A Software/Reconfigurable Hardware SAT Solver

Iouliia Skliarova and António B. Ferrari, *Member, IEEE*

***Abstract*—This paper introduces a novel approach for solving the Boolean satisfiability (SAT) problem by combining software and configurable hardware. The suggested technique avoids instance-specific hardware compilation and, as a result, allows the total problem solving time to be reduced compared to other approaches that have been proposed. Moreover, the technique permits problems that exceed the resources of the available reconfigurable hardware to be solved. The paper presents the results obtained with some of the DIMACS benchmarks and a comparison of our implementation with other available SAT solvers based on reconfigurable hardware. The hardware part of the satisfier was realized on Virtex XCV812E FPGA, which has a large volume of embedded memory blocks that provide direct support for the proposed approach.**

***Index Terms*—Boolean satisfiability, configurable computing, field-programmable gate array (FPGA), partitioning.**

I. Introduction

THE Boolean satisfiability (SAT) problem plays an important role in the computer-aided design of integrated circuits [1], artificial intelligence, testing [2], etc. Because of the universality and wide scope of applications, the problem has been studied extensively. SAT is an NP-complete problem [3] and the available algorithms are time and resource consuming. Thus it is difficult (and sometimes even impossible) to solve many existing problems in a reasonable time, or within the computational resources that are typically available. Consequently a great deal of research effort is aimed at accelerating the execution of the relevant algorithms.

There are many different ways to increase the performance of computationally intensive algorithms. For example, it is possible to design ASICs with hardwired control and specialized functional units that can be optimized for the given problem. However, ASIC-based solutions have two main disadvantages. First, they are completely inflexible since their functionality cannot be modified after fabrication. Second, they have very high development costs, which can only be justified for large volume production.

Another solution is based on a pure software approach implemented on general-purpose computers (GPC). The latter have a fixed architecture and instruction set, and different applications are programmed within these predefined constraints. An advantage of this technique is the high level of flexibility since any change in the algorithm can be easily provided in the software. However for many time-consuming procedures the performance provided by a pure software approach is not sufficient.

An implementation based on reconfigurable hardware avoids the disadvantages associated with "pure software" (implemented in GPC) and "pure hardware" (ASIC) solutions. Indeed, the speed and resources provided by recent FPGAs are comparable with ASICs and they allow much of the flexibility of software implementations. Note, that typical clock rates for FPGAs are much lower than for GPCs of similar technology. This is due to the need to accommodate programmable interconnections. Thus a simple mapping of an application from software to FPGA does not necessarily provide a higher performance than GPCs. To achieve much higher performance three techniques are usually employed [4].

- Design of custom functional units that can perform in a few clock cycles operations that require many more clock cycles in a GPC (this is especially true for bit-level operations).
- Use of pipelining and parallel processing techniques.
- Tailoring of the memory interface to the problem requirements.

The majority of researches in the area of accelerating SAT solutions using reconfigurable hardware apply an instance-specific approach, i.e., a specialized hardware circuit is generated for each problem instance to be considered [5]–[7]. In this case the total problem solving time is equal to "hardware circuit generation time" + "FPGA configuration time" + "actual execution time." The primary advantage of this strategy is that a direct mapping of a given problem data to functional components permits performance to be increased significantly and it provides a good utilization of the resources. There exist a number of special-purpose techniques that allow the generation of the required configurations for FPGAs to be accelerated [7], [8]. These techniques include developing of customized software tools instead of using commercially-available ones, exploiting modular design styles, etc. Nevertheless the time for hardware compilation is still considerable and for many problems (usually easy ones) it is higher than the time for the execution of combinatorial algorithms in both hardware and software. Thus this method can only be used efficiently for difficult tasks, for which the hardware compilation time is offset by the reduced execution time.

Another important issue affecting any algorithm implemented on reconfigurable hardware is related to the capacity of the hardware platform. If the required circuit cannot be implemented in a single FPGA, it is possible to employ several FPGAs by applying special methods for multi-FPGA partitioning. Nevertheless there is no *a priori* guarantee that a given task can be efficiently mapped onto a given set of reconfigurable hardware resources, and that it will complete in reasonable time.

Manuscript received October 10, 2001; revised January 21, 2003. This work was supported in part by the Portuguese Foundation of Science and Technology under Grant FCT-PRAXIS XXI/BD/21353/99.

The authors are with the Department of Electronics and Telecommunications/IEETA, University of Aveiro, 3810-193 Aveiro, Portugal (e-mail: iouliia@det.ua.pt; ferrari@det.ua.pt).

Digital Object Identifier 10.1109/TVLSI.2004.825859

The remainder of this paper is organized as follows. In Section II, the SAT problem is described in detail. Section III provides a survey of the related work. Section IV is devoted to the algorithm we use. The architecture of the proposed SAT solver is presented in Section V. Section VI describes the methodology for partitioning the problem between software and reconfigurable hardware. Experimental results on benchmark problems, including performance comparisons, are presented in Section VII. Finally, concluding remarks are given in Section VIII.

II. Problem Statement

The SAT problem consists of determining if a Boolean formula is satisfiable, i.e., whether there exists an assignment of values to variables that forces the formula to evaluate to "1." 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 one or more literals. A literal is a variable or its negation. For example, the following formula contains 3 variables and 4 clauses, and is satisfied when $x_1 = x_2 = x_3 =$ "1":

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

On the other hand the following formula is unsatisfiable:

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

Algorithms for solving the SAT problem can be divided in two major groups: complete and incomplete [9]. 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.

The most known complete algorithm for SAT is the classical Davis-Putnam (DP) algorithm [10], in which the search process is organized by implicitly traversing the space of all possible assignments of values to variables, and is usually represented by a decision tree. The root of the decision tree corresponds to a start point, where all the variables are unassigned. The other nodes represent situations that can be reached during the search process. These nodes are characterized by the respective partial assignments. If at any node a partial 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).

In order to pass from one node of the decision tree to the subsequent one it is necessary to make a *decision*. The *decision* consists of choosing one unassigned variable and assigning a value to it (this variable is called 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 pre-ordered 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 such that the formula is more likely to be satisfied (for this purpose different heuristic methods are employed).

After each decision, special techniques can be applied to prune the search space. The most known of these techniques are the *unit clause* rule and the *pure literal rule* [9].

The *unit clause* rule permits the current partial assignment to be extended by identifying all direct and transitive *implications* of the decision variable on other variables. This is accomplished by finding *unit clauses*, i.e., clauses that have only one unassigned literal l. Obviously, in order to satisfy such clauses the respective literal l must get the value "1." The variable corresponding to this literal is said to be implied to the respective value (to "1" if l is positive, or to "0" if l is negative).

The *pure literal rule* consists of extending the current partial assignment by finding *pure* literals. A literal is called *pure* if all its occurrences in a formula are either all positive or all negative [9]. The variables corresponding to such literals can be set to either the value "1" (if a literal is positive) or the value "0" (if a literal is negative), without affecting the satisfiability of the formula.

When a value is assigned to a variable in a formula (either by means of decision, implication, or application of the pure literal rule), some clauses may become satisfied and some may remain undefined. All satisfied clauses as well as all literals with the value "0" are removed from the formula and the search process continues.

If a variable has different values implied by two or more clauses, then a *conflict* exists. In this case it is necessary to backtrack. For that the algorithm erases all the actions performed after the last decision and inverts the value of the current decision variable. If a conflict occurs again, the algorithm recedes to the most-recently assigned variable with unfinished revision and inverts its value. If backtracking beyond the first decision variable is attempted, it means that all possible assignments have been exhausted and there is no solution to the problem.

