Cdn3.2-20s	3066.1	893.4	1002.9	353.2	376.2	313.5	687.5	**96.5**
Cdn3.3-19u	316.1	225.6	133.9	104.7	107.9	107.6	**53.2**	55.0
Cdn3.3-20s	2305.4	456.4	863.1	385.8	507.0	236.9	307.2	**101.2**
Total speedup:		4.2x	3.0x	7.2x	5.7x	9.3x	6.9x	22.3x
Arithmetic average speedup:		3.9x	3.6x	6.5x	6.3x	7.6x	9.2x	19.7x
Harmonic average speedup:		2.7x	2.9x	4.8x	5.3x	4.9x	6.6x	11.5x

Problem	**SAT Runtime** (sec) – IBM BMC (orig)	S	D	DS	T	TS	DT	DTS
ibm18-28u	83.7	82.6	39.2	41.9	45.0	54.2	23.2	**18.5**
ibm18-29s	93.6	47.6	46.8	25.1	36.9	23.5	25.9	**20.9**
ibm20-43u	805.5	890.1	402.3	488.0	540.3	283.6	219.9	**215.1**
ibm20-44s	1260.2	278.4	305.6	**83.8**	277.2	422.2	265.7	303.6
ibm22-51u	361.8	194.6	109.2	88.6	145.8	170.8	**67.0**	82.5
ibm22-52s	408.4	489.0	148.3	135.7	187.2	177.9	120.5	**91.3**
ibm23-35u	540.3	365.9	264.2	241.5	260.1	220.2	181.4	**130.7**
ibm23-36s	856.2	743.4	527.9	356.8	436.2	585.7	**144.7**	238.1
ibm29-25u	329.7	375.6	39.0	29.4	42.9	56.6	28.5	**11.4**
ibm29-26s	71.3	190.5	41.7	**20.9**	71.5	31.5	28.0	25.4
Total speedup:		1.3x	2.5x	3.2x	2.4x	2.4x	4.4x	4.2x
Arithmetic average speedup:		1.5x	3.0x	4.9x	2.8x	2.8x	4.7x	6.5x
Harmonic average speedup:		1.0x	2.4x	3.1x	2.1x	2.4x	4.0x	4.3x

Table 2. *SAT runtime with different preprocessing.* "**(orig)**" denotes the original Tseitin encoding; "**D**" DAG-Aware minimization; "**T**" CNF generation through Technology Mapping; "**S**" SATELITE style CNF preprocessing. Given times do *not* include preprocessing, only SAT runtimes. Speedups are relative to the "**(orig)**" column.

Problem	**SAT Runtime** (sec) – SAT Race (orig)	S	D	DS	T	TS	DT	DTS
c10id-s	26.7	**5.1**	25.1	23.6	50.6	25.2	49.8	14.7
c10nidw-s	710.5	624.7	700.3	880.4	383.6	698.1	**212.7**	856.6
c6nidw-i	414.4	267.1	734.7	412.5	244.5	**209.7**	540.1	710.3
c7b	**29.4**	167.2	76.3	58.4	34.6	43.9	63.9	435.5
c7b-i	101.4	54.2	68.1	52.0	**49.5**	93.2	293.4	154.5
c9	**10.8**	51.2	11.4	32.8	11.8	21.0	44.1	83.1
c9nidw-s	**122.5**	625.2	246.9	864.8	287.2	446.7	952.6	285.2
g10b	385.3	388.8	446.0	183.6	**106.5**	225.6	291.2	182.5
g10id	736.0	350.7	524.0	723.9	98.3	**92.0**	190.6	188.4
g7nidw	119.4	24.8	78.3	67.3	**13.5**	17.2	63.6	37.8
Total speedup:		1.0x	0.9x	0.8x	2.1x	1.4x	1.0x	0.9x
Arithmetic average speedup:		1.8x	1.0x	1.1x	2.8x	2.3x	1.3x	1.4x
Harmonic average speedup:		0.5x	0.8x	0.6x	1.2x	0.9x	0.5x	0.3x

Table 3. *SAT runtime with different preprocessing (cont. from Table 2).*

Problem	Nodes before and after minimization	BMC runtimes before and after minimization
Cdn1	3,527 → 949	37.8 s → 9.6 s
Cdn2	7,918 → 3,126	17.5 s → 0.8 s
Cdn3.1	84,718 → 28,637	607.1 s → 275.3 s
Cdn3.3	84,698 → 28,611	>1 h → 1823.7 s
Cdn4	2,936 → 1,538	>1 h → >1 h
nusmv:tcas$_5$	2,661 → 1,975	9.11 s → 2.27 s
nusmv:tcas$_6$	2,656 → 1,965	4.12 s → 0.67 s
texas.parsesys$_1$	11,860 → 939	0.64 s → 0.03 s
texas.two-proc$_2$	791 → 335	0.23 s → 0.01 s
vis.eisenberg	720 → 306	1.63 s → 2.01 s

Table 4. *Incremental BMC on original and minimized AIG.* The above problems all contain bugs. Runtimes are given for performing incremental BMC upto the shortest counter example. In the columns to the right of the arrows, the design has been minimized by DAG-aware rewriting before unrolling it. The node count is the number of ANDs in the design. Note that in this scheme, there can be no cross-timeframe simplifications. The experiment confirms the claim in [2] of the applicability of DAG-aware circuit comparession to formal verification. The original paper only listed compression ratios and did not include runtimes.

