SAT-based Redundancy Removal

Krishanu Debnath
Synopsys India Private Ltd.,
Bengaluru, India

Rajeev Murgai
Synopsys India Private Ltd.,
Noida, India

Mayank Jain, Janet Olson
Synopsys
Mountain View, California, USA

Abstract

Logic optimization is an integral part of digital circuit design. It reduces design area and power consumption, and quite often improves circuit delay as well. Redundancy removal is a key step in logic optimization, in which redundant connections in the circuit are determined and replaced by constant values 0 or 1. The resulting circuit is simplified, resulting in area and power savings. In this paper, we describe a redundancy removal approach for combinational circuits based on a combination of logic simulation and SAT. We show that this approach can handle large industrial-strength designs in a reasonable amount of CPU time.

1 Introduction

Optimizing a gate-level netlist is an important step in the synthesis of digital circuits. Optimization involves sharing of common expressions, logic simplification, logic rewriting, etc. Redundancy removal (RR) is one technique used to simplify a gate-level netlist, reducing its area, power, and often delay. Also, it makes the circuit more testable.

There are different ways to simplify or minimize a logic circuit. Two-level Boolean function minimization minimizes the sum-of-products (SOP) representation of a Boolean function by using the minimum number of prime implicants [7]. In a multi-level netlist, two-level minimization is applied to the SOP representation of each node, often using the don't cares associated with the node [10]. Another way to simplify a multi-level netlist is to identify connections that either always evaluate to a constant (logic 0 or 1), or when replaced by a constant, do not affect the primary output values. These are called **redundant connections** or **redundancies**. By replacing a redundant connection with the constant value, the netlist can be simplified by propagating the constant in the fanout cone of the connection and also removing the resulting floating logic in the fanin cone.

The paper is organized as follows. Section 2 gives an introduction to the redundancy removal problem and an overview of a SAT solver. Section 3 provides details of the SAT-based redundancy removal algorithm. A summary of the experimental results on industrial-strength designs is provided in Section 4. We conclude with some directions for future research in Section 5.

2 Background

Given a gate-level netlist or circuit, any connection is a potential fault site. The connection can be between a primary input and an input pin of a gate, between a gate's output pin and another gate's input pin, or between a gate's output pin and a primary output. A stuck-at-0/stuck-at-1 (SA-0/SA-1) fault f at a connection is **testable** if there exists an input vector that generates different values at a primary output with and without the fault. Such an input vector is called a **test** for the fault f. A SA-0/SA-1 fault f at a connection is **untestable** if replacing the connection by constant 0/1 does not change the value of any circuit output for any input vector. In other words, no test exists for an untestable fault. Checking if a fault is untestable can be done using Automatic Test Pattern Generation (ATPG) algorithms such as D algorithm [9] and PODEM [2]. ATPG algorithms that use a SAT solver have also been proposed [5, 11].

There is a one-to-one correspondence between redundant connections and untestable faults. Hence, ATPG algorithms can be used to find redundancies in a circuit. Once a redundancy is found, it can be replaced by the appropriate constant. The resulting floating (or dead) logic gates in the fanin cone of the connection can be removed, since they do not fan out anywhere. Also, the constant can be propagated to the fanout gates, simplifying the fanout logic.

Figure 1 shows an example circuit with two primary inputs A and B, and a primary output Z. It has two gate instances: an XOR2 followed by AO12 (i.e., an AND-OR gate). The global function $Z = A + B$. The connection between A and the AO12 gate is redundant (or stuck-at-1 untestable), since by removing this connection and replacing it with constant 1 does not change the global function of the output Z. Hence, this connection can be replaced with the constant 1. The AO12 gate can then be simplified to an OR2 gate, which has smaller area.

Let C be a circuit with a single output Z (multiple-output case can be handled similarly). If a fault f is introduced in C, let the resulting faulty circuit be C^f with output Z^f. Let us construct a circuit with output $Y = Z \bigoplus Z^f$. Figure 2 shows the circuit for Y. The fault f is testable if there exists an input vector i such that $Z(i) \neq Z^f(i)$. In other words, f is testable if $Y = 1$ for some input vector. This can be easily formulated as a SAT problem, as shown in the next section.

