Implementation of the Advanced SAT Search Techniques in Reconfigurable Hardware

IOULIIA SKLIAROVA, ANTÓNIO B. FERRARI
Department of Electronics and Telecommunications, IEETA
University of Aveiro
3810-193 Aveiro
PORTUGAL
{iouliia, ferrari}@det.ua.pt

Abstract: - This paper presents an application-specific approach to solving the Boolean satisfiability (SAT) problem with the aid of reconfigurable hardware. In the proposed architecture an instance-specific hardware compilation is completely avoided, requiring for each problem instance just the formula information to be downloaded to an FPGA. The previously suggested method of software/reconfigurable hardware partitioning enables problems to be solved that exceed the available FPGA resources. The distinctive feature of this work consists of addressing the possibility of implementation of advanced search techniques such as nonchronological backtracking and conflict clause addition, in reconfigurable hardware.

Key-Words: - Boolean satisfiability, reconfigurable hardware, software/reconfigurable hardware partitioning

1 Introduction

During the last eight years a great deal of research effort was aimed at the implementation of efficient Boolean satisfiability (SAT) solvers on the basis of reconfigurable hardware (in FPGA, in particular). This was stimulated by the fact that current reconfigurable systems possess both the flexibility and low development costs of software and a high speed of hardware realizations. Besides, the SAT problem is very well suited to parallel implementations that take advantage of the basic capabilities of reconfigurable computing.

SAT is a very well known combinatorial problem that consists of determining whether a given Boolean formula can be satisfied by some truth assignment. The search variant of this problem requires at least one satisfying assignment to be found. Usually, the formula is presented in *conjunctive normal form* (CNF), which is composed of a conjunction of a number of clauses, where a clause is a disjunction of a number of *literals*. Each literal represents either a Boolean variable or its negation. For example, the following formula in CNF is satisfied when x_1='1', x_2='0' and x_3='-' (don't care):

$$(\bar{x}_1 \vee \bar{x}_2)(\bar{x}_2 \vee \bar{x}_3)(x_1 \vee x_2 \vee x_3)$$

The SAT problem has a lot of practical applications in a variety of engineering areas, including the testing of electronic circuits, pattern recognition, logic synthesis, etc. A good review of possible applications with the respective references can be found in [1].

It should be noted that SAT was the first problem shown to be NP-complete [2]. It means that the existing algorithms have an exponential worst-case complexity. Implementations based on reconfigurable hardware enable the primary operations of the respective algorithms to be executed in parallel. Consequently, the exponential growth of the execution time, as a function of the size of a problem instance, can be delayed.

This paper analyzes the possibility and effectiveness of implementation of advanced search techniques (that are successfully applied in the state-of-the-art software SAT solvers) in reconfigurable hardware. Section 2 that follows this introduction is devoted to the description of the algorithm employed. Section 3 gives a brief overview of the existing hardware SAT solvers emphasizing their advantages and weaknesses and formulates the primary requirements of the reconfigurable hardware SAT satisfier. The proposed architecture of FPGA-based SAT solver is presented in section 4. The conclusion is given in section 5.

2 The SAT Algorithm

Practically all the existing reconfigurable hardware SAT solvers employ some variations of the well-known Davis-Putnam algorithm [3]. The algorithm starts with an empty variable assignment. After that a *unit clause* rule, a *pure literal rule* and *decisions* are alternately applied.

A *unit clause rule* consists of finding *unit clauses*, i.e. such clauses that contain just one unassigned literal [4]. The respective variable can be assigned a value (either '1' if the literal is positive, or '0' if the literal is negative) without losing any possible solution. The selected variable is said to be *implied* to the respective value.

A *pure literal rule* is based on finding *pure literals*, i.e. such literals that are either all positive or all negative. The variable corresponding to a pure literal can be assigned a value (either '1' if all the occurrences of the literal are positive, or '0' if all the occurrences of the literal are negative) without influencing in any way the satisfiability of the formula.

These two rules are known as *reduction methods* because they allow an initial formula to be simplified without affecting the satisfiability of the formula.

