Versatile SAT-based Remapping for Standard Cells

Alan Mishchenko Robert Brayton

Department of EECS, UC Berkeley

{alanmi, brayton}@berkeley.edu

Thierry Besson Sriram Govindarajan

Harm Arts Paul van Besouw

ICD Synthesis, Mentor Graphics

{Firstname_Lastname}@mentor.com

Abstract

The paper describes a versatile framework for optimizing a mapped netlist according to user-specified requirements to reduce a combination of area, delay, and power. A variety of delay models is supported. The framework is based on a SAT-based engine enumerating gate-level implementations of a node while using the node's full flexibility in the network. The enumeration favors considering low-cost alternatives early in the process. Full flexibility at a node is constructed as a Boolean circuit representing a Boolean relation, which is sampled by recursive cofactoring using different variable orders. Each variable order can result in an 8-input truth table of a function at the node, compatible with the relation. The computation is made efficient by reusing minterm samples of the relations at different nodes.

1. Introduction

In modern design flows a mapped netlist is often optimized using a cost function. For example, it is often desirable to reduce area while preserving timing, or to improve timing at the cost of a reasonable area increase. In both cases, the user provides a cost-function, which is used by the design flow to perform the required optimization.

This paper presents a versatile SAT-based framework that can be integrated into a design flow to enable user-guided remapping of a netlist with a specific goal in mind. The framework is versatile because it can work with practically any cost function. It is SAT-based because it relies on Boolean satisfiability [2] to compute alternative implementations of the target node.

To our knowledge, this is the first SAT-based framework, which efficiently explores the space of feasible standard-cell mappings. Previous work includes a BDD-based framework for standard-cell remapping [3] and a SAT-based framework for LUT-based remapping.

The proposed framework uses a novel SAT-based procedure to enumerate standard-cell implementations of the target node. The procedure performs recursive sampling of a Boolean circuit, representing the Boolean relation of the full flexibility of the target node and candidate divisors while taking don't-cares into account. The procedure exploits structural analysis, guided simulation, counter-examples derived by the SAT solver, and a cache of successful proofs to enable efficient exploration of the search space of alternative implementations.

The rest of the paper is organized as follows. Section 2 contains necessary background. Section 3 gives a top-level view of the remapping framework. Section 4 describes library preprocessing. Section 5 explains the SAT-based Boolean relation solver. Experimental results are given in Section 6. Section 7 concludes the paper.

2. Background

2.1 Boolean function

In this paper, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function f is the set of variables X, which influence the output value of f. The support size is denoted by $|X|$.

Expressions $\bar{x}$ and x are the *negative literal* and the *positive literal* of variable x, respectively. "Negative" and "positive" are *polarities* of variable x in the literals.

Boolean function $R(X, Y, Z)$ in terms of input variables X, intermediate variables Y, and output variables Z, can be seen as a characteristic function of a Boolean relation relating valuations of inputs X and outputs Z. Thus, valuation $(x_1, x_2, \ldots, x_n; z_1, z_2, \ldots, z_m)$ belongs to the relation if there exists a valuation $(y_1, y_2, \ldots, y_k)$ of intermediate variables such that $R(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_k, z_2, \ldots, z_m) = 1$.

In this paper, we will be using the SAT solver to check whether a given minterm $(x_1, x_2, \ldots, x_n; z_1, z_2, \ldots, z_m)$ belongs to the relation specified as a CNF formula.

2.2 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting the nodes.

A node n has zero or more fanins, i.e. nodes driving n, and zero or more fanouts, i.e. nodes driven by n. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A transitive fanin (fanout) cone (TFI/TFO) of node n is a subset of the network nodes, reachable through the fanin (fanout) edges of the node. If the network is sequential, it contains registers whose inputs/outputs are treated as additional POs/PIs.

Internal flexibilities of a node in a network arise because of limited controllability and observability. Lack of controllability occurs because some combinations of values are never produced at the fanins of the node. Lack of observability occurs because the node's value does not have impact on the values of the POs under some values of the PIs. Examples can be found in [7].

These internal flexibilities result in *don't-cares* at the node *n*. The complement of the don't-cares is the *care set*.

