

Reconfigurable Hardware SAT Solvers: A Survey of Systems

Iouliia Skliarova and António de Brito Ferrari, *Member, IEEE*

Abstract—By adapting to computations that are not so well-supported by general-purpose processors, reconfigurable systems achieve significant increases in performance. Such computational systems use high-capacity programmable logic devices and are based on processing units customized to the requirements of a particular application. A great deal of the research effort in this area is aimed at accelerating the solution of combinatorial optimization problems. Special attention in this context was given to the Boolean satisfiability (SAT) problem resulting in a considerable number of different architectures being proposed. This paper presents the state-of-the-art in reconfigurable hardware SAT solvers. The analysis and classification of existing systems has been performed according to such criteria as algorithmic issues, reconfiguration modes, the execution model, the programming model, logic capacity, and performance.

Index Terms—Boolean satisfiability, reconfigurable computing, FPGA, hardware acceleration.

1 INTRODUCTION

ALTHOUGH the concept of reconfigurable computing was proposed in the early 1960s [1], it is only recently that technologies that allow it to be put into practice became available. The interest started at the beginning of the 1990s as FPGA densities broke the 10K logic gate barrier [2]. Since then, reconfigurable computing has become a subject of intensive research. For some classes of applications, reconfigurable systems allow very good performance to be achieved compared to general-purpose computers. Other types of applications were mapped to reconfigurable hardware because it offers innovative opportunities to explore. According to the primary objective to be achieved, all these applications can be broadly divided into three categories: hardware emulation and rapid prototyping, evolvable hardware, and the acceleration of computationally intensive tasks. The last category is, without doubt, the prevalent one.

A common characteristic of the considered computationally intensive tasks is that they are very well-suited to parallel implementations that take advantage of the basic capabilities of reconfigurable computing. Cryptography, signal and image processing are good examples in this category. These applications are characterized by large amounts of data to process and by inherent parallelism and are suitable for pipelining [3]. All these factors contribute to an increase in the performance of a reconfigurable system compared to a similar implementation on general-purpose computers [4].

Recently, a series of attempts has been made to accelerate applications that involve rather complex control flow. In this context, special attention was given to problems in the area of combinatorial optimization. Among them, the Boolean satisfiability (SAT) problem stands out. This may

be partially explained by the extremely wide range of practical applications in a variety of engineering areas, including the testing of electronic circuits, pattern recognition, logic synthesis, etc. [5]. In addition, SAT has the honor of being the first problem shown to be NP-complete [6]. This means that 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 effect of exponential growth in the computation time can be delayed, thus allowing larger size instances of SAT to be solved [5]. Besides, finding an efficient way to solve the SAT problem with the aid of reconfigurable hardware may help to accelerate the solution of other combinatorial optimization problems.

In this paper, we present the current status of reconfigurable hardware SAT solvers and give an overview of the existing approaches and their trade offs. The remaining part of the paper is organized as follows: Section 2 is devoted to the definition of the problem and the description of the algorithms that are usually employed by hardware SAT solvers. Section 3 presents the most well-known architectures of reconfigurable hardware SAT solvers. Analysis and classification of these architectures according to different criteria is performed in Section 4. Finally, concluding remarks are given in Section 5.

2 PROBLEM DEFINITION

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 $x_1 = "0"$, $x_2 = "1"$, and $x_3 = "0"$:

• The authors are with the Department of Electronics and Telecommunications, IEETA, University of Aveiro, 3810-193 Aveiro, Portugal.
E-mail: {iouliia, ferrari}@det.ua.pt.

Manuscript received 3 Dec. 2003; revised 10 Apr. 2004; accepted 6 May 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0267-1203.

```

procedure DP
input: a CNF formula  $f$ , partial variable assignment  $T$  (initially empty)
output: a satisfying variable assignment (if it exists) or false (if it does not exist)
begin
  deduct() - compute implications, apply the pure literal rule
  if () in  $f$  then -  $f$  contains an empty clause  $\Rightarrow$  conflict
    return false -  $f$  is unsatisfiable
  if  $f = \emptyset$  then -  $f$  is empty  $\Rightarrow$  all clauses have been satisfied
    exit with  $T$  - a satisfying variable assignment was found
   $p = \text{decide}()$  - select the next decision variable and its value
  DP ( $f \cup \{p\}$ ,  $T \cup \{p\}$ )
  DP ( $f \cup \{\bar{p}\}$ ,  $T \cup \{\bar{p}\}$ )
  return false -  $f$  is unsatisfiable
end

```

Fig. 1. Pseudocode of the DP algorithm.

$$(\bar{x}_1 \vee x_3)(\bar{x}_2 \vee \bar{x}_3)(x_1 \vee x_2 \vee x_3)(x_1 \vee \bar{x}_3)(\bar{x}_1 \vee x_2). \quad (1)$$

Algorithms for solving the SAT problem can be divided in two major groups: *complete* and *incomplete* [5]. The majority of *incomplete* algorithms are able to find a solution in favorable cases, but either do not terminate or get stuck in other cases. In situations where a solution is not found, it is impossible to determine whether the formula is unsatisfiable or the algorithm did not explore the search space sufficiently. Consequently, the incomplete algorithms do not always verify satisfiability and cannot prove unsatisfiability. Nevertheless, these methods are of particular interest for problem instances that are either underconstrained (having many solutions) or so difficult that a complete algorithm cannot solve them in a reasonable time. Besides, incomplete algorithms can be used to solve the maximum satisfiability problem (whose objective is to maximize the number of satisfied clauses).

Complete algorithms are always able either to find a solution (although it may take an unacceptable time) or to conclude that a formula is unsatisfiable, i.e., such methods can verify either the satisfiability or the unsatisfiability of a given problem instance.

2.1 Complete Algorithms

Among complete algorithms that are usually employed by hardware SAT solvers, the most well-known are the Davis-Putnam (DP) algorithm (in Loveland form) [7] and the PODEM (*Path-Oriented DEcision Making*) algorithm [8]. Since the former method is much more popular, we will describe it in more detail.

In the DP algorithm, the search process is organized by implicitly traversing the space of all possible assignments of values to variables (the pseudocode of the algorithm is presented in Fig. 1). The process starts with an empty variable assignment. Then, a *unit clause* rule, a *pure literal rule*, and *decisions* are applied alternately.

A *unit clause rule* consists of finding *unit clauses*, i.e., clauses that contain just one unassigned literal [5]. 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., 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.

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 (this variable is referred to as a *decision variable*). There are two basic approaches to the selection of the decision variables: *static* and *dynamic*. In the static approach, all the variables are initially preordered using some criteria. The resulting static sequence is used to fetch the next decision variable when required. In the dynamic approach, 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), then the formula is unsatisfiable.

In the DP algorithm, the search process is usually represented by a *decision tree*, whose nodes are associated with intermediate subformulae obtained during the search and edges correspond to the decisions taken. An example of a decision tree constructed for (1) is given in Fig. 2. The root of the decision tree corresponds to a start point, where all the variables are unassigned. If, at any node, a partial variable assignment satisfies a formula, then the search process is terminated. In the opposite case, the search must proceed either forward (if there are no conflicts) or backward (if a conflict has appeared). The decision tree is traversed using the depth-first-search approach.

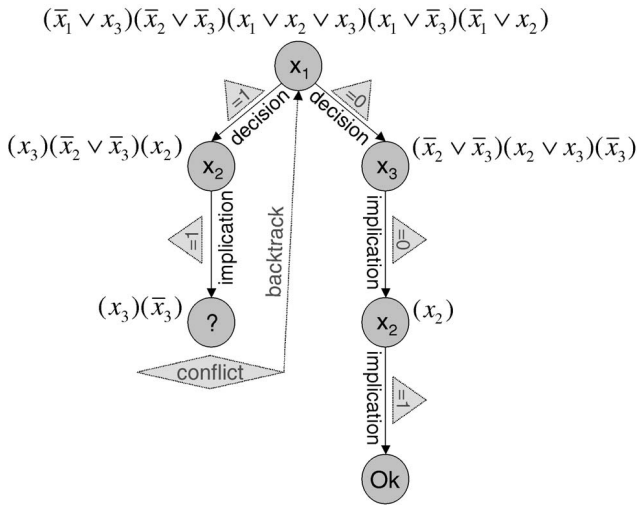


Fig. 2. The decision tree constructed for x_3 satisfying (1) with the aid of the DP algorithm.