The majority of the state-of-the-art software SAT solvers (GRASP [11], Relsat [12], zChaff [13], BerkMin [14], etc.) have been derived from the DP algorithm by augmenting it with special sophisticated techniques to prune the search space. Examples of such techniques are dynamic selection of the next decision variable, nonchronological backtracking (backtracking is called nonchronological if the algorithm is able to go back a number of levels in the decision tree by identifying and skipping those of its branches that cannot lead to a solution), conflict clause recording (the purpose of conflict clause recording is to prevent the occurrence of conflicts similar to those that have already arisen), etc.

III. Related Work

Recently, several research groups have explored different approaches to solve the SAT problem with the aid of reconfigurable hardware [6]–[8], [15]–[21].

Zhong *et al.* implemented a version of the DP algorithm. In their early work [15] they constructed an implication circuit and a state machine for each variable in the formula. All the state machines are connected in a serial chain. In each period of time, only one state machine is active. As soon as this state machine completes processing, it transfers control either to the next state machine (forward search) or to the previous one (backtracking). Thus the solver uses a distributed control for variable assignments. 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 acti-

vate 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. For every decision variable the value "1" is always tried before the value "0." In [15] hardware implementation of nonchronological backtracking was also proposed. The resulting hardware execution time was quite good but the design possessed two distinctive drawbacks. First, the clock frequency was low (ranging from 700 kHz to 2 MHz for different formulae [15]). Second, the hardware compilation time took several hours (on a Sun 5/110 MHz/64 MB) thus canceling all the advantages of fast hardware execution for many problems.

In more recent work [6], [8], the basic design decisions were revised. As a result, a regular ring-based interconnecting structure was employed instead of irregular global lines, essentially reducing the compilation time (to an order of seconds) and increasing the clock rate (to 20–30 MHz). Besides, a technique enabling conflict clauses to be generated and added was proposed. As opposite to [15], the design described in [6], [8] is clause-oriented, with the clauses arranged in processing elements distributed along a pipelined communication network. The main control unit keeps track of the current state of the SAT solver and monitors the network for both variable's value changes and conflicts. To address large problems instances that do not fit into one FPGA chip, the authors propose to employ several interlinked FPGAs. The experimental results are based on both hardware implementation (on an IKOS emulator) and simulation. The speedups achieved over the software satisfier GRASP [11] run on a Sun5/110 MHz/64 MB (in a restricted mode), including the hardware compilation and configuration time, are an order of magnitude [8] for a subset of the DIMACS SAT benchmarks [22].

The SAT solver proposed by Abramovici *et al.* [16] is based on the PODEM algorithm 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 having initially an unknown value "x" [16]. 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. In [7] an improved architecture is suggested that employs the DP algorithm and implements an enhanced variable selection strategy. For hardware implementation Abramovici *et al.* suggest creating a library of basic modules that are to be used for any formula. The modules have predefined internal placement and routing. In this case the solver circuit will be built from modules, which allows the compilation time to be reduced (to the order of minutes). The authors implemented simple circuits on XC6264 FPGA and simulated the bigger ones. For a circuit occupying the whole area of the XC6264 FPGA the clock frequency is about 3.5 MHz. In [7] Abramovici *et al.* report speedups from 0.01 to 7000 (after time unit justification) compared to GRASP [11] for a subset of DIMACS benchmarks [22]. In [7] a virtual logic system was also proposed. It allows reconfigurable circuits to be constructed for solving SAT problems that are larger than the available hardware resources. This is achieved by decomposing a formula into independent sub-formulae that can be processed in separate FPGAs either concurrently or sequentially.

A SAT solver proposed by Platzner *et al.* [17], [18] is similar to that of Zhong [15]. It consists of a column of finite state machines (FSM's), deduction logic and a global control unit. The deduction logic computes the result of the formula based on the current partial variable assignment. All variable assignments are tried in a fixed order and for each variable the value "0"' is tested before the value "1." 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 active 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 evaluates always 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 4 Xilinx XC4028 FPGAs. The speedups obtained for $\mathrm{hole6}\ldots\mathrm{hole10}$ benchmarks from DIMACS [22], including hardware compilation and configuration time, range from 0.003 (for $\mathrm{hole6}$) to 7.408 (for $\mathrm{hole10}$) compared to GRASP run on a PII/300 MHz/128 MB [18]. The designs for the holex problems run at 20 MHz [18]. In [17] two architecture extensions were described. In the first, additional clauses are introduced to identify the conditions that point to *don't care* variables. This allows decisions on these variables to be avoided. In the second architecture, logical implications that are caused by value assignments are explored (the same as in the DP algorithm).

More recent work in this direction is targeted at avoiding instance-specific layout compilation. Boyd and Larrabee [20] proposed an architecture for a SAT-specific PLD that excludes instance-specific placement and routing. The suggested design consumes polynomial hardware resources (with respect to the number of variables and clauses) and requires polynomial time to configure. The authors implemented a small version of their satisfier for a problem having 8 variables and 8 clauses on a Xilinx XC4005XL running at 12 MHz. However, no results on large benchmark problems were reported.

Sousa *et al.* [21] were the first to propose partitioning 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, and the control-oriented tasks (such as conflict analysis, backtrack control and clause database management) performed in software. In order to deal with instances that exceed the available hardware capacity, a virtual hardware scheme with context switching has been proposed. The results reported in [21] are based on a software simulator of the system under an estimated clock frequency of 80 MHz, assuming that the context-

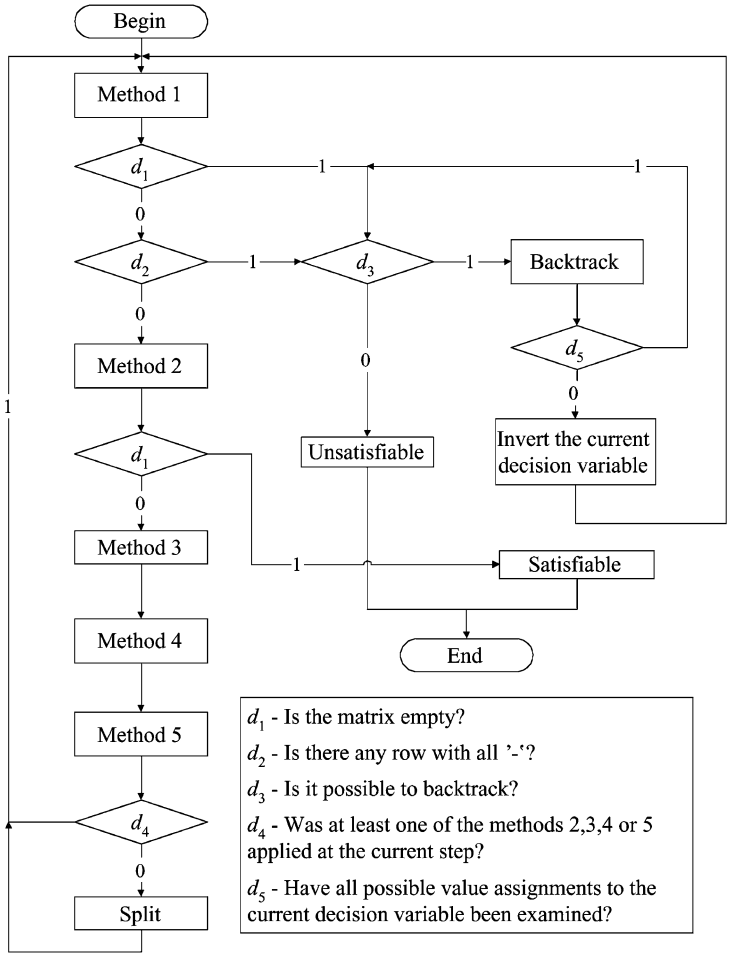

Fig. 1. Algorithm for solving the satisfiability problem formulated over a discrete matrix.

switching device can swap pages in one cycle. For hole9, hole10 problems from DIMACS the speedup compared to GRASP is about 3 ×.