Given a network with PIs *x* and PO functions $\{z_i(x)\}$, the care set $C_n(x)$ of a node *n* is a Boolean function of the PIs:

$$C_n(x) = \sum_i [z_i(x) \oplus z_i'(x)]$$

where $z_i'(x)$ are the POs in a copy of the network but with node *n* is complemented, as shown in Figure 2 [7].

2.3 Mapped network

A *mapped network* is a Boolean network whose nodes are associated with cells (gates) from the given standard cell library represented in Liberty format [4]. Each cell in the library is characterized by its name, Boolean function, and various parameters. The parameters relevant for this work, are cell area and delay information.

Area is a floating point number associated with each cell. Delay information is given as a set of two-dimensional tables specifying delay and slew at the output of the cell as functions of capacitance and slew at each of the inputs.

The presented remapping engine uses a load-independent abstraction of the delay information, which approximates the delay at the output using the delay at each of the inputs based on the gain-based approach [10]. The timing model can be extended to handle a load-dependent delay model; however, this extension is beyond the scope of this paper.

A typical standard-cell library contains multiple cells, with the same function, that differ only in cell size. In this work it is assumed that each cell has only one size. Removing this restriction is left for future work.

Finally, although a typical library contains sequential cells, this paper is limited to combinational logic mapping.

2.4 Boolean satisfiability

A *satisfiability problem* (SAT) takes a propositional formula representing a Boolean function and decides if the formula is satisfiable or not. The formula is *satisfiable* (SAT) if there is an assignment of variables that evaluates the formula to 1. Otherwise, the formula is *unsatisfiable* (UNSAT). A software program that solves SAT problems is called a *SAT solver*. SAT solvers provide a satisfying assignment when the problem is satisfiable.

Modern SAT solvers can accept assumptions, which are single-literal clauses holding for one call to the SAT solver. The process of determining the satisfiability of a problem under given assumptions is called *incremental SAT solving*.

2.5 Conjunctive Normal Form (CNF)

To represent a propositional formula for the SAT solver, important aspects of the problem are encoded using Boolean *variables*. The presence or absence of a given aspect is represented by a positive or negative *literal* of the variable. A disjunction of literals is called a *clause*. A conjunction of clauses is called a *CNF*. CNFs can be processed efficiently by mainstream CNF-based SAT solvers, such as MiniSAT [2], used in this work.

Deriving a CNF for a subset of nodes of the Boolean network is performed by putting together CNFs obtained by converting each node. A CNF for a node is derived by deriving SOPs of the on-set and off-set of the Boolean function of the node, and converting these SOPs into CNF using the De Morgan rule.

3. Remapping framework

Remapping is done by a software framework designed to perform several interdependent tasks. The computation begins by preprocessing the standard cell library, which involves collecting functions up to 8 inputs realizable using at most two cells from the library.

Next, remapping considers nodes in an order determined by the optimization goals. For example, area optimization is performed by processing nodes in a topological order, while delay optimization is performed by computing the critical paths and optimizing nodes on the paths.

For each target node, the remapping framework performs the following steps, as shown in Figure 1:

- It computes a structural window centered around a target node and containing nodes in the limited TFI/TFO of the target node, along with nodes found on the reconvergent paths.
- It identifies a subset of nodes in the window, which can be used as divisors to express the target node. It orders the candidate divisors in reduced desirability. For example, for delay optimization, the order first lists the nodes having short delay and low criticality.
- It converts the window into CNF and derives a SAT instance representing the complete flexibility of the target node [7]. The SAT variables corresponding to the target node and those of the candidate divisors are identified and stored. These will be used to express assumptions during subsequent SAT solving.
- The SAT instance is used to derive one or more implementations of the target node using candidate divisors, as described in Section 5.
- The implementations are matched by Matcher with the current pre-processed library, timed by Timer using the selected timing model, and compared by Evaluator against the current implementation using the given cost-function, as shown in Figure 1.