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. For instance, in the case of a conflict in the algorithm described, the control process always backtracks to the most recently assigned decision variable (such backtracking is called *chronological* [9]). 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* (also known as *intelligent backtracking*) [9].

The technique of nonchronological backtracking for SAT was proposed in GRASP (*Generic seaRch Algorithm for the Satisfiability Problem*) [9]. 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, the algorithm may jump directly to the most recently assigned decision variable that appears in the conflict-induced clause. 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*.

2.2 Incomplete Algorithms

A number of incomplete algorithms have been developed that allow for a solution of the SAT problem, including local search methods [5], genetic algorithms [10], etc. For instance, local search algorithms can be applied to SAT by introducing an objective function that counts the number of satisfied clauses and solving to maximize the value of this function. Such methods are easily applicable since, given a feasible solution, little effort is needed to both generate a new solution and test whether there is any improvement in the objective function. The major weakness of local search is that the algorithms have a tendency to get stuck at a local maximum.

Among local search algorithms for SAT, perhaps the most widely known are GSAT (greedy local search) [11] and WSAT (GSAT with random walk) [12]. An outline of both methods is presented in Fig. 3. The algorithms start by randomly generating a feasible solution. Then, a loop is executed in which the GSAT algorithm iteratively examines the neighborhood of the current solution and selects a new feasible solution that gives the greatest increase in the number of satisfied clauses, whereas the WSAT algorithm randomly selects a variable in an unsatisfied clause and inverts its value. Both algorithms either return a satisfying assignment or give the answer “no solution found” (the latter does not mean that a formula is unsatisfiable). The parameters *MAX_TRIES* (the number of new search sequences) and *MAX_FLIPS* (the number of variable values flips per try) are used to control the maximum runtime of the algorithms.

Incomplete algorithms are well-suited to a reconfigurable hardware implementation because they do not require complex control structures and have an inherent parallelism (exhibited, in particular, during the clause evaluation stage). However, as has already been mentioned, fast

<pre> procedure GSAT input: a CNF formula f, MAX_FLIPS and MAX_TRIES output: a satisfying variable assignment (if found) begin for i in 1 to MAX_TRIES T = randomly generated variable assignment for j in 1 to MAX_FLIPS if T satisfies f then return T p = a variable such that a change in its value gives the largest increase in the number of clauses of f that are satisfied by T T = T with the truth assignment of p reversed end for end for return "no satisfying assignment found" end </pre>	<pre> procedure WSAT input: a CNF formula f, MAX_FLIPS and MAX_TRIES output: a satisfying variable assignment (if found) begin for i in 1 to MAX_TRIES T = randomly generated variable assignment for j in 1 to MAX_FLIPS if T satisfies f then return T c = a random unsatisfied clause p = a random variable in c T = T with the truth assignment of p reversed end for end for return "no satisfying assignment found" end </pre>
---	--

Fig. 3. Pseudocode of GSAT and WSAT algorithms.

incomplete algorithms are primarily intended for processing large CNF formulae, for which complete algorithms may not be applicable. An effective representation, storage and evaluation in reconfigurable hardware of a significant number of clauses is a challenging problem (as will be shown in Section 4.5). Therefore, the implementation of incomplete algorithms in field-programmable devices currently seems to have a limited application because all the potential acceleration that could be gained would be offset by the communications overhead and high memory requirements.

3 ARCHITECTURES OF SAT SOLVERS

Recently, several research groups have explored different approaches to solve the SAT problem with the aid of reconfigurable hardware [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38]. In this section, the most widely known and successful architectures will be described. Since names have not typically been given to hardware SAT solvers, we will refer to them according to the first author's names of the respective publications.

3.1 Suyama et al.

Suyama et al. [13], [14], [15], [16] suggested an architecture of an *instance-specific* SAT solver capable of finding all the solutions (or a fixed number of them) of a given problem instance. The algorithm employed is characterized by the fact that, at any moment, a full variable assignment is evaluated. In order to choose the next decision variable, static selection was applied in [13] and two dynamic techniques have been tried in [14], [15], [16]. The first of the dynamic approaches is based on experimental unit propagation, which requires both possible values to be assigned experimentally to each free variable. A variable and a value are then selected that cause the maximum number of implications to be generated. The second approach relies on a *maximum-occurrence-in-clauses-of-minimum-size* heuristics selecting a variable that appears in the maximum number of binary clauses [14].

The high-level architecture of the SAT solver [14], [15], [16] is shown in Fig. 4. The topmost block stores the value assigned to each variable either by means of implication or decision and indicates the level of the search tree at which a variable was assigned. The values of all the variables are fed to the respective clauses, which in turn are evaluated concurrently. Then, the results of evaluating every clause are combined and, depending on the final result, either branching or backtracking is performed.

Each CNF formula has first to be converted to 3-SAT format (in which the number of literals per clause is equal to 3) by introducing auxiliary variables. Then, a specially developed C program analyzes the resulting formula and generates the respective behavioral HDL description. On the basis of this HDL code, an instance-specific circuit is synthesized and implemented with the aid of commercially available tools. Using this strategy, a number of circuits have been implemented on an Altera FLEX10K250 FPGA clocked at 10 MHz. Suyama et al. were able to achieve speedups of 1-10 times compared to the POSIT (*Propositional Satisfiability Testbed*) algorithm [39] executed on an UltraSPARC-II/296 MHz over some instances from the DIMACS (*Center for*

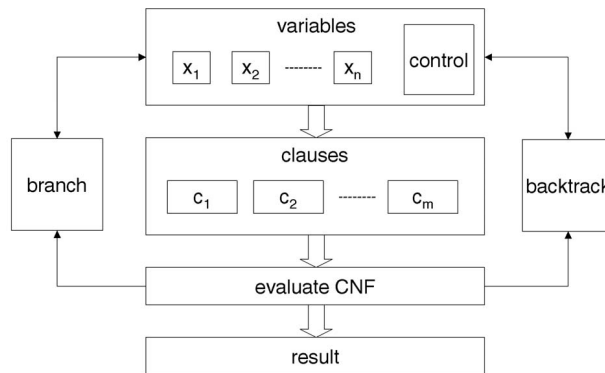


Fig. 4. The high-level view of the SAT solver proposed by Suyama et al. in [14].

Discrete Mathematics & Theoretical Computer Science) benchmark suite [40]. However, the time spent in hardware compilation and configuration (about one hour) was not taken into account. When a CNF formula cannot be accommodated in one FPGA, it was proposed to divide the circuit into multiple FPGA chips. However, because of high interchip communication requirements, the resulting logic utilization was very low (about 13 percent) [14], [15].

3.2 Zhong et al.

Zhong et al. implemented an instance-specific SAT solver based on the DP algorithm [7]. In their early work [17], they constructed an implication circuit and a finite state machine (FSM) for each variable in the formula, all the state machines being connected in a serial chain (see Fig. 5a). In each period of time, only one FSM is active. As soon as this state machine completes processing, it transfers control either to the next FSM (forward search) or to the previous one (backtracking). Each state machine knows the current value of its variable (that can be either "0," "1," or free) and is aware of whether that value has been assigned or implied. The solution is found if the last (right) state machine tries to activate the next state machine. If the first (left) state machine tries to pass control to the left, then the solution does not exist. As a preprocessing step, all the variables are sorted taking into account the number of their appearances in a given formula. This static order is used to arrange the variables in the serial chain.