When there are no more unit clauses and pure literals, a *decision* is taken. The *decision* consists of choosing one unassigned variable and assigning a value to it. There exist two basic approaches to the selection of the decision variables: *static* and *dynamic*. In the *static* approach all the variables are initially pre-ordered using some criteria. The resulting static sequence is used to fetch the next decision variable when required. In the *dynamic* approach such a variable and a value are chosen that are most likely to help in satisfying the formula (a variety of heuristic methods are employed for this purpose).

When a variable is assigned a value (either by means of a decision or a reduction), all the satisfied clauses together with the falsified literals are removed from the formula.

A *conflict* appears if either a variable is implied to two opposite values or there exists an empty clause. In this case it is necessary to erase all actions performed after the last decision and invert the value of the current decision variable. If both possible values have already been tried, the algorithm backtracks to the previous decision variable.

When the last clause becomes satisfied and is deleted from the formula, it means that the current variable assignment represents a solution. If all possible assignments of values to variables have been implicitly tested (i.e. both values of the first decision variable were tried out without success), thus the formula is unsatisfiable.

In the DP algorithm, the search process is usually represented with the aid of a *decision tree*, whose nodes correspond to intermediate sub-formulae obtained during the search, and edges represent the decisions taken. The decision tree is traversed using the depth-first-search approach.

In the present-day software SAT solvers a lot of advanced techniques are applied that enable those regions of the search space that do not contain any solution to be discovered and their exploration to be avoided. Let us describe some of these techniques that we are going to implement in reconfigurable hardware.

In the DP algorithm, in case of a conflict, the control process backtracks always to the most recently assigned decision variable. Such backtracking is called *chronological* [5]. However, an analysis of the conflict causes may lead to the discovery of the decision variables that are really responsible for the conflict occurrence. Thus, the algorithm can backtrack directly to the most recent of these decision variables enabling in this way some branches of the decision tree to be pruned. The process is usually referred to as *nonchronological backtracking* [5].

Let us illustrate this feature with the aid of the following formula:

$$\begin{aligned}&(\bar{x}_0 \vee \bar{x}_1)(\bar{x}_1 \vee \bar{x}_2)(x_0 \vee x_1 \vee x_2)(x_1 \vee x_6) \wedge \\ &\wedge (x_1 \vee x_7)(x_1 \vee x_8)(\bar{x}_3 \vee \bar{x}_4)(\bar{x}_4 \vee \bar{x}_5) \wedge \\ &\wedge (x_3 \vee x_4 \vee x_5)(x_2 \vee x_{12})(x_4 \vee x_7) \wedge \\ &\wedge (x_4 \vee x_6)(x_4 \vee x_8)(\bar{x}_{10} \vee \bar{x}_{11})(\bar{x}_{10} \vee x_{11}) \wedge \\ &\wedge (x_{10} \vee \bar{x}_{11})(\bar{x}_1 \vee x_{10} \vee x_{11})\end{aligned} \quad (1)$$

In order to find out a solution we could apply an algorithm that includes application of the unit clause rule and employs dynamic selection of the next decision variable based on a *maximum-occurrence-in-clauses* heuristics. This heuristics, at each step tries to satisfy as many clauses as possible by choosing such a variable that occurs in the maximum number of clauses. If the majority of the respective literals are positive, the variable is assigned a value '1', in the opposite case the variable is set to '0'.

The resulting decision tree for formula (1) is presented in fig. 1a). As it can be seen the algorithm requires 12 nodes to be visited.

The technique of nonchronological backtracking for SAT was proposed in GRASP [5]. It relies on the construction of a directed implication graph that represents the sequence of implications generated during the search. When a conflict arises, the implication graph is analyzed to determine those variable assignments that are directly responsible for the conflict. This requires a *conflict-induced* clause to be constructed. After that, instead of performing chronological backtracking, the algorithm may jump directly to the most recently assigned decision variable that appears in the conflict-induced clause. As a result, some branches of the decision tree may

be pruned away, thus saving the time that would otherwise be needed to explore them. Applying the technique to the formula (1) will lead to the construction of the decision tree depicted in fig. 1b). In this case only 9 nodes must be visited.

Additionally, the conflict-induced clauses may be recorded, allowing the occurrence of similar conflicts to be prevented later on during the search. This process is referred to as *dynamic clause addition*.