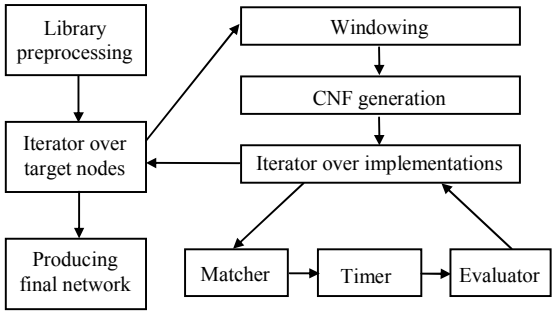

Figure 1. Illustration of a miter used in CDC computation.

The best implementations among those improving the cost functions, are used to replace the current target node. When the optimization is finished or a timeout occurs, the final network is produced and returned to the user.

4. Standard-cell library preprocessing

A typical standard-cell library contains cells with different functionalities and sizes. In the preprocessing phase, representative cells are selected from each functionality class and resource-aware enumeration of cell pairs is done, deriving two-cell super-cells with support up

to 8 inputs [6]. The resulting super-cells are hashed using their truth tables as hash keys while keeping track of their area and pin-to-pin delays. Super-cells that are identical to (or dominated by) other super-cells are discarded.

The preprocessing takes a fraction of a second and is performed only once for a given library even if remapping is performed repeatedly or for several designs.

In principle, the preprocessing can compute super-cells with more than 8 inputs containing more than two cells. However, experiments indicate that this is unlikely to improve the quality of results, while pre-computation time and memory used for storage, could increase dramatically. This might make it necessary to do this offline and then to read in the result from a file.

5. SAT-based Boolean relation solver

5.1 Constructing SAT instance

The SAT instance is constructed by deriving CNF for the Boolean relation representing the care set of the target node in the structural window carved out in the network. The window contains a fixed number of TFI/TFO levels of logic centered at the node, plus all paths originating in the limited TFI and terminating in the limited TFO.

The computation of care set is based on the formulation used for node optimization with don't-cares [7] illustrated in Figure 2. The circuit contains two copies of the node's TFO, one of which has an inverter inserted at the node n output. The comparator XORs are added for the pairs of corresponding outputs and the output of the comparator's OR gate is assumed to be 1, meaning that complementing the node's function makes a difference at the POs, and thus the constructed circuit represents the care set of the node.

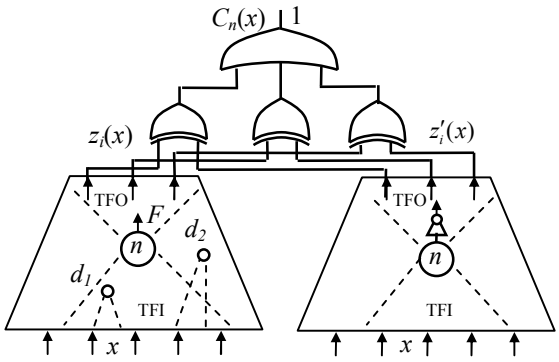

Figure 2. Deriving Boolean relation for computing multiple implementations of the target node in terms of candidate divisors.

The Boolean relation represented by the circuit structure in Figure 2 relates the function of the node n with functions of candidate divisors whose support is a subset of the support of the node's TFI, such as divisors d_1 and d_2, shown at the bottom of Figure 2.

5.2 Deriving a feasible implementation of the node

Figure 3 contains the pseudo-code of procedure SolveBR_rec() used to derive one implementation of the target node that is in agreement with the Boolean relation. Recursive calls to SolveBR_rec() (Line 5) build the resulting implementation as a co-factoring tree whose nodes are multiplexers and leaves are constants or elementary variables. The tree is collapsed on the fly and its function is returned as a truth table. To reduce the clutter in the pseudo-code, it is not shown that the procedure returns "None" if the support has more than 8 variables.

The procedure SolveBR_rec() takes an ordered set of unassigned divisors and a set of divisors currently assigned a value. The assigned divisors are used as assumptions in the SAT calls, which check if the cofactors of Boolean relation w.r.t. the divisor variables are UNSAT. For example, if the following three cofactors are all UNSAT: (1) $F = 0$ & $d_1 = 1$ & $d_2 = 1$, (2) $F = 1$ & $d_1 = 0$, (3) $F = 1$ & $d_2 = 0$, the node can be implemented as $F = \text{AND}(d_1, d_2)$.