The main drawbacks of the design suggested in [17] are a low clock frequency (ranging from 700 KHz to 2 MHz for different formulae) and a very long hardware compilation time (up to several hours on a Sun 5/110MHz/64MB). With the aim to improve performance, hardware implementation of *nonchronological backtracking* was proposed in [18]. However, this did not lead to significant improvements and therefore motivated the revision of the initial design decisions. As a result, a regular ring-based interconnecting structure was employed instead of irregular global lines, essentially reducing the compilation time to the order of seconds and increasing the clock rate (to 20-30 MHz) [19], [20]. In addition, a technique enabling conflict clauses to be generated and added was proposed. Differently from [18], the design described in [19], [20] is clause-oriented, with the clauses (all of the same size) arranged in processing elements distributed along a pipelined communication network (see Fig. 5b). The main control unit keeps track

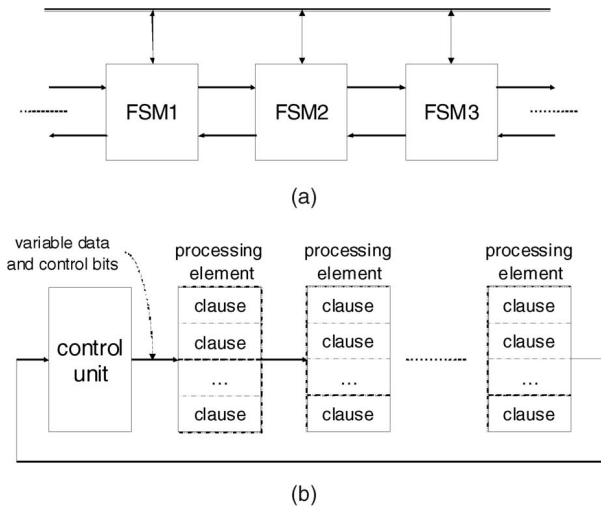


Fig. 5. Zhong et al. architectures: (a) Each FSM corresponds to a variable in a CNF formula in the satisfier proposed in [17]. (b) In architecture [19], [20], all the variables pass through the pipeline of processing elements, each containing a number of clause modules.

of the current state of the SAT solver and monitors the network for both variable's value changes and conflicts.

To address large problem instances that do not fit into one FPGA chip, the authors propose to employ several interlinked FPGAs (the architectures [19], [20] are easily scalable). The experimental results are based both on hardware implementation (on an IKOS emulator containing a number of FPGA array boards) and on simulation. The speedups achieved over the software satisfier GRASP [9] executing on a Sun5/110MHz/64MB (in a restricted mode), including the hardware compilation and configuration time, are of an order of magnitude [20] for a subset of the DIMACS SAT benchmarks [40]. It should be noted that the architectures proposed by Zhong et al. are among the most widely known and served as prototypes for a number of reconfigurable hardware SAT solvers that were developed subsequently.

3.3 Platzner et al.

The SAT solver proposed by Platzner et al. [21], [22] is similar to that of Zhong et al. [17]. It consists of a column of finite state machines, deduction logic, and a global control unit (see Fig. 6) and implements a DP-based algorithm for CNF formulae. The deduction logic computes the result of the formula from the current partial variable assignment. All variable assignments are tried in a fixed order. Initially, all variables are unassigned and the control unit activates the first FSM. This FSM tries to assign "0" to its variable and the deduction logic calculates the result. If the formula evaluates to "1," the solution has been found. Otherwise, if the formula evaluates to "0," the FSM complements its value. If the formula evaluates to "x," the next FSM is activated. If an FSM tries both variable assignments and the formula always evaluates to "0," the FSM resets its value and passes control to the previous FSM.

The authors implemented an accelerator prototype on the base of a Pamette board containing four Xilinx XC4028 FPGAs. The speedups obtained for *hole6...hole10* SAT benchmarks from DIMACS [40], including hardware compilation and configuration time, range from 0.003 to

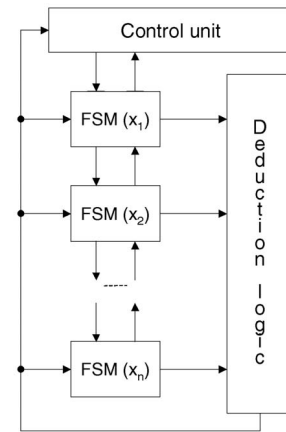


Fig. 6. The top-level view of the architecture proposed by Platzner et al. [21], [22].

7.408 compared to GRASP executing on a PII/300MHz/128MB [21]. The clock frequency in [21] varies from 65 MHz (for *hole6*) to 27 MHz (for *hole10*). The hardware compilation time dominates the hardware runtime for all the problem instances considered and, consequently, constitutes the principal limitation of this architecture. Besides, no solution was proposed for a case when a given formula does not fit the chosen FPGA.

3.4 Abramovici et al.

Abramovici and Saab [23] applied a technique of modeling a given Boolean formula (not necessarily in CNF format) by an arbitrary circuit. The designed SAT solver is based on the PODEM algorithm [8] that is generally used to solve test generation problems. Here, the goal is to set the primary output of a combinational logic circuit (which represents a Boolean function to be satisfied) to "1" by finding a suitable assignment of the primary inputs. An important concept of this algorithm is an objective, which is a desired assignment of some value to a signal initially having an unknown value [23]. An objective can only be achieved by primary input assignments. A backtrace procedure is used to propagate an objective along a path to primary inputs and determines the primary input assignment that is likely to help to achieve the objective. To accomplish this goal, two models of the circuit have been constructed: a forward model for propagating primary input assignments and a backward model for propagating objectives (see Fig. 7a).

In [24], an improved architecture (shown in Fig. 7b) is suggested that employs the DP algorithm for CNF formulae (modeled by a two-level circuit) and implements an enhanced variable selection strategy. The variable logic block stores the current values of all the variables. These values are sent to the literal logic block, which distributes them between clauses. The clause logic block evaluates every clause and the entire formula and also determines the desired values for literals (objectives). The objectives are sent back to the literal logic block, which combines the objectives arriving from different clauses into one objective for each variable. Finally, the variable logic block transforms the objectives into variable assignments (either implications or decisions). The control unit initiates backtracking when the formula evaluates to "0." It is important that all the objectives propagate concurrently along all possible paths, enabling multiple variables to be assigned in parallel.

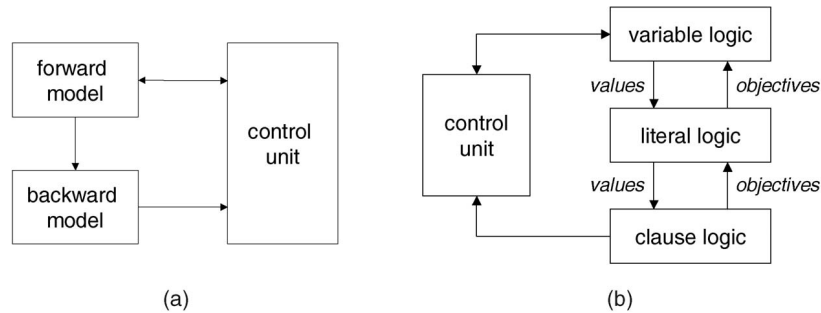


Fig. 7. Abramovici et al. architectures: (a) For FPGA-based implementation of the PODEM algorithm, two models of the circuit have been constructed in [23]. (b) The top-level view of the satisfier [24].

For hardware implementation Abramovici et al. suggest creating a library of basic modules that have predefined internal placement and routing and are to be used for any formula. The solver circuit is built from modules, reducing the compilation time to the order of minutes. The authors implemented simple circuits on the XC6264 FPGA and simulated the bigger ones. For a circuit occupying the whole area of the XC6264 FPGA, the clock frequency was about 3.5 MHz. In [24], Abramovici and de Sousa report raw speedups (not including the compilation and configuration overheads) from 0.01 to 7000 (after time unit adjustment) achieved over GRASP [9] for a subset of DIMACS SAT benchmarks [40].