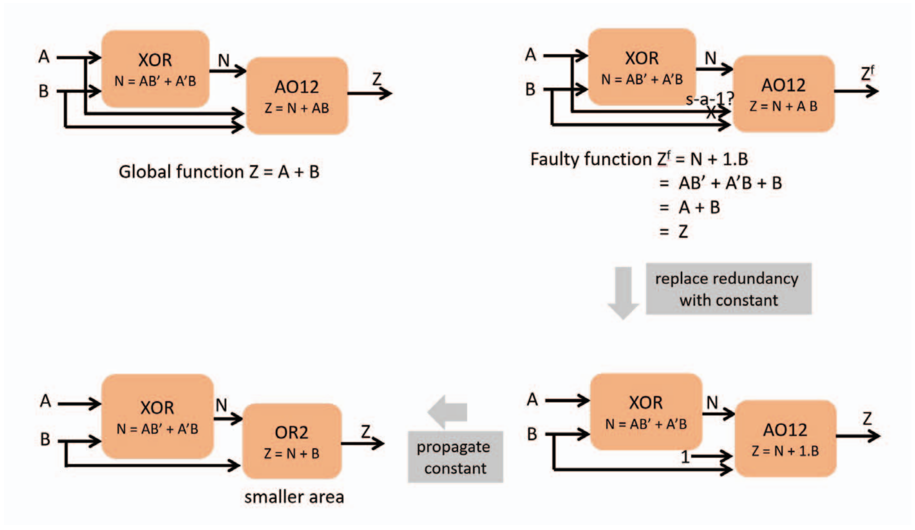

Figure 1: Redundancy Removal example

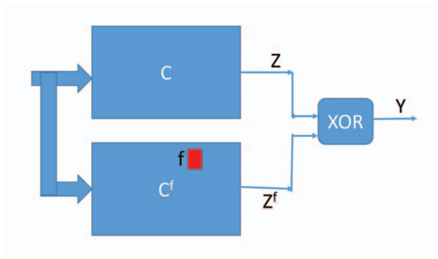

Figure 2: Formulating redundancy identification as a SAT problem

2.1 SAT Solver Basics

Given a Boolean formula, a SAT Solver finds a variable assignment such that the formula evaluates to true, or proves that no such assignment exists. A formula is represented as a conjunctive normal form (CNF), i.e., a product of one or more clauses.

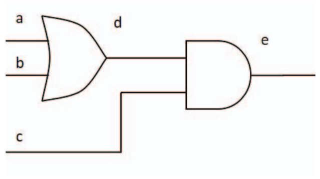

Figure 3: Example of a circuit

A circuit can be represented as CNF. For example, CNF representation of the circuit in Figure 3 is $(a'+d)(b'+d)(a+b+d')(d+e')(c+e')(c'+d'+e)$. The first three clauses define the OR gate and the last three, the AND gate. The first clause, $(a'+d)$, represents the condition that $a = 1 \Rightarrow d = 1$. The third clause, $(a + b + d')$, represents the condition that $\{(a = 0)(b = 0)\} \Rightarrow d = 0$.

The pseudo-code of an iterative SAT Solver [8, 6] is shown in Figure 4.

```
sat_solver(cnf)
while (true) {
    if (decide()) {
        while (bcp() == conflict) {
            backtrack_level = analyze_conflicts()
            if (backtrack_level < 0) {
                return unsatisfiable
            } else {
                backtrack(backtrack_level)
            }
        }
    } else {
        return satisfiable
    }
}
```

Figure 4: Outline of an iterative SAT Solver algorithm

The procedure **decide** selects some unassigned variable and assigns it a value (0 or 1). Various variable selection heuristics have been proposed; the ones in Chaff [8] and BerkMin [3] are among the most popular heuristics.

