FPGA based Accelerator for 3-SAT Conflict Analysis in SAT Solvers

Mona Safar[1], M. Watheq El-Kharashi[2], and Ashraf Salem[3]

[1] Computer and Systems Department, Ain Shams University, Cairo, Egypt
[2] University of Victoria, Victoria, Canada
[3] Mentor Graphics Egypt, Cairo, Egypt

Abstract. We present an FPGA-based accelerator for 3-SAT clause evaluation and conflict diagnosis and propose an approach to incorporate it in solving the Combinational Equivalence Checking problem. SAT binary clauses are mapped onto an implication graph and the ternary clauses are kept in an indexed clause database and mapped into the clause evaluator and conflict analyzer on FPGA

1 Introduction

Algorithms to solve SAT involve many compute-intensive, logic bit-level, and highly parallelizable operations that make reconfigurable computing appealing [1]. Various approaches have been proposed to accelerate SAT solving using reconfigurable computing [2]. An important module of any SAT solver is a clause evaluator that checks the consistency of variables assignment. Conflict diagnosis [3] helps pruning search space. We present an FPGA-based 3-SAT clause evaluator and conflict analyzer.

Combinational Equivalence Checking (CEC) is a widely used formal verification methodology for digital systems. Due to its hardness, current checkers mostly combine multiple checking engines, like BDD [4], SAT [5], and SAT/BDD [6]. We propose an approach to incorporate our clause evaluator to accelerate SAT-based CEC.

2 FPGA based 3-SAT Clause Evaluator and Conflict Analyzer

The architecture of our clause evaluator (Fig. 2) consists of a 3-input LUT acting as 3-input OR gate for each ternary clause, and an active low priority encoder that detects conflict when a clause is unsatisfiable and returns to the host the clause index. The value of each variable is encoded in two bits corresponding to its positive and negative literals. If the value of the variable is "0" or "1", the encoding is "01" or "10", respectively. The encoding "11" indicates that the variable is free.

Since same architecture is used for different SAT instances, except for the literal configurations, direct modification can be done on the bitstream reducing the synthesis and place-and-route overhead. Run-time reconfiguration can be used for dynamic clause addition. It also allows configurations larger than the available FPGA capability [1]. For large number of variables, virtual wires time multiplexing can be used [7].

Table 1 shows the number of 4-input LUTs needed for the encoder part of the architecture for some cases from DIMACS benchmark suite. AIM, DUBOIS, and PRET families consist of ternary clauses, so they map directly into our architecture.

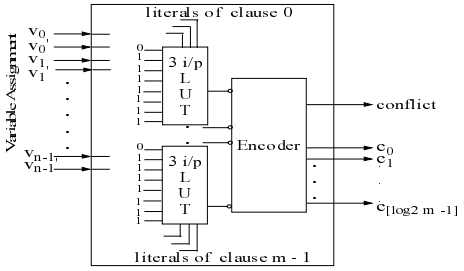

Fig.2. The clause evaluator, n is the number of variables, m is the number of ternary clauses, and (C_0 C_1 ….. $C_{[\log 2m-1]}$) represents the binary equivalent of the first clause that evaluates to 0.

Table 1. Number of 4-input LUTs occupied by the encoder component for DIMACS instances.

Instance	# Clauses	# 4-input LUTs
aim-50-1_6-z-j[1]	80	178
aim-50-6_0-z-j	300	692
aim-100-2_0-z-j	200	453
aim-100-3_4-z-j	340	811
dubois21	168	390
dubois30	240	545
pret60-xx[2]	160	370
pret150-xx	400	1007

3 SAT-based CEC Accelerator

Since SAT is an NP complete problem, using it in CEC transforms a problem that in worst case takes exponential time in the number of the circuit inputs into another problem that takes exponential time in the number of variables. Interestingly, most of the clauses of CNF formula produced by combinational circuits are binary that can be easily mapped onto an implication graph [8]. Ternary or more clause information can guide the search. A conflict arises when one of these clauses is unsatisfiable. Conflict diagnosis determines the backtracking level which helps pruning the search space [3].

Fig. 3 illustrates our proposed software/reconfigurable hardware accelerator for SAT-based CEC. The software running on the host computer first converts the structural description of the miter circuit into 3-SAT formula. The binary clauses are mapped into an implication graph. The ternary clauses are kept in an indexed clause database on the host computer and mapped into the clause evaluator on the FPGA.

[1] zzzz is "no" or "yes1", the former denoting a no-instance and the latter a single-solution yes-instance. j means simply the j^{th} instance at that parameter.
[2] xx is the horn percentage, it can take values 25,40,60 or 75.