In [24], a virtual logic system was proposed allowing circuits to be constructed for solving SAT problem instances that are larger than the available hardware resources. This is achieved by decomposing a formula into independent subformulae that can be processed in separate FPGAs either concurrently or sequentially. The important feature of the suggested method is that subformulae may be solved in any order and inter-FPGA signals are not required. However, when the available hardware resources are much smaller than the original problem size, the decomposition can take an excessive time and will produce too many subproblems.

3.5 Dandalis et al.

Dandalis et al. [25], [26] proposed an architecture for evaluating clauses in parallel during the implication deduction phase. A distinctive characteristic of their approach is that the circuit evolves dynamically during execution, trying to achieve a more adequate level of parallelism and, consequently, optimize performance.

In the suggested architecture, all the clauses of a CNF formula are split into p groups, which deduce implications in parallel. Thus, implications are resolved independently in each group and, subsequently, a merging process combines all the results allowing the next variable assignment to be formed. If a conflict is detected, the deduction process is terminated and backtracking is performed. If no implications occur, a new decision variable is selected and assigned a value. Otherwise, the resulting variable assignment is fed again to the clause evaluator and the entire procedure is repeated until no more implications are deduced or a conflict is detected. An example of the clause evaluator architecture for $p = 3$ is shown in Fig. 8. The decision and backtrack processes are assumed to be executed by a host computer.

Each group of clauses is similar to that proposed by Zhong et al. in [19] and is organized as a linear array of

modules operated in a pipelined manner (see Fig. 8). Every module corresponds to a clause with a limited number of literals and the structure of a module is the same for all the clauses. The variables associated with a given clause are stored in the local memory of the respective module. To update the contents of local memories (to match a particular problem instance), the use of partial reconfiguration was suggested.

During problem solving, the circuit is reconfigured to implement deduction engines with different levels of parallelism p . In this task, given a set of template circuits with a different number of modules per group, the objective is to find a template that minimizes the average implication propagation delay. Obviously, this time also depends on the distribution of clauses into groups and their relative ordering within the group. However, these issues have not been considered. Instead, a greedy heuristic algorithm was proposed which is executed by a host machine and tries all the available templates in turn, starting with the one having the minimum number of modules per group. Different templates are tested while any improvement in performance is detected. Thus, the architecture evolution consists of finding the optimum number of groups for a given problem instance. The execution of the SAT algorithm is not restarted for each template, i.e., after switching templates, the search process continues with the previously found partial variable assignment. The authors report speedups of 1-6 times [25] compared to the case where $p = 1$ for a number of DIMACS SAT benchmarks [40]. These results were achieved with the aid of a software simulator and do not

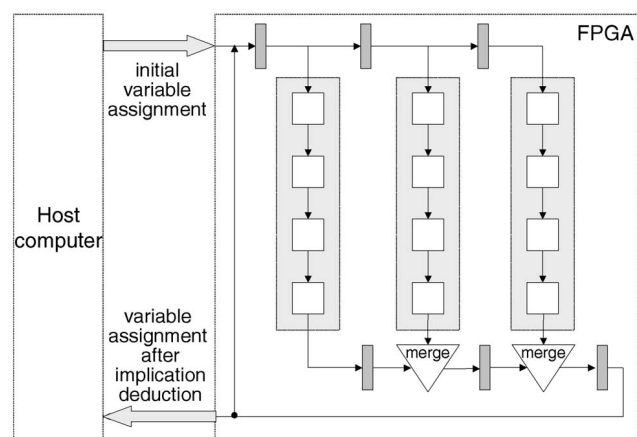


Fig. 8. A general view of the parallel clause evaluator proposed in [25].

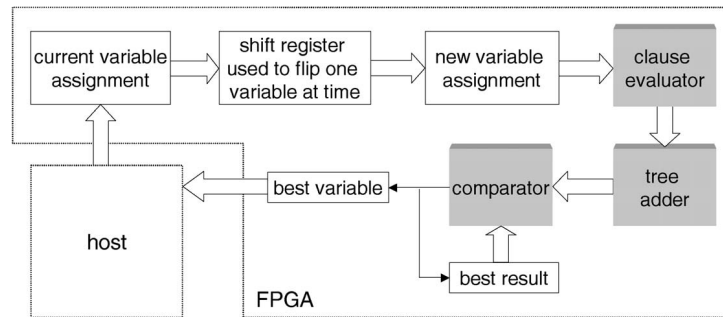


Fig. 9. Structure of the unit implementing the inner loop of the GSAT algorithm.

reflect the required hardware reconfigurations. It was also assumed that FPGA resources are sufficient for solving a SAT instance.

3.6 Leong et al.

Leong et al. studied the possibility and efficiency of the realization of complete [27] and incomplete [28], [29] SAT algorithms in reconfigurable hardware. In particular, the inner loops of the GSAT [28] and WSAT [29] algorithms for 3-SAT were partially mapped to FPGA.

In the architecture implementing the GSAT algorithm [28] (see Fig. 9), an initial variable assignment is randomly generated by a software program and downloaded to a Xilinx XC6200 series FPGA. The hardware sequentially flips each variable and evaluates the CNF formula, attempting to maximize the number of satisfied clauses. The variable conducting to the maximum number of satisfied clauses is dispatched to software, which then computes the next variable assignment by flipping that variable.

The SAT circuit was synthesized from the VHDL description, with the exception of the clause evaluator, which is a problem-dependent component and is therefore customized directly for each individual problem instance (using the possibility of partial dynamic reconfiguration, provided by XC6200 family devices). The design was tested in hardware for small CNF formulae (having up to 50 variables and 80 clauses) and the results were either worse than or comparable to the software implementation of the GSAT algorithm [28].

In [29], a similar implementation of the inner loop of the WSAT algorithm is described. The design is targeted at Virtex series FPGAs and is based on a template supporting at most 50 variables and 170 clauses. A circuit matching a particular 3-SAT formula is generated by a software program which customizes the template bit stream file. A direct manipulation of a bit stream is much faster than a complete synthesis, mapping, placement, and routing cycle, which is usually performed by instance-specific SAT solvers. But, the constraints imposed on the maximum number of variables and clauses limit the application of the circuit to rather simple CNF formulae.

The results achieved on an XCV300 FPGA under a clock frequency of 33 MHz for a number of DIMACS SAT instances [40] show an acceleration of 0.1-3.3 compared to the implementation of the WSAT algorithm in software (on a Sun SparcStation 20) [29]. These results include both the time required to customize and download to FPGA the template bit stream and the time spent in communications between the host processor and the FPGA. However,

nothing was proposed for solving problem instances whose sizes exceed the template dimensions.

3.7 Sousa et al.

De Sousa et al. [30], [31], [32] implemented a DP-based search algorithm for 3-SAT augmented with a diagnosis engine, which is activated in case of a conflict and attempts to identify the decision variables that are directly responsible for the conflict occurrence. The conflict analysis is also used to construct and add new clauses to a CNF formula, thus accelerating the search process. It was decided to partition the job between software and reconfigurable hardware with the most computationally intensive tasks (such as computing implications and choosing the next decision variable) assigned to hardware, while the control-oriented tasks (such as conflict analysis, backtrack control, and clause database management) are performed in software.

The suggested SAT solver has an *application-specific* architecture that uses configuration registers for CNF formula instantiation [31]. In order to deal with instances that exceed the available hardware capacity, a virtual hardware scheme with context switching has been proposed. In this case the SAT solver configuration data is organized in pages, which are stored in on-board memory. Each page configures a clause pipeline, as shown in Fig. 10. The data corresponding to the variables is kept in two memory blocks. The variables are read sequentially from one memory block, processed in the clause pipeline, and written into the other memory block. The next hardware page is then loaded and the variables are processed while moving from the second memory block back to the first one. The process continues while there are implications generated and no conflicts are detected.