The procedure **bcp** (Boolean Constraint Propagation) deduces the variable assignments required to make the clauses and the CNF formula satisfiable. These variable assignments are implied from the previous variable assignments. For example, consider the CNF formula $(a' + d)(b' + d)(a + b + d')(d+e')(c+e')(c'+d'+e)$ of the circuit of Figure 3. If we assign $e = 1$, to make the clause $(d + e')$ satisfiable we must assign $d = 1$. A clause is called *unit clause* if it has all but one of its literals evaluate to 0. The remaining unassigned literal of the unit clause is called *unit literal*. For example, after assigning $e = 1$, $(d + e')$ and $(c + e')$ become unit clauses, with d and c being the unit literals respectively. Therefore, **bcp**'s task is to identify all the unit clauses and set their unit literals to true. The **bcp** procedure does this iteratively until either no unit clause is left in the clause database, or it finds a conflict. A conflict occurs when a variable is implied to be true (1) as well as false (0).

The procedure **analyze_conflict** creates conflict clauses, which capture the information that certain variable assignments lead to a conflict. During future search, the solver can avoid the conflicts by avoiding the search space covered by the conflict clauses. It also computes a backtracking level from the conflict clauses. More details on conflict analysis can be found in [6].

The procedure **backtrack** undoes all the variable assignments between the current decision level and the backtracking level.

2.2 Hybrid SAT Solver

Boolean reasoning problems from domains such as ATPG and verification are typically derived from the circuit structure. So it is possible to improve the efficiency of the SAT procedure for RR using the circuit structure information (note that the structure information is lost when we model the satisfiability problem using CNF). A circuit-based SAT Solver uses the circuit structure by working directly on the circuit [4]. However, conflict analysis of a circuit-based SAT Solver is not as efficient as that of a CNF-based SAT Solver. A Hybrid SAT Solver combines the benefits of both CNF- and circuit-based SAT Solvers [1].

In a Hybrid SAT Solver, the original problem is represented in circuit format, and the learnt information (i.e., conflict clauses) is recorded as CNF clauses. The circuit is typically maintained in the form of an And-Inverter graph. All procedures (**decide**, **bcp**, **analyze_conflict**) can work on both gates and CNF clauses. The solver exploits both circuit-based and CNF-based heuristics in the **decide** procedure. As the circuit consists of basic primitive gates, it is possible to implement efficient **bcp** on the circuit using a fast table lookup scheme [4]. As the **bcp** procedure of a SAT solver takes majority of the runtime, efficient circuit-based **bcp** improves the overall runtime of the SAT Solver significantly.

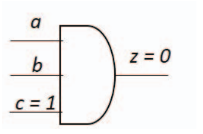

Figure 5: Example of an unjustified gate

To solve the RR problem, our Hybrid SAT Solver uses the notion of *unjustified gates*. A gate whose output has been assigned, but its assigned inputs do not imply the output value, is called an *unjustified gate*. For example, in Figure 5, c = 1 does not imply z = 0. We must assign either a or b to 0. Unjustified gates are created by **bcp**.

3 SAT-based Redundancy Removal

The pseudo-code for our RR algorithm is shown in Figure 6. A circuit with N wires can have at most 2N single stuck-at-value faults. The RR algorithm first reduces the total number of faults using fault collapsing and populates a fault list. Then, it uses Random Test Generator (RTG) and filters out faults that are proven testable by RTG. The RR algorithm iterates on the faults in the pruned fault list. It invokes Hybrid SAT Solver (HybridSAT) on each stuck-at-value fault f to check if f is untestable. If f is untestable, the algorithm replaces the connection involved in f with the corresponding constant value. After all the faults have been processed, as the final step, the constants are propagated and all the floating logic gates are removed, simplifying the circuit.

```
remove_redundancies(C)
decompose the circuit C to And-Inverter gates
fault_list = collect stuck-at-value faults of the circuit C after fault collapsing
pruned_fault_list = filter out the testable faults from fault_list using
                    random test generator
for each fault in pruned_fault_list {
    status = use HybridSAT Solver to determine if the fault is testable
    If (status == untestable) {
        replace the fault pin with constant
    }
}
run constant propagation and floating logic removal in the circuit C
```

Figure 6: Procedure to remove redundancies from a circuit

3.1 Fault collapsing

Fault collapsing is used to reduce the total number of faults in a circuit. We use the concepts of equivalence faults and dominant faults for fault collapsing.