Problem	Technology Mapping for CNF		
	#clauses	#vars	SAT-time
Cdn1-70u	62 k → 54 k	12 k → 15 k	6.6 s → 4.1 s
Cdn1-71s	64 k → 55 k	13 k → 15 k	6.6 s → 3.1 s
Cdn2-154u	327 k → 312 k	58 k → 77 k	23.3 s → 34.4 s
Cdn2-155s	333 k → 318 k	58 k → 78 k	21.4 s → 50.6 s
Cdn3.1-18u	1990 k → 905 k	145 k → 248 k	125.9 s → 78.8 s
Cdn3.1-19s	2147 k → 977 k	156 k → 267 k	161.2 s → 137.1 s
Cdn3.2-19u	2146 k → 977 k	156 k → 266 k	189.9 s → 73.7 s
Cdn3.2-20s	2302 k → 1049 k	167 k → 285 k	501.6 s → 313.5 s
Cdn3.3-19u	2147 k → 977 k	156 k → 267 k	136.4 s → 107.6 s
Cdn3.3-20s	2302 k → 1049 k	167 k → 285 k	311.7 s → 236.9 s

Table 5. *Comparing CNF generation through standard technology mapping and technology mapping with the cut cost function adapted for SAT.* In the adapted CNF generation based on technology mapping (**righthand side of arrows**), the area of a LUT is defined as the number of clauses needed to represent its boolean function. In the standard technology mapping (**lefthand side of arrows**), each LUT has unit area "1". In both cases, the mapped design is translated to CNF by the method described in section 6.4, which introduces one variable for each LUT in the mapping. The standard technology mapping minimizes the number of LUTs, and hence will have a lower number of introduced variables. From the table it is clear that using the number of clauses as the area of a LUT gives significantly fewer clauses, and also reduces SAT runtimes.

References

1. A. Biere. **AIGER** (AIGER is a format, library and set of utilities for And-Inverter Graphs (AIGs)). *http://fmv.jku.at/aiger/*.
2. P. Bjesse and A. Boralv. **DAG-Aware Circuit Compression For Formal Verification**. In *Proc. ICCAD'04*, 2004.
3. D. Chen and J. Cong. **DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs.** In *ICCAD*, pages 752–759, 2004.
4. R. Drechsler. **Using Synthesis Techniques in SAT Solvers**. *Technical Report, Intitute of Computer Schience, Unversity of Bremen, 28359 Bremen, Germany*, 2004.
5. N. Een. http://www.cs.chalmers.se/~een/SAT-2007.
6. N. Een and A. Biere. **Effective Preprocessing in SAT through Variable and Clause Elimination**. In *Proc. of Theory and Applications of Satisfiability Testing, 8^{th} International Conference (SAT'2005)*, volume 3569 of *LNCS*, 2005.
7. N. Een and N. Sörensson. **Translating Pseudo-Boolean Constraints into SAT**. In *Journal on Satisfiability, Boolean Modelling and Computation (JSAT)*, volume 2 of *IOS Press*, 2006.
8. B. L. S. Group. **ABC: A System for Sequential Synthesis and Verification**. *http://www.eecs.berkeley.edu/~alanmi/abc/*.
9. P. Jackson and D. Sheridan. **Clause Form Conversions for Boolean Circuits**. In *Theory and Appl. of Sat. Testing, 7th Int. Conf. (SAT'04)*, volume 3542 of *LNCS, Springer*, 2004.
10. S. Minato. **Fast Generation of Irredundant Sum-Of-Products Forms from Binary Decision Diagrams**. In *Proc. SASIMI'92*.
11. A. Mishchenko, S. Chatterjee, and R. Brayton. **DAG-aware AIG rewriting: A fresh look at combinational logic synthesis**. In *Proc. DAC'06*, pages 532–536, 2006.
12. A. Mishchenko, S. Chatterjee, and R. Brayton. **Improvements to Technology Mapping for LUT-based FPGAs**. volume 26:2, pages 240–253, February 2007.
13. C. Sinz. **SAT-Race 2006 Benchmark Set**. *http://fmv.jku.at/sat-race-2006/*.
14. G. Tseitin. **On the complexity of derivation in propositional calculus**. *Studies in Constr. Math. and Math. Logic*, 1968.
15. M. N. Velev. **Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors**. *Proc. of Conf. on Asia South Pacific Design Aut. (ASP-DAC)*, 2004.
16. E. Zarpas. **Benchmarking SAT Solvers for Bounded Model Checking**. In *Proc. SAT'05*, number 3569 in LNCS. Springer-Verlag, 2005.