IV. Algorithm

A SAT problem can be formulated over different mathematical models such as Boolean functions and discrete matrices. It is important that one model can be formally transformed into another. We have selected discrete matrices as the primary mathematical model because they can easily be represented in both software and hardware. Besides, the architecture of the SAT solver that we are going to propose must allow a variety of problem instances to be processed, and the suggested matrix model is more suited to this purpose because it significantly eases the process of configuring the circuit with the particular problem data.

Let us formulate a SAT problem over a ternary matrix $\mathbf{T}$ by setting a correspondence between clauses and rows of $\mathbf{T}$, and between variables and columns of $\mathbf{T}$. If there exist m clauses and n variables in a formula then each element $t_{ij}, i = 1, \ldots, m, j = 1, \ldots, n$, of the matrix $\mathbf{T}$ is equal to:

- "1" — if clause c_i contains variable x_j;
- "0" — if clause c_i contains variable x_j with negation;
- "-" (*don't care*)—if clause c_i does not contain variable x_j.

For example, the following formula:

$$(x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee x_2) \qquad (1)$$

can be represented by a ternary matrix $\mathbf{T}$

$$T = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 \end{matrix} \\ \begin{bmatrix} \mathbf{1} & - & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & - \\ - & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & - \end{bmatrix} \end{array} \begin{matrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{matrix}.$$

Note that the problem of satisfying the Boolean formula is equivalent to finding a ternary vector $\mathbf{v}$, which is orthogonal to each row of the corresponding matrix $\mathbf{T}$ [23]. Two ternary vectors $\mathbf{a} = [a_1 a_2 \ldots a_n]$ and $\mathbf{b} = [b_1 b_2 \ldots b_n]$ are orthogonal if there exists $i, i = 1, \ldots, n$, such that either $a_i =$ '0' and $b_i =$ '1', or $a_i =$ '1' and $b_i =$ '0'. 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 formula. For the example considered above, a solution is $\mathbf{v} =$ [-01], then $x_2 =$ '1' and $x_3 =$ '0'. It is easy to check that such assignment of values to variables satisfies the formula (1).

In order to solve the SAT problem formulated in terms of a ternary matrix we applied an algorithm proposed in [23] (see Fig. 1). The algorithm consists of a sequential application of various reduction and splitting methods. The reduction

methods allow the matrix $\mathbf{T}$ to be simplified by deleting rows and columns that cannot influence the final solution, and to assign the value "1" or "0" to some component of vector $\mathbf{v}$ that initially is completely undetermined, i.e., $\mathbf{v} = [\text{-}\ldots\text{-}]$. Let us describe the reduction methods applied indicating how they relate to the DP algorithm outlined in Section II. There are five reduction methods:

- Method 1. All the columns that are completely undetermined (i.e., do not contain either "0" or "1") are deleted from the matrix $\mathbf{T}$. This method enables us to detect *don't care* variables. The variable is *don't care* because it does not influence in any way the satisfiability of the formula, and consequently it can be excluded from further consideration.
- Method 2. All the rows that are orthogonal to the vector $\mathbf{v}$ are deleted from the matrix $\mathbf{T}$. This method is equivalent to deleting the satisfied clauses from the formula.
- Method 3. All the columns that correspond to determined components of vector $\mathbf{v}$ are deleted from the matrix $\mathbf{T}$. This operation deletes an assigned variable from further consideration.
- Method 4. If there exists a row in the matrix $\mathbf{T}$ that has only one component with the value "0" or "1" and all the other components equal to '-', then the corresponding element of vector $\mathbf{v}$ is set to the inverse value. This operation corresponds to the application of the unit clause rule.
- Method 5. If there exists a column in the matrix $\mathbf{T}$ that does not contain the value "0" ("1") then this value is assigned to the corresponding component of vector $\mathbf{v}$. This method is equivalent to the application of the pure literal rule.

When further reduction becomes impossible a splitting method is applied. The method selects an undetermined component of vector $\mathbf{v}$ and tries to assign a value to it. It is necessary to choose the component that corresponds to the more determined column of the matrix $\mathbf{T}$, i.e., to a column that has a minimal number of values "-." Basically, the splitting method employed corresponds to the selection of the next decision variable and assigning a value to it. The implemented strategy is equivalent to the so-called *maximum occurrence in clauses* heuristics, which tries to satisfy as many clauses as possible. In our case we choose a variable that appears in the maximum number of rows of the matrix $\mathbf{T}$, and assign a value to it that makes as many rows as possible become orthogonal to the vector $\mathbf{v}$. Thus a *dynamic selection* of the next decision variable is performed.

If after deleting a row the matrix becomes empty, then the current value of the vector $\mathbf{v}$ represents the solution, because in this case all the clauses are satisfied. On the other hand, if the matrix becomes empty after deleting a column or if it contains a row without values "1" and "0," then the current partial assignment will not lead to the solution. Such a situation corresponds to having an empty clause in a formula. In this case the algorithm backtracks to the most-recently assigned component of vector $\mathbf{v}$ with unfinished revision and inverts its value, i.e., *chronological* backtracking has been implemented. If backtracking beyond the first decision variable is attempted, it means that the formula is unsatisfiable.

The algorithm can be presented in the form of a decision tree. In this case each node of the decision tree is characterized by a ternary vector $\mathbf{v}$ and by the matrix $\mathbf{T}'$ composed of some minors of $\mathbf{T}$ (i.e., some submatrices of $\mathbf{T}$). At the beginning the vector $\mathbf{v}$ is completely undetermined and $\mathbf{T}' = \mathbf{T}$. The transition from one node of the decision tree to another is performed when a reduction or splitting method is applied.

Let us employ the algorithm considered above to verify if the following Boolean formula is satisfiable:

$$\begin{aligned} f(x_1,\ldots,x_7) &= (\bar{x}_1 \vee x_5) \wedge (x_5 \vee \bar{x}_6 \vee x_7) \\ &\wedge (x_1 \vee x_5) \wedge (x_2 \vee x_4 \wedge \bar{x}_5) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_5) \\ &\wedge (x_2 \vee \bar{x}_5 \vee x_7) \wedge (x_1 \vee \bar{x}_4 \vee \bar{x}_5). \end{aligned} \quad (2)$$

The formula can be converted to the following matrix $\mathbf{T}$:

$$\mathbf{T} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{matrix} \\ \begin{bmatrix} \mathbf{0} & - & - & - & \mathbf{1} & - & - \\ - & - & - & - & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{1} & - & - & - & \mathbf{1} & - & - \\ - & \mathbf{1} & - & \mathbf{1} & \mathbf{0} & - & - \\ \mathbf{0} & \mathbf{0} & - & - & \mathbf{0} & - & - \\ - & \mathbf{1} & - & - & \mathbf{0} & - & \mathbf{1} \\ \mathbf{1} & - & - & \mathbf{0} & \mathbf{0} & - & - \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix}$$

Fig. 2 depicts the decision tree for this example. The decision tree is relatively simple and has just two branching points (nodes 1*a*-*b* and 3*a*). The numbers of nodes indicate the order in which various intermediate situations are examined. Fig. 2 also presents data about current values of the vector $\mathbf{v}$ and intermediate matrices $\mathbf{T}'$, which have been constructed during the search for results, and the names of methods that have been employed at the various steps of the algorithm (see information in parentheses).