Procedure SolveBR_rec() in Figure 3 begins by detecting trivial cases (Lines 1 and 2). In particular, if constraining the node's function F to a constant is UNSAT under the assumptions, the complement constant function is returned. Next (Line 3), the procedure tries to implement the cofactor of a target node as a candidate divisor d. This can be done if the following two conditions are UNSAT: (1) $F = 0$ & $d = 1$ and (2) $F = 1$ & $d = 0$. To prove that the target node can be expressed as an inverter, we show that the following are UNSAT: (1) $F = 0$ & $d = 0$ and (2) $F = 1$ & $d = 1$.

Finally, if the node's function is not a constant and cannot be expressed in terms of one divisor, the next unassigned variable is selected for cofactoring (Line 4). The resulting functions computed for the two cofactors are multiplexed with the given variable and returned (Line 5). Note that forthcoming cofactoring is effected by adding the selected variable or its complement to the set of assigned variables.

Multiple implementations of the target node can be derived using procedure SolveBR(), shown in Figure 3. The implementations differ because different variable orders are used to cofactor the Boolean relation. It is possible that two different variable orders lead to the same implementation, but in practice such situations are rare. Each implementation is checked against the pre-computed super-cell library for a Boolean match and evaluated according to the cost function.

5.3 Using counter-examples to speed up search

The input space of the Boolean relation is the set of all candidate divisors, $\{d_i\}$. The output space is the target node, F. All other variables used to represent the relation in the SAT solver are treated as intermediate variables, according to the discussion in Section 2.1.

Counter-examples produced by the SAT solver are either care set minterms of the function of the target node, according to the Boolean relation. These minterm samples are collected and represented as two *sample matrices*, one for the onset and one for the offset. The rows of the sample matrices represent minterms while the columns represent candidate divisors. The matrices are reused while working on the same node.

While enumerating feasible implementations of the node, we will check satisfiability of many cofactors of the Boolean relation. Recursive cofactoring of the relation is mirrored in our implementation by recursive splitting of the onset/offset sample matrices. This amounts to selecting a subset of samples that agree with the current values assigned to the divisor variables. Only when an onset/offset sample matrix is empty (there are no samples that agree

with the assigned divisors), is the SAT solver run to check if the cofactor is a constant, or if it is satisfiable. In the latter case, the new minterm provided by the counter-example is added to the appropriate sample matrix.

A significant reduction in runtime (2-5x) is obtained due to the use of the sample matrices in this application.

5.4 Reusing counter-examples across windows

In many SAT-based applications, including this one, the runtime is dominated by satisfiable SAT calls. Each SAT call produces a relevant on-set or off-set minterm of the target node in terms of candidate divisors and helps us converge on a feasible implementation of the node.

The number of satisfiable SAT calls can be achieved by seeding the sample matrices described in Section 5.3 with care-set minterms computed using random simulation. However, it was found experimentally that randomly generated minterms are not as useful, for guiding the search, as minterms generated by the SAT solver. Thus, when we move to work on a new target node, instead of simulating the node's window randomly, we re-simulate counter-examples computed by the SAT solver for previous nodes. To this end, when optimization of a node is completed, we store counter-examples generated for this node in an internal data-structure. When a new node is selected, the backed-up counter-examples are retrieved and re-simulated through the new node's window. At the end of the simulation, those counter-examples that do not belong to the care-set of the new node are removed while the remaining ones are used to seed the sample matrices.

5.5 Caching successful proofs

While trying to find a feasible implementation of the target node, the SAT solver is called repeatedly with different sets of assumptions. Each of these calls checks the satisfiability of one cofactor of the Boolean relation. Since different unrelated variable orders are used, often the same cofactors are checked multiple times. To avoid duplicated SAT calls, a cache of *unsatisfiable* calls can be maintained. The unsatisfiable calls are cached by remembering sets of assumptions that were tried and found unsatisfiable.