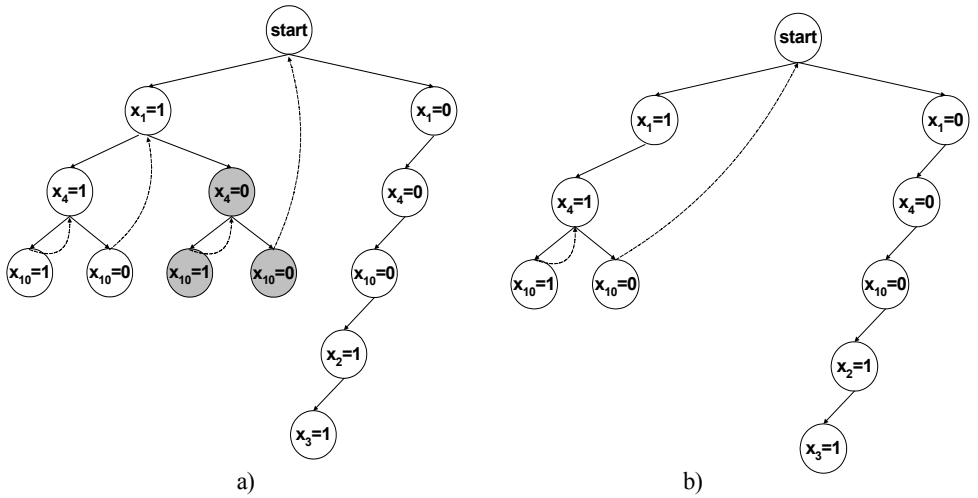

Fig. 1. The decision tree constructed for formula (1) applying chronological (a) and nonchronological (b) backtracking

3 The Primary Requirements of the SAT Solver

In this section we will formulate the primary requirements of the reconfigurable hardware SAT solver taking into account the previous work done in this domain [6-13].

Initially, all the proposed hardware SAT solvers were based on the instance-specific approach [6-9]. In this case a specific hardware circuit is compiled and downloaded to FPGA for each problem instance to be solved. However, the compilation time may constitute a large portion of the total solving time. For easy problem instances it even dominates and cancels out all the benefits of fast hardware execution. Therefore, a lot of work in this direction was done to enable the generation of hardware configurations to be accelerated [9]. The more recent investigation [10, 11, 12, 13] is tailored to either partial or complete avoiding of instance-specific hardware compilation.

In this work we will follow the policy proposed in [12] where a dynamically reconfigurable architecture is described enabling both the FPGA configuration time to be reduced and the hardware compilation overhead to be eliminated. Consequently, the SAT satisfier should have such architecture that is not required to be modified from problem instance to problem instance. In this case, the respective circuit would be downloaded to the FPGA and after that it could be employed to solve a series of problem instances. For each Boolean formula to be considered, only the formula-specific information should be communicated to the FPGA.

Of course such a circuit could not be used for processing an arbitrary problem instance because the available hardware resources are always limited. Thus, the SAT solver should address and efficiently overcome this limitation. For such a purpose the following four approaches are usually applied.

The first one requires the original formula to be decomposed into a series of sub-formulae, each of which respects the capacity constraints [9]. The resulting sub-formulae can be handled either sequentially or in parallel by employing several unconnected FPGAs. The main limitation of this technique is that decomposing some formulae may be very inefficient, therefore increasing the computation time to unacceptable levels [10].

The second approach consists of partitioning the circuit into multiple interconnected FPGAs [7]. The primary limitation of this method is that it requires signals to propagate off-chip, resulting in large delays. Besides, fast and efficient multi-FPGA partitioning and routing is quite a difficult task.

The third method is based on a virtual hardware scheme proposed in [10], which relies on dividing the circuit into a series of hardware pages that are

successively run. Since all the hardware pages have the same structure with only a number of registers being reconfigured, page switching is performed very fast. The computed variable assignment is stored in external memory blocks and is processed when traversing the FPGA from one memory block to another

The last approach that is currently intensively investigated is based on partitioning the problem solution between software and reconfigurable hardware. Within the domain of software/reconfigurable hardware partitioning two different techniques are applied. The first one [10] requires the most computationally intensive tasks (namely, computing implications and selecting dynamically the next decision variable) to be assigned to reconfigurable hardware, while the control-oriented tasks that are inherently sequential (in particular, the conflict analysis) are realized in software. It is difficult to estimate the efficiency of this approach because the published results [10] are based on simulation in software and do not account the time that would be spent in communication between the host processor and an FPGA.