It is easy to check that the discovered vector $\mathbf{v}$ is orthogonal to each row of the matrix $\mathbf{T}$. Thus formula (2) is satisfied by the following assignment of values to variables: $x_1 =$ '0' $x_2 =$ '1' $x_4 =$ '1' $x_5 =$ '1' $x_6 =$ '0', and $x_7 =$ '1'.

V. Architecture of SAT Solver

As mentioned in the introduction, hardware compilation time restricts the range of problems where the instance-specific approach can be useful compared to software SAT solvers. In order to eliminate this constraint we have attempted to design a universal circuit, or hardware template, whose structure is not changed for different problems.

The basic components of the proposed architecture are shown in Fig. 3. The central control unit executes the algorithm described in Section IV. The control unit is described in VHDL and is implemented as a finite state machine. Its behavior corresponds to the flowchart depicted in Fig. 1.

The block Registers consists of the following components:

- $\mathrm{cur\_row\_addr}$ — is a $\log_2 m$-bit register that indicates the address of the active row of the matrix $\mathbf{T}$.

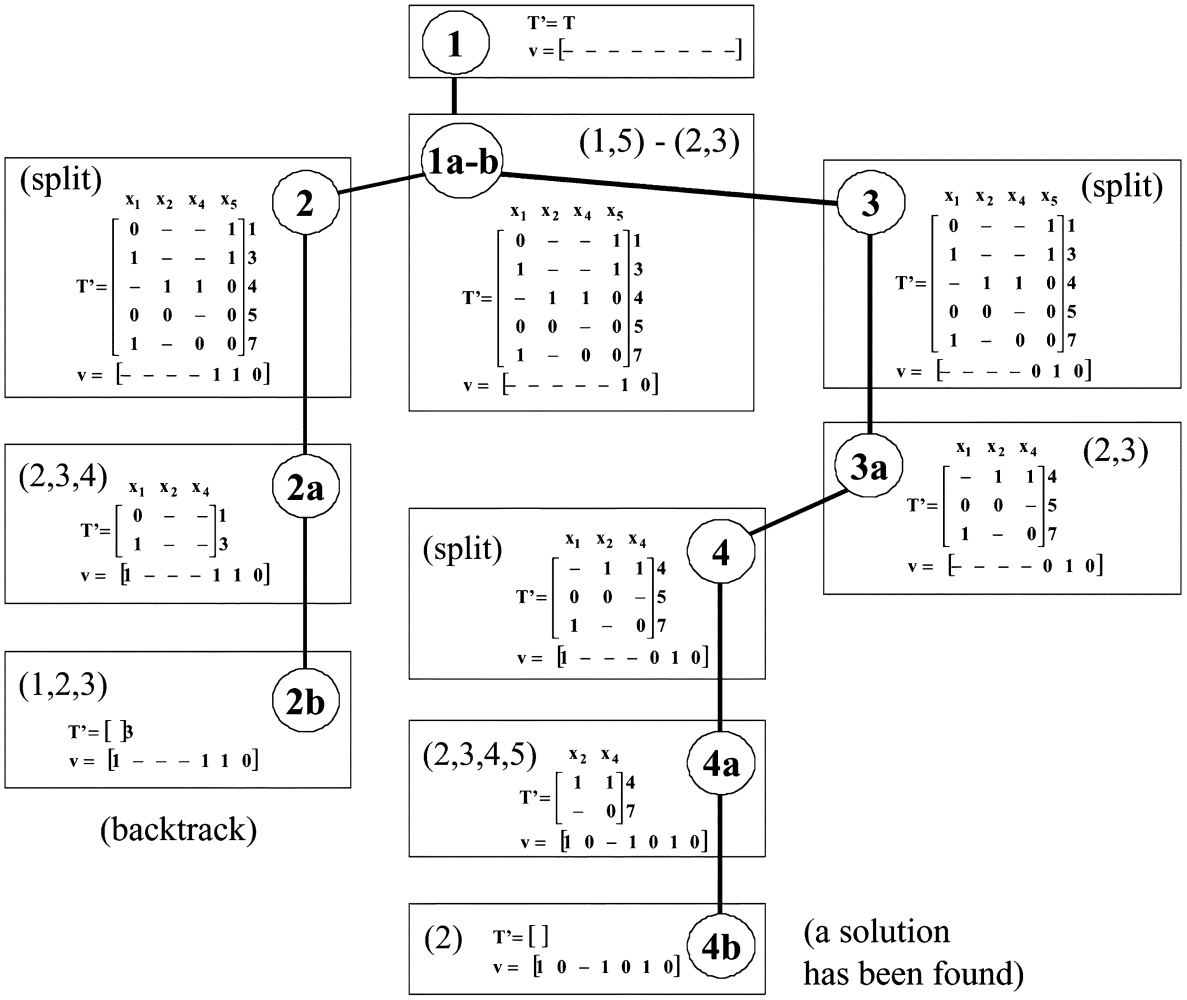

Fig. 2. Decision tree for finding the vector $\mathbf{v}$, which is orthogonal to any row of the matrix $\mathbf{T}$.

- cur_col_addr — is a $\log_2 n$-bit register that keeps the address of the active column of the matrix $\mathbf{T}$.
- cur_v — is composed of two n-bit registers containing the current value of the vector $\mathbf{v}$. The first of these registers (cur_v_o) contains ones in the positions in which vector $\mathbf{v}$ is equal to "1," whereas the second register (cur_v_z) holds ones in the positions where $\mathbf{v}$ is equal to "0." As a result, cur_v_$o[i]$ ='1' if and only if $\mathbf{v}[i]$ ='1', and cur_v_$z[i]$ = '1' if and only if $\mathbf{v}[i]$ = '0' $i = 1, \ldots, n$.
- del_rows — is an m-bit register that indicates (by ones) which rows of the matrix $\mathbf{T}$ have been deleted.
- del_cols — is an n-bit register that indicates (by ones) which columns of the matrix $\mathbf{T}$ have been deleted.
- des_var — is a $\log_2 n$-bit register that stores the number of the current decision variable.
- test_val — is a 2-bit register that keeps track of what value has been assigned to the current decision variable and indicates whether both values have already been tried.
- aux_regs — a number of auxiliary registers that will not be discussed here.

There are four RAM blocks storing matrices in Fig. 3. The first two (Rows_o and Rows_z) correspond directly to the matrix $\mathbf{T}$ and the remaining blocks (Cols_o and Cols_z) represent a transpose of $\mathbf{T}$. Each element $t_{ij}, i = 1, \ldots, m, j = 1, \ldots, n$, of $\mathbf{T}$ is encoded as follows:

- if t_{ij} ='1', then Rows_o[i][j] ='1', Rows_z[i][j] = '0', Cols_o[j][i] = '1' and Cols_z[j][i] = '0'.
- if t_{ij} = '0', then Rows_o[i][j] = '0', Rows_z[i][j] = '1',Cols_o[j][i] = '0', and Cols_z[j][i] = '1'.
- if t_{ij} = '–', then Rows_o[i][j] = '0', Rows_z[i][j] = '0', Cols_o[j][i] = '0', and Cols_z[j][i] = '0'.