6. Experimental results

The proposed optimization framework for standard cells was implemented as command *mfs3* in ABC [1]. This command differs from two optimization frameworks developed earlier for LUT-based FPGAs (command *mfs* and its improved version, *mfs2*) [8].

The main difference between optimization for K-LUTs and that for standard cells, is that the former expresses the target node using any K-input function, while the proposed framework uses only functions expressible in terms of standard cells. Expressing the resulting functions using standard cells requires a dedicated SAT-based engine introduced in this paper.

The experimental evaluation reported in this section is meant to illustrate the use of the framework and to show its scalability; it is not meant to be a comprehensive evaluation of remapping with different cost functions. Thus, the experimental results are limited to area-oriented optimization using typical cells from an industrial library where a unit-area model has been imposed. Under these assumptions, only the number of cells is optimized, without considering the actual area and delay listed in the library.

The benchmarks considered are ten combinational logic cones used in earlier publications by the first author [9]. The benchmarks were preprocessed by logic synthesis and mapped into the target library using two area-oriented mappers in ABC (commands *amap* and *&nf –R 1000*). The proposed remapping engine is applied in each case as a post-processing step. Combinational equivalence checking was performed using command *&cec* in ABC.

The results are shown in Table 1. The number of primary inputs and outputs is reported in columns *Inputs* and *Outputs*. The number of cells after mapping is reported in columns *amap* and *&nf*. The number of cells after remapping is reported in columns *+mfs3*. The table shows that the proposed engine has reduced the number of cells by 2.5% after running *amap* and by 0.5% after running *&nf*. It should be noted that these improvements are on top of well-tuned heuristic mappers, which perform area-optimization mode with the same unit-area version of the library. Moreover, calling the mappers repeatedly without synthesis does not produce similar improvements.

In this experiment, command *mfs3* was used with command line options (*mfs3 -ae -I 4 -O 2*) selected to limit the scope of optimization to 4 levels of TFI and 2 levels of TFO. With these options, the total runtime of *mfs3* is close to that of each of the mappers (*amap* and *&nf*) and was about 3 minutes for all the logic cones listed in Table 1.

The transcript of running *mfs3* for one testcase (*ex02*) from Table 1 is shown in Figure 4. Besides numerous internal parameters, the transcript shows the runtime breakdown for individual tasks performed by the optimizer. In particular, the SAT solver performs 5.4 million incremental calls while trying to optimize 34,817 nodes with changes introduced to 8,764 nodes. The SAT solver runtime accounts for the 97% of the total time (11 seconds), meaning that the framework is running with little overhead on top of the SAT solver.

7. Conclusions

The paper describes a novel SAT-based optimization framework applicable to netlists mapped into standard cells. The engine considers one node at a time, and looks for alternative implementations of the node using one or two cells from the library. The implementations are evaluated using a cost-function and the best is accepted, before moving on to the next node. The engine is novel in that it expresses implementations of a node in terms of standard cells from a given library rather than arbitrary functions [8]. Efficient implementation is based on incremental SAT used to sample the Boolean relation of a node's full flexibility represented as a CNF formula.

Future work will focus on customizing the framework to support different cost functions and on developing other flavors of SAT-based mapping. It may be also interesting to optimize the implementation by using cubes rather then minterms in representing the sampling matrix, which may lead to improved runtime and better quality of results.

8. REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] N. Een and N. Sörensson, "An extensible SAT-solver", *Proc. SAT'03*, LNCS 2919, pp. 502-518.

[3] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization". *Proc. DAC'04*, pp. 438-441.

[4] Liberty Format. https://www.opensourceliberty.org/

[5] A. Malik, R. K. Brayton, A. R. Newton, and A. Singiovanni-Vincentelli, "Two-level minimization of multi-valued functions with large offsets", *IEEE TCAD'91*, Vol. 10(4), pp. 413-424.

[6] A. Mishchenko, X. Wang, and T. Kam, "A new enhanced constructive decomposition and mapping algorithm", *Proc. DAC '03*, pp. 143-148.

[7] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *DATE '05*, pp. 418-423.

[8] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang. "Scalable don't-care-based logic optimization and resynthesis", *ACM TRETS*, Vol. 4(4), April 2011, Article 34.