In [31], a hardware implementation of the SAT solver based on the RC1000 PCI board from Celoxica containing one XCV2000E Xilinx FPGA and four SRAM banks (2 MB each) is described. The preliminary results [31] show that this system can run at 47 MHz and should allow formulae having up to 7,680 variables and 214,304 clauses to be processed. However, the interface with software has not been implemented yet, so real execution times are not available.

3.8 Skliarova and Ferrari

Skliarova and Ferrari [33] proposed and implemented an application-specific SAT solver based on the DP algorithm. The problem was formulated over a ternary matrix by setting a correspondence between clauses and variables of a CNF formula and rows and columns of the matrix and

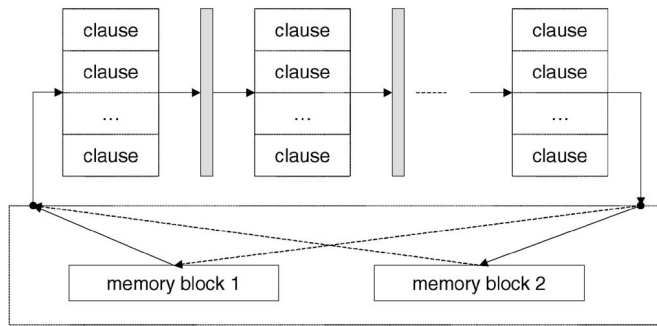


Fig. 10. Sousa et al. [30], [31], [32] hardware architecture.

finding a ternary vector, \mathbf{v} , which is orthogonal to each row of the constructed matrix. If vector \mathbf{v} cannot be found, then the formula is unsatisfiable. On the other hand, if vector \mathbf{v} exists, then the zeros and ones in it correspond to those variables that must receive values "1" and "0," respectively in order to satisfy the CNF formula.

The proposed satisfier architecture is shown in Fig. 11. The sequence of hardware operations is managed by a central control unit using a stack memory to support the backtracking process. There exist four memory blocks storing the initial matrix and its transpose. The matrices are not modified during the search process. All possible changes (such as deleting rows and columns) are reflected in the registers. The ALU executes different operations over rows and columns of matrices such as counting the number of ones, etc. In order to solve various problem instances, it is only necessary to download the respective matrix data. All the other components of the satisfier remain unchanged. This allows local reconfigurability to be used and reduces the configuration overhead.

In [34], an improved SAT satisfier architecture was proposed implementing a hybrid algorithm. The suggested algorithm requires a *favorites list* to be constructed, which contains only *permitted* partial variable assignments. Then, the DP algorithm is applied and, at each node of the decision tree, the current partial variable assignment is verified for consistency with the contents of the favorites list. When an inconsistency is detected, *premature backtracking* is performed, allowing the number of visited nodes in the decision tree to be reduced.

The SAT problem is partitioned between software and reconfigurable hardware in such a way that an FPGA is only responsible for processing subproblems that appear at various levels of the decision tree and satisfy the imposed hardware constraints (such as the maximum allowed number of rows and columns in the matrix). This technique permits formulae to be processed that exceed the resources of the available reconfigurable hardware. However, the efficiency of software/hardware partitioning strongly depends on the characteristics of a particular problem instance.

The SAT satisfier was implemented on an ADM-XRC PCI board from Alpha Data containing one XCV812E Virtex-EM FPGA (running at 40MHz). The speedups obtained for *holec* SAT benchmarks from DIMACS [40], including the FPGA configuration and software/hardware communication time, achieve two orders of magnitude compared to GRASP [9] executing on an AMD/Athlon/1GHz/256MB [33]. For other benchmarks from the same

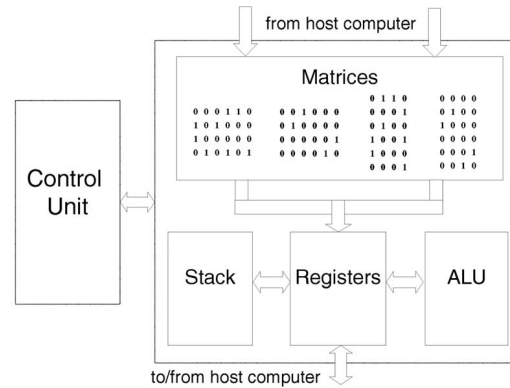


Fig. 11. The architecture of the SAT solver proposed in [33], [34].

suite [40], which are easily solved by GRASP (in fractions of a second), the hardware SAT satisfier [33] does not provide useful speedups.

4 ANALYSIS OF HARDWARE SAT SOLVERS

In this section, we attempt to analyze the reconfigurable hardware SAT solvers according to such criteria as algorithmic issues, the programming model, the execution model, reconfiguration modes, logic capacity, and performance. Table 1 summarizes the respective characteristics of the architectures considered in the previous section.

4.1 Algorithmic Issues

The majority of the existing reconfigurable hardware SAT solvers employ some variation of the Davis-Putnam algorithm [7]. The exceptions are the architectures proposed by Leong et al. [28], [29], Yap et al. [35], and Hamadi and Merceron [36], which implement incomplete algorithms, and the SAT satisfiers of Abramovici and Saab [23] and Rashid et al. [37], which realize PODEM-based algorithms. Various reconfigurable hardware SAT solvers differ also in the input format they support. For instance, the architectures of Suyama et al. [14], Leong et al. [29], and de Sousa et al. [30] are only able to work on Boolean expressions in 3-SAT CNF format; the SAT satisfiers of Zhong [20] and Dandalis and Prasanna [25] are limited to k -SAT CNF formulae; the implementations of Mencer and Platzner [21], Skliarova and Ferrari [33], and Boyd and Larrabee [38] accept any CNF formulae; and, finally, the architecture of Abramovici and Saab [23] is capable of processing an arbitrary Boolean expression. The decision variable selection strategy also varies among the SAT solvers analyzed. Although dynamic selection has been considered to be difficult to implement in hardware, it was realized in a number of architectures [14], [30], [31], [33], [34].

In present-day software SAT solvers, a lot of advanced search techniques (such as nonchronological backtracking [9] and dynamic clause addition) are employed. However, up to now, these techniques have been largely ignored by hardware SAT solvers. The only exception to this is the SAT satisfier of Zhong [20]. The main reason is that sophisticated decision and backtracking techniques require complex data structures that favor software over hardware implementations.

TABLE 1
Principal Characteristics of the Reconfigurable Hardware SAT Solvers

SAT solver	Algorithmic issues	Programming model	Execution model	Reconfiguration mode	Logic capacity
Suyama et al. [14], [15], [16]	DP-like algorithm with dynamic selection	instance-specific	hardware only	static	multi-FPGA system
Zhong [20]	DP-based algorithm with static selection, nonchronological backtracking, and conflict analysis	instance-specific	hardware only	static and dynamic (global)	multi-FPGA system
Platzner et al. [21], [22]	DP-based algorithm with static selection	instance-specific	hardware only	static	use larger device
Abramovici et al. [24]	DP-based algorithm with optimized static selection	instance-specific	hardware only	dynamic (global)	logic partitioning in subformulae
Dandalis et al. [25]	DP-based algorithm with static selection	application-specific	implications are processed in hardware, which evolves during execution; the rest is implemented in software	dynamic (partial and global)	multi-FPGA system
Leong et al. [28], [29]	incomplete algorithms (WSAT, GSAT)	application-specific	the inner loop is executed in hardware, the outer loop – in software	static and dynamic (partial)	nothing proposed
Sousa et al. [30], [31]	DP-based algorithm with dynamic selection and conflict analysis (in software)	application-specific	software/hardware partitioning according to computational complexity	dynamic (partial)	virtual hardware scheme
Skliarova and Ferrari [33], [34]	DP-based algorithm with dynamic selection	application-specific	software/hardware partitioning according to logic capacity	dynamic (partial)	software/hardware partitioning

4.2 Programming Model

There are two basic approaches to mapping a SAT formula to a reconfigurable system: *instance-specific* and *application-specific*. The first approach has been extensively explored by the reconfigurable computing community [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24] and assumes the generation of an individual hardware configuration for each problem instance. In this case, a typical design flow is to describe, synthesize, and implement either a whole instance-specific circuit or a number of precompiled primary modules, which are further customized by specially developed software tools to match the respective formula (see Fig. 12a).