Since we keep both the matrix $\mathbf{T}$ and its transpose, we can access any row and column of $\mathbf{T}$ within one clock cycle. The matrices themselves are not modified during the search process. All possible changes (such as deleting rows and columns) are reflected in the registers. Thus there is no need to keep the intermediate matrices in the stack. The matrices are stored in the FPGA embedded memory blocks.

The maximal dimensions of matrix $\mathbf{T}$ have been fixed and they are equal to $m_{\max} \times n_{\max}$. If it is necessary to process a matrix of smaller dimensions then the actual parameters, m_{act} and n_{act}, will be stored in two special registers. Taking into account these values the control unit will only enforce the handling of the required area of the matrices as shown in Fig. 4.

The ALU is used to calculate the number of ones and zeros in rows and columns of the matrix, to determine whether any row

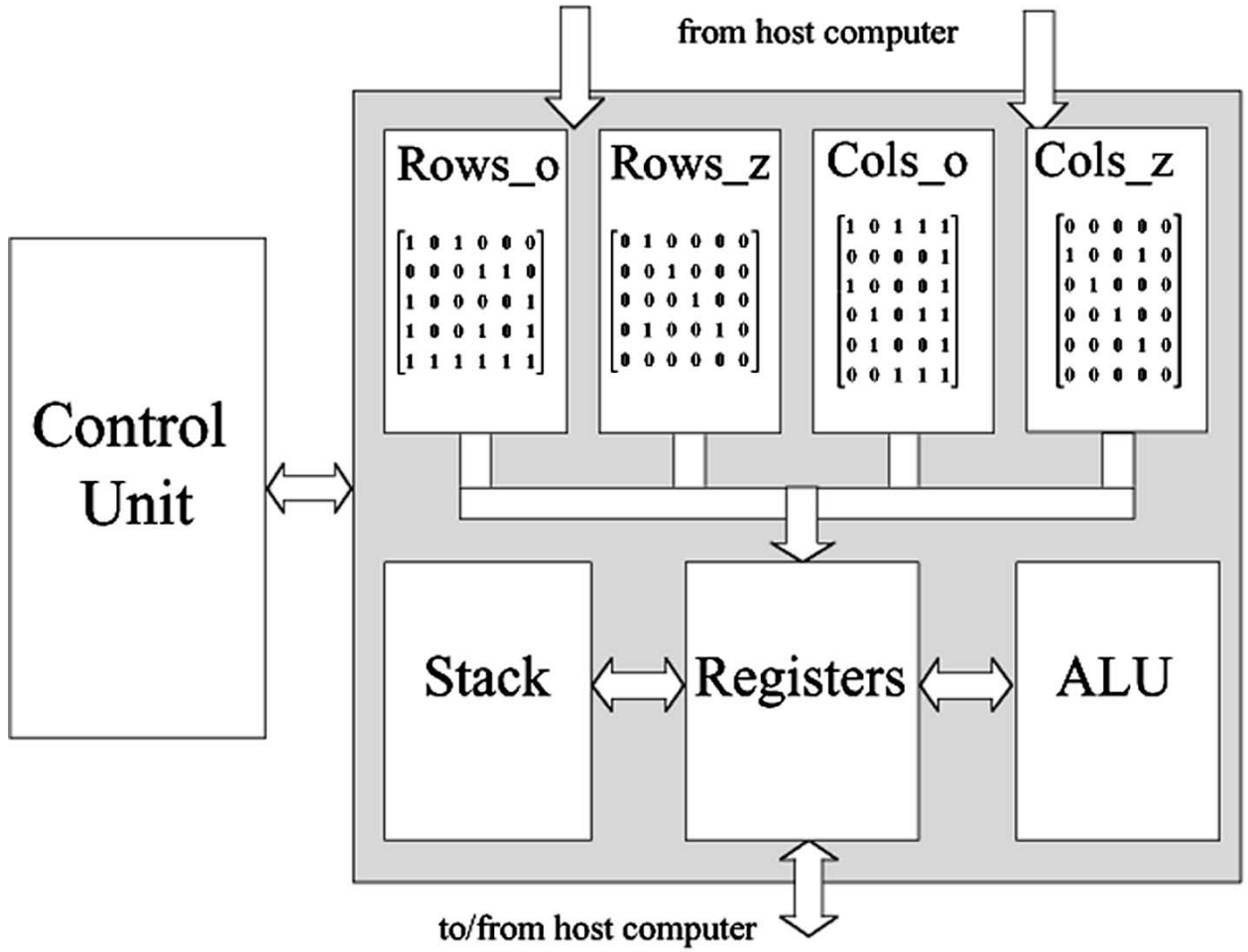

Fig. 3. The SAT solver architecture.

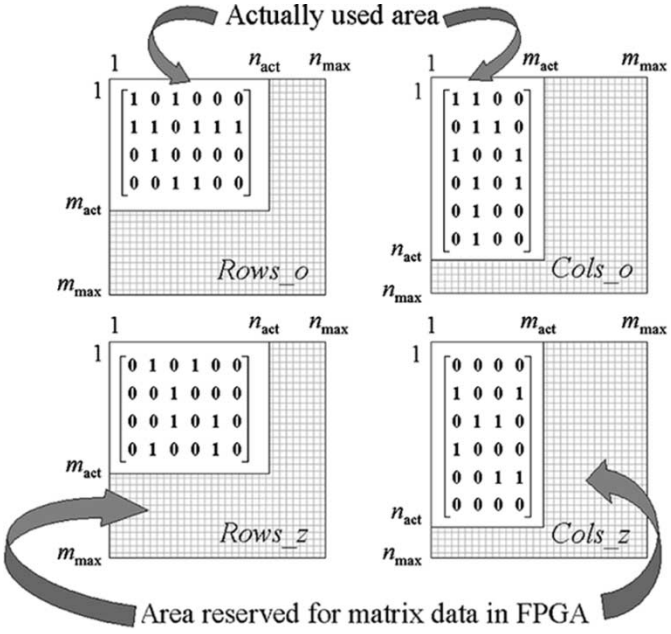

Fig. 4. Treatment of the matrices in the FPGA.

is orthogonal to the vector $\mathbf{v}$, etc. For example, Fig. 5 presents a sub-circuit that is utilized in methods 1 and 5 of the algorithm from Fig. 1 and allows a column to be processed in one clock cycle.

The stack memory supports the backtracking process. Structurally, the stack is composed of a (n-1)-bit counter, which stores the current decision level (i.e., the current depth of the decision tree), and six RAM blocks of depth (n-1) that store the values of the registers $\mathrm{cur\_}v\mathrm{\_}o, \mathrm{cur\_}v\mathrm{\_}z, \mathrm{del\_rows}, \mathrm{del\_cols}, \mathrm{des\_var}$, and $\mathrm{test\_val}$, respectively for each decision level. As a result, the largest depth of the decision tree can be (n-1). Obviously, such a big number is not required and can be reduced. When a splitting method is applied, the current values from the registers are stored at the respective addresses in the RAM blocks, and the stack pointer is incremented. During the backtracking process, the stack pointer is decremented and the required values are restored, i.e., the data is moved from the RAM blocks back to the registers.