The second partitioning technique was proposed in [14]. In this case an FPGA is only responsible for processing sub-problems that appear at various levels of the decision tree and satisfy the imposed hardware constraints (such as the maximum allowed number of variables and clauses in a sub-formula). Those sub-formulae that do not respect the constraints are initially processed by a software program that implements the same algorithm as the FPGA circuit. In this work we will follow the same method of partitioning.

The remaining set of requirements is related to the algorithm itself. First of all, the dynamic selection of the next decision variable should be performed because it has a significant impact on the convergence of the search. The majority of hardware SAT satisfiers employ just a static selection mechanism. Second, a conflict analysis engine should be implemented in hardware. Practically all FPGA-based SAT solvers lack this feature. An attempt to realize a conflict analysis in hardware was performed in [7]. Our approach is similar to that proposed in [7].

The following list summarizes the formulated requirements:

1. The circuit should be application-specific (as opposed to instance-specific) eliminating in this way the hardware compilation step.
2. The problem has to be partitioned between software and reconfigurable hardware, what allows the limited FPGA capacity problem to be alleviated.
3. The next decision variable has to be selected dynamically.
4. The conflict analysis should be implemented both in software and in FPGA.

It should be noted that a SAT solver architecture that satisfies the first three points, was proposed in [12, 13]. The main objective of this work is to incorporate the hardware support for nonchronological backtracking and dynamic clause addition.

4 SAT Solver Architecture

As it was mentioned in section 3, the architecture we are going to propose is based on the work described in [12, 13]. Thus, the principle design decisions made in [12, 13] should be first outlined.

The SAT problem containing m clauses and n variables is formulated over a ternary matrix with dimensions $m \times n$. Consequently, the required hardware resources depend only on the size of a formula and do not depend on its complexity. It should be underlined that the suggested design does not require an initial formula to be transformed to k-SAT problem, as it is demanded by some other proposed architectures [6, 10]. In order to find a satisfying assignment, we need to discover a ternary vector, which is orthogonal to every row of the matrix. If such vector cannot be found, the formula is unsatisfiable.

The basic components of the proposed architecture are shown in fig. 2.

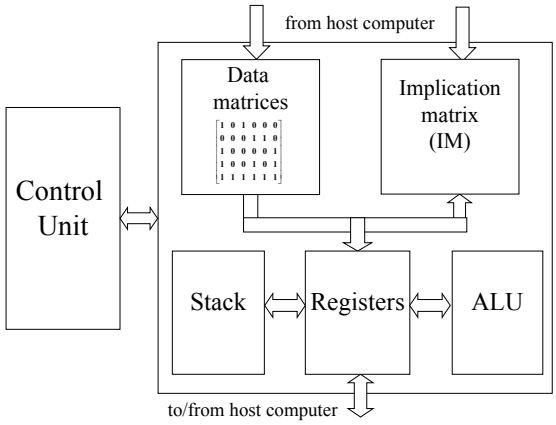

Fig. 2. Architecture of SAT solver

The central control unit executes the required algorithm. The block *Registers* consists of components that store such kind of data as addresses

of the currently active row and column, the deleted rows, etc. The matrix is implemented with the aid of two memory blocks corresponding to the original matrix and to its transpose. Therefore, any row and column can be read in just one clock cycle. During the search process some rows and columns of the matrix have to be deleted. However, the matrix itself is not modified and all possible changes are reflected in the respective registers. The ALU is used to perform the primary computations over different ternary vectors that include calculation of the number of ones and zeros, checking two vectors for orthogonality, etc. The stack memory supports the backtracking process. When a decision is performed, the current values from all the registers are stored in the stack and these values are restored during the backtracking process, i.e. they are moved from the stack back to the registers.

Let us now describe the control algorithm we are going to implement (see fig. 3). There are two primary reduction methods employed. They are the unit clause rule and the pure literal rule explained in section 2.