Two faults f_1 and f_2 are **equivalent** if all the tests that detect f_1 also detect f_2, and vice-versa. We keep only one fault from a set of equivalent faults in the fault list. For example, stuck-at-0 fault at the input pins and the output pin of an AND gate are equivalent. So we keep just one fault instead of three.

If all tests of a fault f_1 also detect another fault f_2, f_2 is said to **dominate** f_1. Then f_2 can be removed from the fault list. For example, stuck-at-1 fault at the output of an AND gate dominates stuck-at-1 fault at each of the input pins. So we keep the stuck-at-1 fault at the input pins and remove the output fault from the fault list.

3.2 Random Test Generation

We can detect a large number of faults as testable using Random Test Generation (RTG). In RTG, we simulate the circuit using n random input vectors and store the output values of the simulation. Then, for every fault we insert the fault (n-bit wide) at the fault site and propagate it. The fault is testable if any of the n bits of any output changes value. Testable faults are removed from the fault list and not passed to the HybridSAT solver.

3.3 HybridSAT Solver

The HybridSAT Solver algorithm to determine if a test exists for a stuck-at-value fault f is as follows.

1. HybridSAT Solver considers two circuits, a good circuit and a faulty circuit. In the faulty circuit, the fault f is injected. In the good circuit, opposite value of the fault f is introduced at the fault site.

2. In both the circuits, HybridSAT Solver propagates the value from the fault site to some primary output by path sensitization. A path is sensitized by assigning non-controlling values to all the side inputs. Since we need to justify the non-controlling values of the side inputs, we store the side inputs in an *unjustified gate list* L.

3. Once the fault has propagated to some primary output P in the faulty circuit and the opposite value of the fault has propagated to P in the good circuit, HybridSAT Solver needs to excite the fault f at the fault site in both the circuits. To do this, HybridSAT solver assigns opposite value of the fault at the driver of the fault site and stores the driver in L.

4. If HybridSAT Solver succeeds in justifying all the gates in L, it has found a test for f.

HybridSAT Solver has user-controllable limits on the basic operations. These controls are applied on each individual fault. If HybridSAT Solver cannot determine the testability of a fault within these limits, it aborts the fault and proceeds to check the testability of the next fault in the pruned fault list. These controls on individual faults provide a trade-off between runtime and QoR of the algorithm.

4 Results

We compare the SAT-based RR algorithm described in Section 3 with a technique that uses an explicit implementation of the D algorithm. About 270 combinational netlists extracted from industrial designs are the testcases used for this evaluation. Each netlist corresponds to a hierarchy in the design. The largest netlist has about 1.2 million literals. The average fault-list size (after fault collapsing) is about 216K. The SAT-based algorithm is on average 3X faster, finds more redundancies and removes 50% more literals as compared to the explicit D-algorithm based implementation. The maximum runtime is less than 6 minutes.

5 Conclusions

In this paper, we presented a SAT-based redundancy removal technique for combinational circuits. It uses a combination of fault collapsing, random test generation, and a SAT solver to solve the redundancy removal problem. We show that this combination provides an effective way to solve the redundancy removal problem on large industrial-strength circuits in a reasonable CPU time. This paper focused on combinationally redundant faults. Identification and removal of sequential redundancies in a large circuit should be a promising research area.

References

[1] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik. Combining Strengths of Circuit-based and CNF-based Algorithms for a High-Performance SAT Solver. In *DAC*, 2002.

[2] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, C30(3):215–222, March 1981.

[3] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT Solver. In *DATE*, 2002.

[4] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *DAC*, 2001.

[5] T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(8), January 1992.

[6] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, 1996.

[7] E. McCluskey. Minimization of Boolean Functions. *Bell Laboratories Technical Journal*, November 1956.

[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, 2001.

[9] J. P. Roth. Diagnosis of Automata Failures: a Calculus and a Method. In *IBM Journal of Research and Development*, volume 10, pages 278–291, July 1966.

[10] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[11] P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Combinational Test Generation using Satisfiability. *IEEE Transactions on Computer-Aided Design*, 15(9), 1996.