In an *application-specific approach*, the circuit is designed and optimized only once, after which it can be used for solving different problem instances [25], [30], [31], [33], [34], [38]. This can be achieved with the aid of a hardware template, which is downloaded to an FPGA and customized directly there with data for a particular problem instance (see Fig. 12b). It should be emphasized that, in this case, a hardware compilation step is completely avoided and, consequently, greater acceleration over software can potentially be achieved.

4.3 Execution Model

An SAT problem can be either entirely mapped to reconfigurable hardware (leaving just the tasks of preprocessing and initialization to the host processor) [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [38] or partitioned between hardware and software [28], [29], [30], [31], [33], [34] (see Fig. 13a). In the domain of SAT solvers, two methods of software/hardware partitioning are usually employed:

partitioning according to computational complexity (see Fig. 13b) and *partitioning with respect to logic capacity* (see Fig. 13c).

The first method assigns computationally intensive portions of an application to hardware, while the remaining portions that exhibit little parallelism are handled by the host processor [25], [28], [29], [30], [31]. Reconfigurable systems of this type are based on the 90/10 rule, which states that 90 percent of the execution time of an application is spent by 10 percent of its code. Thus, in order to increase performance, an attempt is made to accelerate this small portion of an application with the aid of programmable logic devices.

The second method performs partitioning according to the available logic capacity of the hardware employed [24], [33], [34]. In this case, if a problem instance does not “fit” to a chosen device, it has first to be decomposed by software into independent subproblems, each of which satisfies the imposed hardware constraints. As a result, the upper part of the decision tree is processed by the host computer, whereas the lower part is handled in an FPGA (see Fig. 13c).

4.4 Reconfiguration Modes

In the domain of reconfigurable computing, it is common to distinguish between two configuration modes: *static mode* and *dynamic mode*. Static configuration is usually employed by instance-specific SAT solvers and assumes fixed functionality of the device once it has been programmed [13], [14], [15], [16], [17], [18], [21], [22], [23] (see Fig. 14a). Dynamic reconfiguration allows the functionality of the SAT satisfier to be changed during the solution of a problem instance (see Fig. 14b). Dynamic reconfiguration can in turn be *partial* or *global*. Global reconfiguration reserves all the

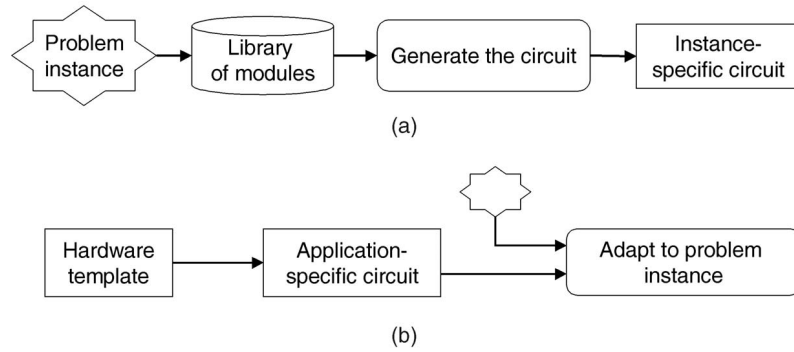


Fig. 12. (a) Instance-specific versus (b) application-specific SAT solvers.

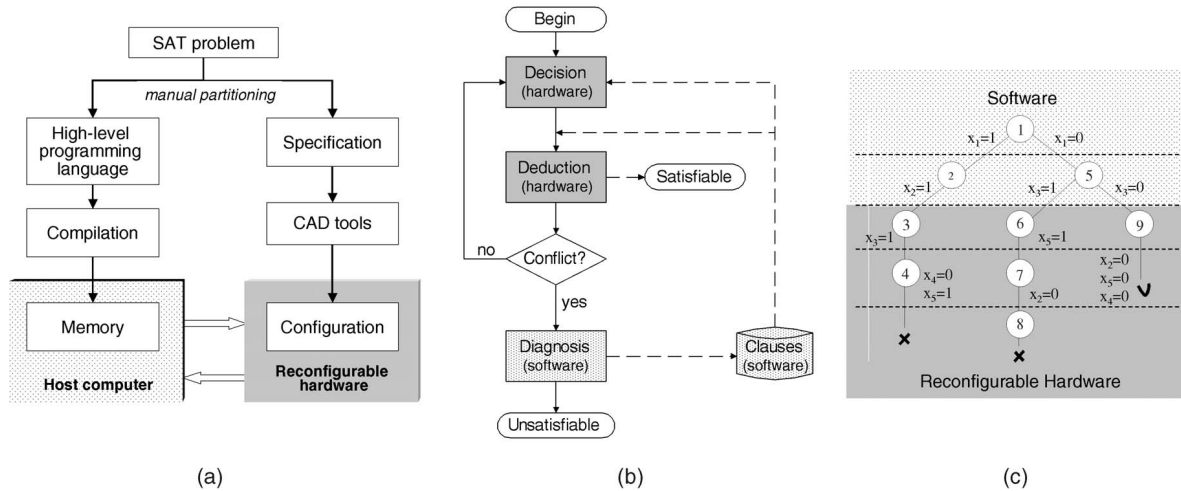


Fig. 13. Execution models: (a) An SAT problem can be either entirely mapped to reconfigurable hardware or partitioned between hardware and software. (b) Partitioning according to computational complexity: computing implications and selection of the next decision variable assigned to hardware, control-oriented tasks performed in software. (c) Partitioning with respect to logic capacity: only subproblems satisfying the resource constraints are assigned to hardware.

hardware resources for each step of execution. After a step has been concluded, the device may be reprogrammed for the next step [24], [25]. Partial reconfiguration implies the selective modification of hardware resources [25], [28], [30], [31], [33], [34].

A variety of reprogrammable devices can be employed to carry out partial dynamic reconfiguration. *Single-context* devices require complete reprogramming in order to introduce even a small change. Although many commercially available FPGAs are single-context, there exist techniques (based on hardware templates) that allow partial reconfiguration to take place [31], [33], [34]. *Multicontext* devices possess various planes of configuration information with just one of them active at any given moment. The use of *multicontext* devices for SAT satisfiers has been studied at the theoretical level [30], but, to the best of our knowledge, no physical implementation has been built. *Partially reconfigurable* devices permit small portions of their resources to be modified without disturbing the remaining parts. This kind of device (such as the XC6200 family of Xilinx) was employed for some SAT solvers [24], [28], but the potential for partial reconfigurability has only been explored by Yung et al. [28], who used this possibility to customize an instance-specific part of their design at runtime.

4.5 Logic Capacity

The logic capacity of the employed hardware device is always limited. Thus, efficient techniques are needed to deal with the situation when a problem instance exceeds the available hardware resources. The answers to this issue differ according to the programming and execution models adopted. Basically, four possibilities have been explored.

The first is the expansion of the logic capacity by interconnecting a number of programmable devices and partitioning the circuit between them. It should be noted that fast and efficient multidevice partitioning and routing is quite a difficult task (of course, modular design styles [20] can alleviate it). Moreover, the working frequency of such multidevice systems is usually limited.