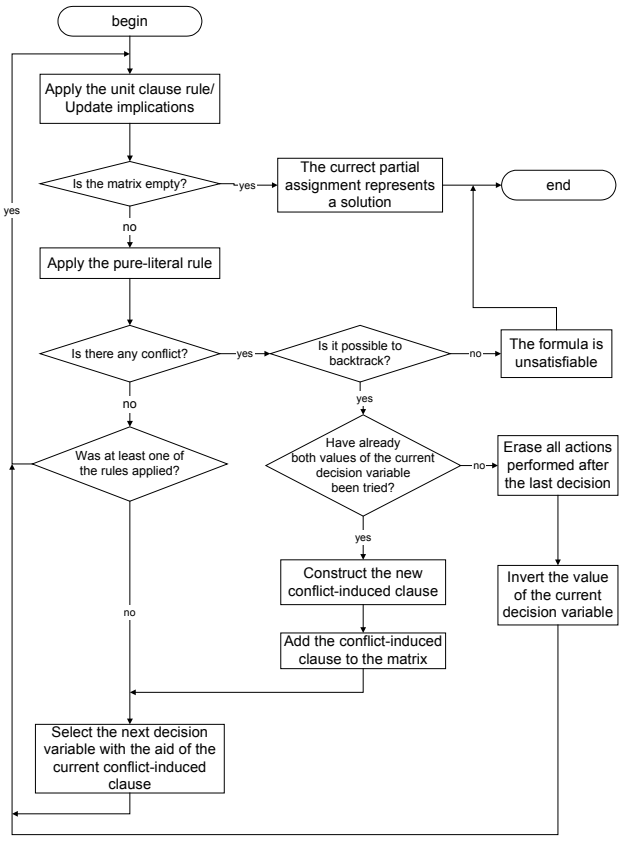

Fig. 3. The implemented algorithm

In order to realize nonchronological backtracking a conflict-induced clause should be constructed. This clause is held in an n-bit register. In our design, the conflict-induced clause is generated when both values of some decision variable were tried out without success. To construct a conflict-induced clause, we should keep track of the sequence of implications produced during the search process. This sequence is maintained in an $n \times n$ DPRAM block referred to as *Implication matrix* (IM) in fig. 2.

When the unit clause rule is applied, the implication matrix needs to be updated. If a variable x_j, $j=1,\ldots,n$, is implied because of a unit clause c_i, $i=1,\ldots,m$, then a value at the address j of IM has to be modified. The new value must reflect the complete sequence of implications that finally caused x_j to be implied.

If an initial clause c_i is equal to $(l_j \vee l_k \vee \ldots \vee l_r)$, where l_j, $l_k,\ldots,l_r$ represent some literals, $1 \le (j,k,\ldots,r) \le n$, then the following value should be written at the address j of IM: $\mathrm{IM}[j]=c_i^B \vee \mathrm{IM}[k] \vee \ldots \vee \mathrm{IM}[r]$, where c_i^B is an n-bit Boolean vector that has *1*s in the positions corresponding to the literals occurring in c_i, and has *0*s in all the remaining positions.

During backtracking, we should erase all the actions performed after the last decision. These actions include:

- restoring recently deleted rows and columns;
- recovering the previous value of a ternary vector that is being looked for;
- undoing the recent sequence of implications.

The first two points are very easy to implement. For such purposes the previous values of the respective registers have to be restored from the stack memory. To implement the last point, the contents of IM should be updated in such a way that the current decision variable is removed from the sequence of implications. To accomplish this task the following operation must be executed for $j=1,\ldots,n$: $\mathrm{IM}[j] = \mathrm{IM}[j] \wedge \overline{cur\_des\_var}$, where *cur_des_var* is a binary decoded value of a $\log_2 n$-bit register that stores the number of the current decision variable. Of course this operation is quite expensive, that is why it is better to perform it with the aid of an n-bit register, which can mark by *1*s the deleted columns of IM.

To construct a conflict-induced clause, we need to perform a disjunction of those rows of IM that correspond to the literals occurring in a clause that is the cause of a conflict.

When the backtracking process is activated, only those variables should be selected, that appear in the current conflict-induced clause, because unless a value of one of these variables is inverted, the same kind of conflict will occur later on during the search.