An ADM-XRC PCI board [24] containing one XCV812E Virtex Extended Memory FPGA [25] was used as the reconfigurable hardware platform. This FPGA is composed of a 56×84 array of CLB's and it incorporates 140 KB of dedicated block RAM. This type of FPGA is very well suited to the proposed architecture of a SAT solver because large amounts of block RAM can be used to store matrices. Interaction with the FPGA is carried out with the aid of the ADM-XRC API library, which provides support for initialization, loading configuration bitstreams, data transfers, interrupt processing, clock management and error handling.

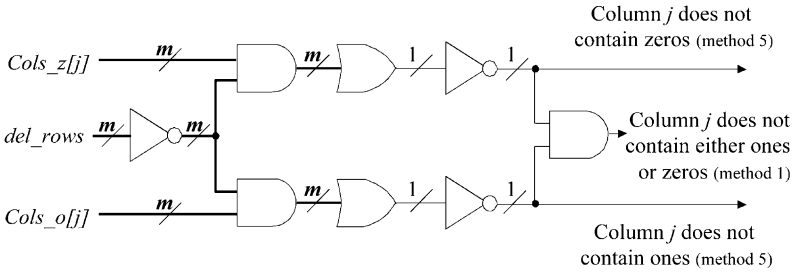

Fig. 5. A circuit that is utilized in methods 1 and 5 of the algorithm.

TABLE I
PARAMETERS OF THE CIRCUITS IMPLEMENTED

Circuit	Area (*slices*)	% of XCV812E resources	Clock frequency (MHz)
c64	1557	16	40.725
c128	2848	30	32.858
c256	5158	54	30.516

Three different circuits have been implemented with the following maximal dimensions $m_{\max} \times n_{\max}$ of matrix $\mathbf{T}$: 64×32, 128×64, and 256×128 (we will refer to these designs as $c64, c128$ and $c256$). Table I contains information about the area occupied by each circuit and its clock frequency. The area is shown in the number of Virtex slices (each CLB is composed of two slices). It should be noted that the hardware requirements of the proposed SAT solver are not a function of the size and complexity of the formula being solved.

VI. SOFTWARE/RECONFIGURABLE HARDWARE PARTITIONING

The hardware SAT solver described in Section V satisfies the requirements considered above: its structure is not changed from one problem to another. It is only necessary to specify the actual dimensions of the matrix $\mathbf{T}$ (i.e., parameters m_{act} and n_{act}).

An important problem to point out is that obviously it is not possible to store and process matrices of arbitrary sizes in an FPGA. Because of that we suggest the strategy considered below should be applied. When the decision tree is being constructed, various splitting and reduction methods are applied. As a result the initial matrix dimensions are gradually decreased as we move from the root of the tree to its leaves (this can be seen in Fig. 2). Thus each problem can be treated according to the technique depicted in Fig. 6.

Initially, the algorithm described in Section IV was implemented by a software application developed in C++. Then, for each problem instance the software program configures the FPGA with the respective SAT circuit. After that, if the initial matrix dimensions satisfy the predefined constraints (i.e., the maximum allowed number of rows $m_{\max}$ and columns $n_{\max}$) then the matrix data will be entirely transferred to FPGA and the problem will be completely solved in hardware. Otherwise, the software will be trying to solve the problem. During this process it will apply special splitting and reduction methods until an intermediate matrix that does not exceed the capacity constraints is obtained. The FPGA will then be responsible for the subsequent steps. If the reconfigurable hardware finds a solution, the problem is solved and the result will be dispatched

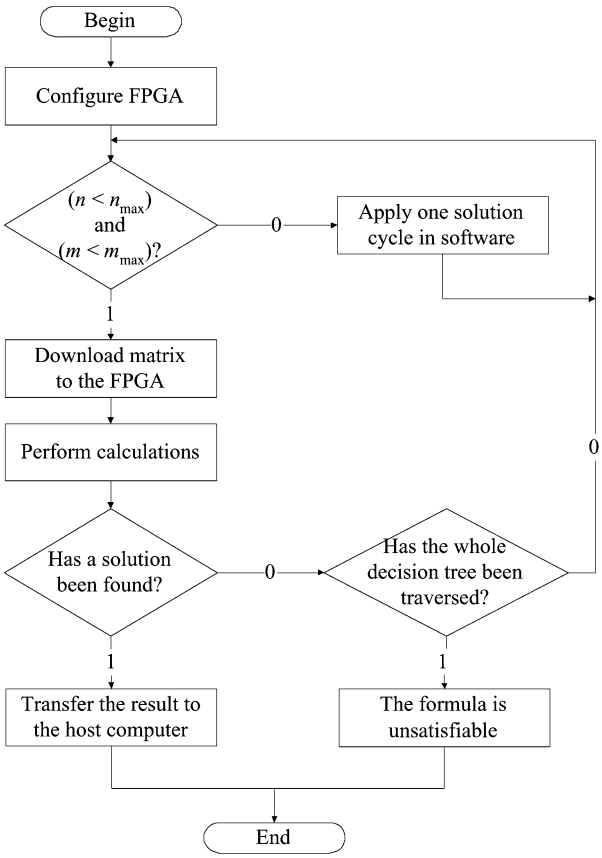

Fig. 6. Collaboration of software and reconfigurable hardware.

to the host computer. If it does not, control will be returned to software. The software will continue to traverse the decision tree eventually reaching some other point where the matrix dimensions will fall within the constraints. Then the matrix data will be transferred to the FPGA and it will try to solve the sub-problem. These steps will be repeated until we either get the solution or conclude that the formula is unsatisfiable.

VII. EXPERIMENTAL RESULTS

In order to estimate the effectiveness of the proposed approach, a number of experiments have been conducted. For this purpose the *Pigeon hole* problem from DIMACS [22] was chosen. The problem consists of checking whether it is possible to place $n + 1$ pigeons in n holes without two pigeons being in the same hole. Obviously, it is not possible, thus all instances are unsatisfiable.

This problem is very time-consuming when it is being solved in software. Tables II–IV contain the results of handling this

TABLE II
EXPERIMENTAL RESULTS FOR THE PIGEON HOLE PROBLEM (SAT SOLVER *SOFT/C64*)

Instance	$m \times n$	GRASP runtime (s)	t_{config} (s)	t_{sw} (s)	t_{hw} (s)	t_{comm} (s)	t_{total} (s)	Speedup	Matrix downloads
Hole6	133 × 42	0.14	0.31	0.0967	0.0598	0.0165	0.483	0.289	60
Hole7	204 × 56	4.31		0.7771	0.4161	0.1188	1.622	2.657	420
Hole8	297 × 72	51.574		7.544	3.3309	0.9831	12.168	4.238	3360
Hole9	415 × 90	531.961		72.1986	30.0108	9.1366	111.656	4.764	30240
Hole10	561 ×110	5685.6		806.771	299.5374	94.1716	1200.79	4.734	302400

TABLE III
EXPERIMENTAL RESULTS FOR THE PIGEON HOLE PROBLEM (SAT SOLVER *SOFT/C128*)