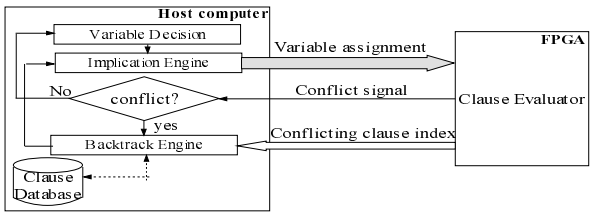

Fig.3. The general view of our software/ reconfigurable hardware SAT based CEC accelerator

Initially all variables are free except for the miter output which is restricted to the value "1". Selecting a free variable and assigning a value to it is done by the software running on the host computer. A decision level is associated with each selected assignment. The implication engine derives the direct and transitive implications of this assignment by operating on the implication graph generated from the binary clauses. The consistency of the variables binding with the ternary clauses is checked at each decision level via the clause evaluator. In case of conflict, the backtracking engine, aided by the conflicting clause index, determines the predecessor set of variables that are responsible for the conflict and hence performs nonchronological backtracking, which helps pruning the search space. The algorithm is shown on the next page. *clause_evaluator( )* sends variables assignment to the FPGA. The FPGA sends back a conflict indication and the index of the unsatisfied clause, if any.

Our architecture 3-SAT restriction is imposed on the structural description of the circuit by having gates of a maximum fan-in of 2, which adds new variables and increases the number of binary and ternary clauses (see Table 2).

4 Conclusions

We presented FPGA-based clause evaluator and conflict detector and proposed a new approach for the combinational equivalence-checking problem. In our approach, the SAT binary clauses are treated on the software side where they are mapped onto an implication graph. The ternary clauses are kept in an indexed clause database and mapped onto the clause evaluator and conflict detector on FPGA. The need for accelerating the ternary clause evaluation is proved using the ISCAS'85 benchmark.

References

1. K. Compton and S. Hauck: Reconfigurable Computing: A Survey of Systems and Software. ACM computing Surveys, Vol. 34, no. 2 (June 2002) 171-210
2. I Skliarova and A. B. Ferrari: Reconfigurable Hardware SAT Solvers: A Survey of Systems IEEE Transactions on Computers, Vol. 53, no. 11 (November 2004) 1449-1461
3. L. M. Silva and K. A. Sakallah: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Trans. Computers, Vol. 48, no. 5 (May 1999) 506-521
4. R. E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Computers, Vol. C-35, No. 8 (August 1986)

5. J. P. Marques-Silva and L. Guerra e Silva: Solving Satisfiability in Combinational Circuits. IEEE Design and Test of Computers (July-August 2003) 16-21
6. S. Reda and A. Salem: Combinational Equivalence Checking using Boolean Satisfiability and Binary Decision Diagrams. IEEE/ACM Design, Automation and Test in Europe (March 2001) 122-126
7. P. Zhong, M. Martonosi, P. Ashar, and S. Malik: Using Configurable Computing to Accelerate Boolean Satisfiability. IEEE Trans. Computer Aided Design of Integrated Circuits and Systems, Vol. 18, no. 6 (June 1999) 861-868
8. T. Larrabee: Test Pattern generation using Boolean Satisfiability. IEEE Transactions on 2Computer Aided Design, Vol. 11, no. 1 (January 1992) 4-15

Table 2. ISCAS'85 benchmark circuits clauses specification

Miter Circuit	Unrestricted SAT formula		3-SAT formula	
	# Binary clauses	# Ternary or more clauses	# Binary clauses	# Ternary clauses
C432	678	373	850	487
C499	504	1077	614	1187
C1908	3234	978	3728	1352
C6288	9662	4862	9692	4892
C7552	12716	4299	13882	5409

The accelerator software algorithm

```
While(True){
 if(dir==forward){
   if(!Decide()) { Formula=SAT;  break; }
   process_implication();
   clause_evaluator(var);
   if(conflict=read_conflict()){
     conflict_clause_index=read-index();
     dir=backward;   //backtrack
   }
   else λ = λ +1;
 }
 if(dir==backward){
   B=Backtrack_level();
   if(B==NULL) { Formula=unSAT;  break; }
   undo(λ ,B);
   λ=B;
   if(!Tried_both_ways()){
     complement_value();
     clause_evaluator(var);
     if(conflict=read_conflict())
        conflict_clause_index=read-index();
     else{ λ = λ +1;  dir=forward; }
   }
 }
}
```