The actual dimensions of the initial matrix are stored in two special registers. If these dimensions (the number of rows, in particular) are smaller than the maximum allowed, then the generated conflict-induced clauses can be added to the matrix to enable the subsequent occurrence of similar conflicts to be avoided.

5 Conclusion

This paper analyzes the possibility of implementing nonchronological backtracking and dynamic clause addition in reconfigurable hardware. The basic architecture of the respective SAT solver is proposed. It should be noted that a lot of work is to be done to draw a more deterministic conclusion. In particular, we are going to implement a small prototype of the SAT satisfier. It will permit to measure the total execution time including the communication between the host processor and the FPGA. The implementation will be based on the Xilinx XC2VP7 FPGA [15] installed on an ADM-XPL PCI board.

The design described in [12, 13] demonstrated very good results for some classes of difficult problems from the DIMACS benchmark suite [16]. Therefore we expect that implementing the proposed architecture will lead to further performance improvements.

Acknowledgment

This work was supported by the Portuguese Foundation of Science and Technology under grants No. FCT-PRAXIS XXI/BD/21353/99 and No. POSI/43140/CHS/2001.

References:

[1] J. Gu, Satisfability Problems in VLSI Engineering, *DIMACS Workshop on Satisfability Problem*, Mar. 1996.

[2] S.A. Cook, The complexity of theorem-proving procedures, *Proc. 3rd ACM Symp. on Theory of Computing*, 1971, pp. 151-158.

[3] M. Davis, G. Logemann, and D. Loveland, A machine program for theorem proving, *Communications of the ACM*, 5:394-397, 1962.

[4] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, Algorithms for the Satisfiability (SAT) Problem: A Survey, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 35, pp. 19-151, 1997.

[5] J.M. Silva, and K.A. Sakallah, GRASP: a search algorithm for propositional satisfiability, *IEEE Trans. Computers*, pp. 506-521, Vol. 48, No. 5, May 1999.

[6] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, Solving satisfiability problems using reconfigurable computing, *IEEE Trans. VLSI Systems*, Vol. 9, No. 1, pp. 109-116, Feb. 2001.

[7] P. Zhong, P. Ashar, S. Malik, and M. Martonosi, Using reconfigurable computing techniques to accelerate problems in the CAD domain: a case study with Boolean satisfiability, *Proc. of the Design Automation Conf. – DAC*, 1998, pp. 194-199.

[8] O. Mencer and M. Platzner, Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment, *Proc. 32nd Hawaii Int. Conf. on System Sciences* - HICSS-32 (Configware - Reconfigurable Engineering track), Island of Maui, 1999.

[9] M. Abramovici and J.T. de Sousa, A SAT solver using reconfigurable hardware and virtual logic, *Journal of Automated Reasoning*, Vol. 24, Nos. 1-2, pp. 5-36, Feb. 2000.

[10] J. de Sousa, J.P. Marques-Silva, and M. Abramovici, A configware/software approach to SAT solving, *9th IEEE Proc. Int. Symp. on Field-Programmable Custom Computing Machines, FCCM*, 2001.

[11] M. Boyd and T. Larrabee, ELVIS – a scalable, loadable custom programmable logic device for solving Boolean satisfiability problems, *Proc. 8th IEEE Int. Symp. on Field-Programmable Custom Computing Machines - FCCM*, 2000.

[12] I. Skliarova and A.B. Ferrari, A SAT Solver Using Software and Reconfigurable Hardware, *Proc. of the Design, Automation and Test in Europe Conference – DATE'2002*, 2002, p. 1094.

[13] I. Skliarova and A.B. Ferrari, A hardware/software approach to accelerate Boolean satisfiability, *Proc. of IEEE Int. Workshop on Design and Diagnostics of Electronic Circuits and Systems – IEEE DDECS'2002*, 2002, pp. 270-277.

[14] I.Skliarova, A.B.Ferrari, Design and Implementation of Reconfigurable Processor for Problems of Combinatorial Computations, *Proc. of the Euromicro Symp. on Digital System Design – DSD'2001*, 2001, pp.112-119.

[15] Xilinx FPGA. [Online]. Available: http://www.xilinx.com/.

[16] DIMACS challenge benchmarks. [Online]. Available: http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html.