Instance	$m \times n$	GRASP runtime (s)	t_{config} (s)	t_{sw} (s)	t_{hw} (s)	t_{comm} (s)	t_{total} (s)	Speedup	Matrix downloads
Hole6	133 × 42	0.14	0.35	0.0099	0.0213	0.0043	0.3855	0.363	4
Hole7	204 × 56	4.31		0.1233	0.1482	0.0307	0.6522	6.608	28
Hole8	297 × 72	51.574		1.1956	1.1884	0.2490	2.983	17.289	224
Hole9	415 × 90	531.961		11.5529	10.6749	2.2802	24.858	21.399	2016
Hole10	561 ×110	5685.6		122.9934	106.7264	23.2562	253.326	22.444	20160

TABLE IV
EXPERIMENTAL RESULTS FOR THE PIGEON HOLE PROBLEM (SAT SOLVER *SOFT/C256*)

Instance	$m \times n$	GRASP runtime (s)	t_{config} (s)	t_{sw} (s)	t_{hw} (s)	t_{comm} (s)	t_{total} (s)	Speedup	Matrix downloads
Hole6	133 × 42	0.14	0.37	0.0016	0.0131	0.0045	0.3892	0.359	1
Hole7	204 × 56	4.31		0.0026	0.0196	0.0054	0.3976	10.840	1
Hole8	297 × 72	51.574		0.154	0.2855	0.0785	0.888	58.079	14
Hole9	415 × 90	531.961		1.6741	2.5706	0.7433	5.358	99.284	126
Hole10	561 ×110	5685.6		17.7909	25.6979	7.2502	51.109	111.245	1260

task with the aid of three architectures that include a software part and one of the implemented circuits $(c64, c128, c256)$. We will refer to these software/reconfigurable hardware implementations as $\mathrm{soft}/c64, \mathrm{soft}/c128$ and $\mathrm{soft}/c256$.

If the initial matrix dimensions for the considered problem instance exceed the capacity of the hardware satisfier then the respective task will be treated at the beginning in software that realizes the same algorithm as hardware. As soon as an intermediate matrix will satisfy the predefined constraints of the hardware SAT solver, it will be transferred to the FPGA and processed there. The right-hand columns in Tables II, –IV show how many times the hardware was activated for each problem. Obviously, the bigger the predefined parameters $m_{\max}$ and $n_{\max}$ are, the larger the part of the decision tree handled in hardware (see Fig. 7) and, as a result, the fewer times it is necessary to transfer the matrix data to the FPGA.

In our case the total time required to solve a problem is equal to:

$$t_{\mathrm{total}} = t_{\mathrm{config}} + t_{\mathrm{sw}} + t_{\mathrm{hw}} + t_{\mathrm{comm}}.$$

The FPGA configuration time t_{config} ranges from 0.31 s for $c64$ to 0.37 s for $c256$ and starting from the instance $\mathrm{hole8}$ it becomes negligible compared with the execution time. It should be noted that the value t_{config} could be omitted altogether from consideration since the FPGA must be configured only once after

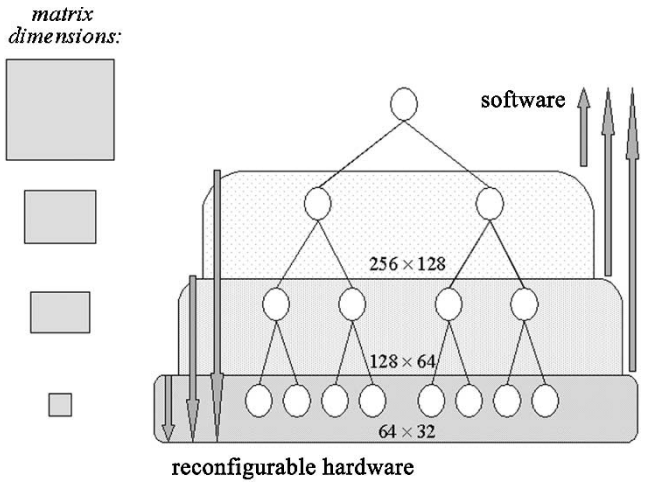

Fig. 7. Processing of the decision tree in software and in the reconfigurable hardware.

which it can be used for a series of problems without the need for reconfiguration. The value t_{sw} is the time for solving a part of the problem in software. The value t_{hw} is the time for solving a part of the problem in the reconfigurable hardware SAT solver. The software part for all the experiments was executed on an AMD Athlon/1 GHz/256 MB running Windows2000 and the hardware part was performed in an ADM-XRC board attached

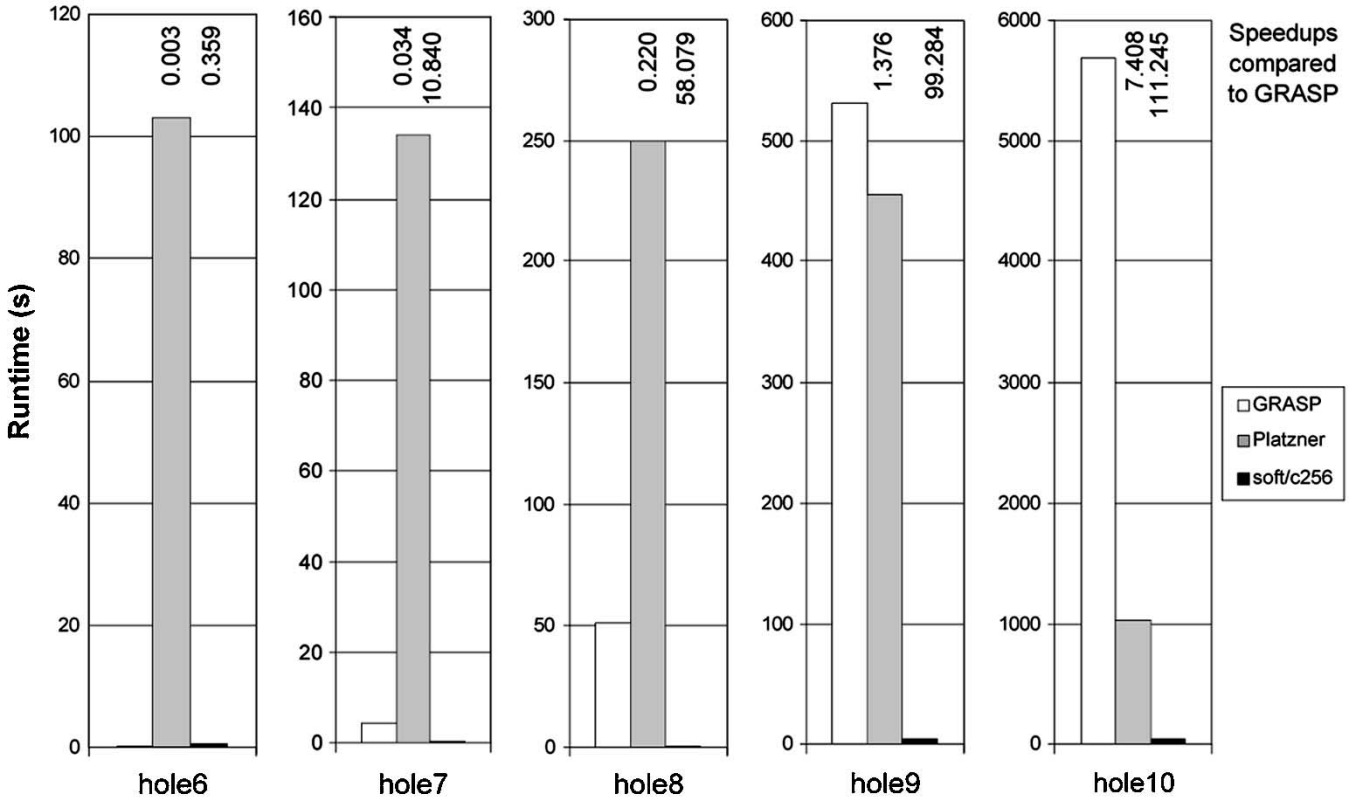

Fig. 8. Comparison with Platzner's hardware SAT satisfier [18].