The second method is to partition the problem into a series of configurations to be run either sequentially or in parallel. The partitioning is performed by decomposing an initial formula into a set of independent subformulae [24]. Each subformula must satisfy the imposed hardware constraints. The main limitation of this method is that the efficiency of the decomposition greatly depends on the characteristics of the formula. As a result, for some problem instances, the partitioning time may increase to unacceptable levels.

The third method is based on software/hardware partitioning according to the available logic capacity of

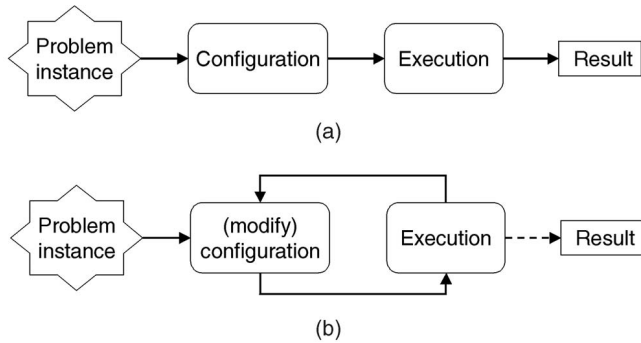


Fig. 14. SAT solvers: (a) static versus (b) dynamic reconfiguration.

the hardware that is employed (see Section 4.3). In this case, just those subproblems that appear at different levels of the decision tree and respect the capacity limitations are assigned to hardware, whereas the remaining portion of the problem is processed by a software application [33], [34]. As in the previous case, the efficiency of this method is a function of the characteristics of a given formula.

The last method is based on a virtual hardware scheme proposed in [30], [31], 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 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.

All these methods alleviate the logic capacity problem, but do not eliminate it completely. The virtual hardware scheme is currently the most successful technique since it permits the biggest formulae to be processed in hardware (up to 7,680 variables and 214,304 clauses as reported in [31]).

4.6 Performance

The total time spent by a reconfigurable hardware SAT satisfier to solve a particular problem instance is comprised of four components: hardware compilation time, hardware configuration time, time required for communication between software and hardware, and actual execution time. If a problem solution is partitioned between software and hardware, then the execution time is composed of software execution time and hardware execution time. It should be noted that the values of these components depend on the programming and execution models employed and some of them may be zero. For example, if a problem instance is entirely mapped to hardware, usually there is no communication (except for notifying the final result) between the host processor and the programmable device. In the same manner, if an application-specific approach is followed, the hardware compilation time is zero. Actually, 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 [13], [14], [15], [16], [17], [18], [21], [22]. That is why, in all recent designs, a clear intention to reduce or even avoid the hardware compilation step is apparent [19], [20], [24], [25], [30], [31], [33], [34], [38]. Therefore, we believe that the most competitive hardware SAT solvers are those that follow the application-specific approach [30], [31], [33], [34].

Of the architectures proposed and implemented so far, that of Zhong [20], because of its original solution of organizing clauses in a pipelined network, which was subsequently reproduced in many other hardware SAT satisfiers, and that of de Sousa et al. [30], [31], because of a virtual hardware scheme allowing large formulae to be processed, deserve special mention.

One characteristic inherent in reconfigurable hardware SAT solvers is that it is very difficult to analyze and compare their performance accurately. As a rule, the designers present the results achieved in the light of the software SAT satisfier GRASP [9]. However, GRASP is run on different platforms and with dissimilar parameters that heavily influence its performance. Moreover, the parameter sets are frequently not published. Besides, the majority of the SAT solvers considered involve a hardware compilation step, which is sometimes ignored (or hidden) when presenting the results.

Generally, one can observe that overall speedups achieved over software SAT solvers (including all overhead times) range from zero to one order of magnitude. The raw speedups (considering only the pure hardware execution time) have been reported to be very high (several orders of magnitude). This, on the one hand, points to a great potential and strengthens the conclusion that more work has to be invested to reduce the overheads. But, on the other hand, it is fair to say that such exciting speedups have been obtained for problem classes that are of little practical interest. These are artificially generated problems (such as *hoxex* from DIMACS [40]) on which GRASP [9], targeted at processing highly structured real-world problems, does not perform very well. Moreover, as shown by the results of the last two software SAT competitions, which took place in 2002 [41] and 2003 [42], GRASP [9] has been surpassed by more recent SAT satisfiers such as *zChaff* [43] and *BerkMin* [44]. Consequently, novel algorithmic and architectural techniques need to be explored in order to put the reconfigurable hardware SAT solvers in a more favorable position when compared to software solutions. In particular, to be highly competitive, hardware SAT solvers should incorporate advanced search strategies.

5 CONCLUSION

This paper is dedicated to the description and comparison of reconfigurable hardware SAT solvers. The performed analysis leads to the following conclusions:

- The majority of designers implement complete search algorithms derived from the DP algorithm. Conflict analysis is usually not performed and just chronological backtracking is executed (with a few exceptions).
- Initially, all the proposed SAT solvers were based on the instance-specific approach. However, the hardware compilation time restricts the range of problems for which a reconfigurable hardware solution is more effective than the software-based approach. That is why all recent efforts have been focused on avoiding instance-specific placement and routing.
- All the reconfigurable SAT solvers considered are loosely coupled systems with the programmable device (usually, a commercially available FPGA)

being attached to the host processor via an external interface.

- It is rather difficult to estimate accurately the results that have been achieved. First, the hardware compilation and configuration times are not always clearly exposed. Second, the results are usually compared to GRASP, executed on different platforms with dissimilar parameters, which can lead to variations in the solving time of up to an order of magnitude. Generally, our analysis has shown that an overall speedup of about one order of magnitude is achievable by many hardware SAT solvers for hard problem instances. No speedup has been attained for easy problem instances because of the compilation and configuration overheads.
- Real-world SAT formulae are quite large, however, how instances that do not fit into an available device can be handled efficiently is not always discussed. Recently, in what seems to be a promising solution, it was suggested that the problem should be partitioned between software and reconfigurable hardware. It should also be noted that, due to the rapid evolution in FPGA capacity, many challenging problem instances can now either be fitted into a single FPGA or at least partitioned more efficiently.
- The speedups achieved by reconfigurable hardware compared to a software solution are significant just for certain classes of SAT instances, for which the optimization techniques proposed and implemented by software SAT satisfiers are not very efficient. Some examples of these techniques are: different decision strategies, exploiting problem symmetry, careful conflict analysis, etc. Consequently, although many interesting and worthwhile architectures have already been proposed, innovative approaches still need to be explored in the reconfigurable hardware domain.