[9] A. Mishchenko, "Enumeration of irredundant circuit structures", *Proc. IWLS'14*.

[10] I. E. Sutherland and R. F. Sproull, "Logical effort: designing for speed on the back of an envelope", *Proc. VLSI'91*, p.1-16.


```
one_boolean_function SolveBR_rec( ordered set of divisors D, assigned divisors A, cnf C ) {
    if ( F == 1 is UNSAT under assumptions in A ) return 0;  // constant 0 Boolean function   (Line 1)
    if ( F == 0 is UNSAT under assumptions in A ) return 1;  // constant 1 Boolean function   (Line 2)
    if ( F == d_i or F == d_i' for some divisor d_i in D under assumptions in A )  return d_i or d_i' ;  (Line 3)
    d = next variable in D;                                                                  (Line 4)
    return d ? SolveBR_rec( D \ d, A ∪ d, C ) : SolveBR_rec( D \ d, A ∪ d_i', C );           (Line 5)
}
set_of_boolean_functions SolveBR( number of functions N, set of divisors D, cnf C ) {
    result = ∅;
    for ( i = 0; i < N; i = i + 1 ) {
        generate a new ordering of divisors in D;
        result = result ∪ SolveBR_rec( D, ∅, C );
    }
    return result;
}
```

Figure 3: Pseudo-code of SAT-based solving of Boolean relation during remapping.

```
abc 01> r ex02.aig;  amap; ps;  mfs3 -aev -I 4 -O 2; ps;  time; echo
ex02 : i/o =25237/18422  lat =    0  nd = 34817  edge = 110193  area =34817.00  delay =308.95  lev = 13
Library processing: Var = 6. Cell = 71. Fun = 38660. Obj = 38660. Ave = 1.00. Skip = 0. Time = 0.07 sec
Remapping parameters: TFO = 2. TFI = 4. FanMax = 10. MffcMin = 1. MffcMax = 3. DecMax = 1. Effort = yes.
Node = 34817. Try = 34817. Change = 8764.   Buf = 44. Inv = 7655. Gate = 1065. AndOr = 0. Effort = 962.
MaxDiv = 111. MaxWin = 136.   AveDiv = 8. AveWin = 11.   Calls = 5442733. (Sat = 2698400. Unsat =
2744333.)
Lib    =    0.07 sec (  0.62 %)
Win    =    0.12 sec (  1.06 %)
Cnf    =    0.10 sec (  0.88 %)
Sat    =   10.98 sec ( 97.08 %)
 Sat   =    8.19 sec ( 72.41 %)
 Unsat =    1.73 sec ( 15.30 %)
Eval   =    0.00 sec (  0.00 %)
Timing =    0.00 sec (  0.00 %)
Other  =    0.04 sec (  0.35 %)
ALL    =   11.31 sec (100.00 %)
Cone sizes:  1=7699  2=5  3=164  4=34  5=861  6=1    Gate sizes:  1=7867  2=897
Reduction:   Nodes    1210 out of  34817 (  3.48 %)   Edges    1702 out of 110193 (  1.54 %)
ex02  : i/o =25237/18422  lat =    0  nd = 33607  edge = 108491  area =33607.00  delay =308.95  lev = 13
```

Figure 4: The transcript produced by running "mfs3" during area-only remapping of a design.

Table 1: Applying area-oriented remapping after area-oriented mapping using two mappers.

Benchmark	Inputs	Outputs	amap	+mfs3	&nf	+mfs3
Ex01	13601	13601	54795	54058	49127	48879
Ex02	25237	18422	34817	33607	26696	26587
Ex03	16300	11243	51169	50435	44783	44408
Ex04	28341	26312	80846	80505	80547	80434
Ex05	26380	22788	82100	81196	71887	71654
Ex06	52711	45549	164256	162319	143673	143206
Ex07	18677	12441	56600	55853	48734	48399
Ex08	15780	12892	78558	72502	53348	52976
Ex09	36087	31407	56949	55190	44856	44764
Ex10	11400	10896	23945	23151	20296	20160
Geomean1			1.000	0.975	0.842	0.838
Geomean2					1.000	0.995