ACKNOWLEDGMENTS

The authors would like to thank Ivor Horton for his valuable comments and suggestions. 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] G. Estrin, "Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer," *IEEE Annals of the History of Computing*, pp. 3-9, Oct./Dec. 2002.
- [2] S.A. Guccione, "Reconfigurable Computing at Xilinx," *keynote talk, EUROMICRO Symp. Digital System Design*, Sept. 2001.
- [3] W.H. Mangione-Smith and B.L. Hutchings, "Configurable Computing: The Road Ahead," *Proc. Reconfigurable Architectures Workshop*, pp. 81-96, 1997.
- [4] R. Tessier and W. Burlinson, "Reconfigurable Computing for Digital Signal Processing: A Survey," *J. VLSI Signal Processing*, vol. 28, nos. 1-2, pp. 7-27, 2001.
- [5] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey," *DIMACS Series in Discrete Math. and Theoretical Computer Science*, vol. 35, pp. 19-151, 1997.
- [6] S.A. Cook, "The Complexity of Theorem-Proving Procedures," *Proc. Third ACM Symp. Theory of Computing*, pp. 151-158, 1971.
- [7] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *Comm. ACM*, no. 5, pp. 394-397, 1962.
- [8] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinatorial Logic Circuits," *IEEE Trans. Computers*, vol. 30, no. 3, pp. 215-222, Mar. 1981.
- [9] L.M. Silva and K.A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506-521, May 1999.
- [10] Z. Michalewicz and D.B. Fogel, *How to Solve It: Modern Heuristics*. Springer, 2000.
- [11] B. Selman, H. Levesque, and D. Mitchell, "A New Method for Solving Hard Satisfiability Problems," *Proc. Nat'l Conf. Am. Assoc. Artificial Intelligence (AAAI'92)*, pp. 440-446, July 1992.
- [12] B. Selman, H. Kautz, and B. Cohen, "Noise Strategies for Improving Local Search," *Proc. 12th Nat'l Conf. Artificial Intelligence*, pp. 337-343, July 1994.
- [13] M. Yokoo, T. Suyama, and H. Sawada, "Solving Satisfiability Problems Using Field Programmable Gate Arrays: First Results," *Proc. Second Int'l Conf. Principles and Practice of Constraint Programming*, pp. 497-509, 1996.
- [14] 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, 2001.
- [15] T. Suyama, M. Yokoo, and A. Nagoya, "Solving Satisfiability Problems on FPGAs Using Experimental Unit Propagation," *Proc. Fifth Int'l Conf. Principles and Practice of Constraint Programming*, 1999.
- [16] T. Suyama, M. Yokoo, and H. Sawada, "Solving Satisfiability Problems Using Logic Synthesis and Reconfigurable Hardware," *Proc. 31st Hawaii Int'l Conf. System Sciences*, vol. 7, pp. 179-186, 1998.
- [17] 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, pp. 861-868, 1999.
- [18] 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. Design Automation Conf.*, pp. 194-199, 1998.
- [19] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Solving Boolean Satisfiability with Dynamic Hardware Configurations," *Field-Programmable Logic: From FPGAs to Computing Paradigm*, R.W. Hartenstein and A. Keevallik, eds., pp. 326-235, Springer, 1998.
- [20] P. Zhong, "Using Configurable Computing to Accelerate Boolean Satisfiability," PhD dissertation, Dept. of Electrical Eng., Princeton Univ., 1999.
- [21] O. Mencer and M. Platzner, "Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment," *Proc. 32nd Hawaii Int'l Conf. System Sciences (HICSS-32 (Configurable-Reconfigurable Eng. track))*, 1999.
- [22] M. Platzner and G. De Micheli, "Acceleration of Satisfiability Algorithms by Reconfigurable Hardware," *Field-Programmable Logic: From FPGAs to Computing Paradigm*, R.W. Hartenstein and A. Keevallik, eds., pp. 69-78, Springer, 1998.
- [23] M. Abramovici and D. Saab, "Satisfiability on Reconfigurable Hardware," *Proc. Seventh Int'l Workshop Field-Programmable Logic and Applications*, pp. 448-456, 1997.
- [24] M. Abramovici and J.T. de Sousa, "A SAT Solver Using Reconfigurable Hardware and Virtual Logic," *J. Automated Reasoning*, vol. 24, nos. 1-2, pp. 5-36, 2000.
- [25] A. Dandalis and V.K. Prasanna, "Run-Time Performance Optimization of an FPGA-Based Deduction Engine for SAT Solvers," *ACM Trans. Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 547-562, Oct. 2002.
- [26] M. Redekopp and A. Dandalis, "A Parallel Pipelined SAT Solver for FPGA's," *Proc. 10th Int'l Conf. Field-Programmable Logic and Applications*, pp. 462-468, 2000.
- [27] C.K. Chung and P.H.W. Leong, "An Architecture for Solving Boolean Satisfiability Using Runtime Configurable Hardware," *Proc. Int'l Workshop Parallel Processing*, pp. 352-357, 1999.
- [28] W.H. Yung, Y.W. Seung, K.H. Lee, and P.H.W. Leong, "A Runtime Reconfigurable Implementation of the GSAT Algorithm," *Proc. Ninth Int'l Workshop Field Programmable Logic and Applications*, pp. 526-531, 1999.
- [29] P.H.W. Leong, C.W. Sham, W.C. Wong, H.Y. Wong, W.S. Yuen, and M.P. Leong, "A Bitstream Reconfigurable FPGA Implementation of the WSAT Algorithm," *IEEE Trans. VLSI Systems*, vol. 9, no. 1, pp. 197-201, 2001.

- [30] J. de Sousa, J.P. Marques-Silva, and M. Abramovici, "A Configurable/Software Approach to SAT Solving," *Proc. Ninth IEEE Int'l Symp. Field-Programmable Custom Computing Machines*, 2001.
- [31] N.A. Reis and J.T. de Sousa, "On Implementing a Configurable/Software SAT Solver," *Proc. 10th IEEE Int'l Symp. Field-Programmable Custom Computing Machines*, pp. 282-283, 2002.
- [32] R.C. Ripado and J.T. de Sousa, "A Simulation Tool for a Pipelined SAT Solver," *Proc. XVI Conf. Design of Circuits and Integrated Systems*, pp. 498-503, Nov. 2001.
- [33] I. Skliarova and A.B. Ferrari, "A Software/Reconfigurable Hardware SAT Solver," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 4, pp. 408-419, Apr. 2004.
- [34] I. Skliarova and A.B. Ferrari, "A Hardware/Software Approach to Accelerate Boolean Satisfiability," *Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pp. 270-277, 2002.
- [35] R.H.C. Yap, S.Z.Q. Wang, and M.J. Henz, "Hardware Implementations of Real-Time Reconfigurable WSAT Variants," *Proc. 13th Int'l Conf. Field-Programmable Logic and Applications*, pp. 488-496, 2003.
- [36] Y. Hamadi and D. Merceron, "Reconfigurable Architectures: A New Vision for Optimization Problems," *Proc. Third Int'l Conf. Principles and Practice of Constraint Programming*, pp. 209-215, 1997.
- [37] A. Rashid, J. Leonard, and W.H. Mangione-Smith, "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability," *Proc. Sixth IEEE Symp. FPGAs for Custom Computing Machines*, pp. 196-205, 1998.
- [38] M. Boyd and T. Larrabee, "ELVIS—a Scalable, Loadable Custom Programmable Logic Device for Solving Boolean Satisfiability Problems," *Proc. Eight IEEE Int'l Symp. Field-Programmable Custom Computing Machines*, 2000.
- [39] J.W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," PhD dissertation, Univ. of Pennsylvania, 1995.
- [40] DIMACS challenge benchmarks, <http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html>, 2001.
- [41] L. Simon, D. Le Berre, and E. Hirsch, "The SAT2002 Competition. Technical Report (preliminary draft)," <http://www.satlive.org/SATCompetition/online-report.pdf>, 2002.
- [42] 2003 SAT Competition, <http://www.lri.fr/~simon/contest03/results/>, 2003.
- [43] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. 38th Design Automation Conf.*, pp. 530-535, 2001.
- [44] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT-Solver," *Proc. Design, Automation and Test in Europe Conf.*, pp. 142-149, 2002.



specific architectures, computer-aided design, and object-oriented programming.



and the IEEE Computer Society.

Iouliia Skliarova received the MSc degree in computer engineering from the Belorussian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus, in 1998, and the PhD degree, in electrical engineering, from the University of Aveiro, Portugal, in 2004. She is currently an assistant professor in the Department of Electronics and Telecommunications at the University of Aveiro. Her research interests include reconfigurable computing, application-

António de Brito Ferrari (M'83) received the electrical engineering degrees from Universidade do Porto, Portugal, and Ecole Supérieure d'Electricité, Paris, and the MSc and PhD degrees from Brunel University, United Kingdom. Currently, he is a professor of computer engineering at the University of Aveiro, Portugal. His main research interests are in computer architecture, computer arithmetic, and reconfigurable systems. He is a member of the IEEE

▷ **For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.**