TABLE V
EXPERIMENTAL RESULTS FOR BENCHMARKS AVAILABLE FROM [22] (SAT SOLVER $\mathrm{soft}/c256$)

Instance	$m \times n$	GRASP runtime (s)	t_{total} (s) (without t_{config})	Speedup
aim-100-3_4-yes1-4	340 × 100	0.03	0.08274	0.36
dubois20	160 × 60	0.02	0.01151	1.74
flat30-100	300 × 90	0.01	0.02290	0.44
ii8a1	186 × 66	0.01	0.00914	1.09
jnh1	850 × 100	0.08	0.04868	1.64
par8-4-c	266 × 67	0.02	0.01912	1.05
uf20-010	91 × 20	0.01	0.00504	1.98
2bitcomp_5	310 × 125	0.02	0.02061	0.97

to the host computer via the PCI bus. The value t_{comm} is the time spent in communications between the host computer and the FPGA. It includes the time required to transfer the matrix data to the FPGA, and to receive the result from the FPGA.

We did a comparison of our results with GRASP [11], which is one of the most known software SAT satisfiers. GRASP was executed on the same platform (i.e., on an AMD Athlon/1 GHz/256 MB) with the options $+bD + d\mathrm{DLIS} + S500$ for $\mathrm{hole6}$-$\mathrm{hole9}$ instances and $+bD + d\mathrm{DLIS} + S5000 + T5000$ for $\mathrm{hole10}$ instance. The speedup resulting from our approach is given by $t_{\mathrm{GRASP}}/t_{\mathrm{total}}$. Since the considered hardware implementation was specially designed for bit-level operations on data with the required size, these operations were performed much faster than in software. As a result, the speedup of the proposed software/hardware implementation compared to GRASP grows with the increase of the predefined matrix dimensions in the FPGA.

Fig. 8 shows a comparison of the proposed $\mathrm{soft}/c256$ SAT solver with the reconfigurable hardware satisfier described by Platzner *et al.* in [18]. The latter circuit runs at 20 MHz and the total problem solving time includes in this case the hardware compilation time (which dominates in all the instances considered) and the hardware execution time. These values together with a speedup of Platzner's implementation compared to GRASP were taken from [18]. However in [18] the software SAT solver GRASP was executed on a Pentium-II/300 MHz/128 MB running Linux. We think that it is very problematic to present a more exact comparison because with changing software platform the hardware compilation time of Platzner's SAT solver should also be changed.

Results achieved with one benchmark cannot be considered as representative. Hence a number of experiments have been conducted on some other benchmarks from [22]. The respective results obtained with the aid of the architecture $\mathrm{soft}/c256$ are presented in Table V. The meanings of the column names are the same as for Tables II–IV. However, in Table V the value t_{config} is omitted because the FPGA is configured with the circuit $c256$ only once and the configuration is reused for all the

problem instances. As it can be seen from Table V the considered benchmarks are easily solved by GRASP. Consequently, our approach does not provide useful speedups for problems that can be solved in fractions of a second by a software application. Essentially, this can be explained by the two following reasons. First, GRASP is based on a more advanced algorithm and includes many sophisticated techniques such as nonchronological backtracking and dynamic clause addition [11]. The complexity of the XCV812E FPGA allows some of these techniques to be implemented in our hardware SAT solver, which should result in further performance improvements. The second reason is that for easy problems the communication overhead turns out to be significant. However, it is possible to enlarge the size of matrices that can be handled in hardware since the capacity of current FPGAs is increasing very rapidly. Thus the communication time between the FPGA and the host computer can be reduced.

VIII. Conclusion

An architecture for a SAT solver that is based on a rational collaboration between software and reconfigurable hardware has been presented. The sequence of hardware operations is managed by a central control unit using a stack memory to support the backtracking process. The proposed technique allows instance-specific hardware compilation to be avoided and permits SAT problems to be solved that are larger than the available capacity of the reconfigurable hardware platform. As a result, a significant speedup has been achieved for the *Pigeon hole* problem compared to the state-of-the-art software SAT solver, GRASP. This considerable acceleration can be explained by the following primary reasons. Firstly, the algorithm requires execution of quite simple operations over multiple regular data and these operations are not well supported by conventional ALUs. In order to realize these operations, custom functional units have been designed and their architectures were specially optimized for the SAT problem. Secondly, memory in general-purpose computers is organized as a collection of fixed size words. The data structures for the SAT problem do not fit exactly into one word. Thus multiple memory accesses are required even to execute a simple operation. On the other hand, in the proposed architecture the memory organization is tailored to the required data size.

It should be noted that the speedups achieved by the considered SAT solver 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. However, we expect to achieve further improvements in the results in future work based on the proposed technique and potential architectural advances. One of the interesting possibilities is to implement nonchronological backtracking in hardware since this would allow large regions of the decision tree to be pruned away, thus speeding up the search for a solution.

Acknowledgment

The authors would like to thank Ivor Horton for his valuable comments and suggestions.

References

[1] G. Micheli, *Synthesis and Optimization of Digital Circuits*: McGraw-Hill, 1994.
[2] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1167–1176, Sept. 1996.
[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
[4] D. Abramson, P. Logothetis, A. Postula, and M. Randall, "FPGA based custom computing machines for irregular problems," in *Proc. 4th Int. Symp. High-Performance Computer Architecture (HPCA)*, Las Vegas, NV, Feb. 1998.
[5] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving satisfiability problems using reconfigurable computing," *IEEE Trans. VLSI Syst.*, vol. 9, pp. 109–116, Feb. 2001.
[6] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Solving Boolean satisfiability with dynamic hardware configurations," in *Field-Programmable Logic: From FPGAs to Computing Paradigm*, R. W. Hartenstein and A. Keevallik, Eds, Berlin, Germany: Springer-Verlag, Aug./Sept. 1998, pp. 326–335.
[7] M. Abramovici and J. T. de Sousa, "A SAT solver using reconfigurable hardware and virtual logic," *J. Automated Reasoning*, vol. 24, no. 1-2, pp. 5–36, Feb. 2000.
[8] P. Zhong, "Using Configurable Computing to Accelerate Boolean Satisfiability," Ph.D. dissertation, Dept. Elect. Eng., Princeton Univ., Princeton, NJ, June 1999.
[9] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the satisfiability (SAT) problem: A survey," *DIMACS Series Discrete Math. Theoretical Comput. Sci.*, vol. 35, pp. 19–151, 1997.
[10] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Commun. ACM*, no. 5, pp. 394–397, 1962.
[11] J. M. Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, pp. 506–521, May 1999.
[12] R. J. Bayardo, Jr. and R. C. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," in *Proc. 14th National Conf. Artificial Intelligence*, 1997, pp. 203–208.
[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. 38th Design Automation Conf.*, June 2001, pp. 530–535.
[14] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Proc. Design, Automation Test Europe Conf.*, 2002, pp. 142–149.
[15] 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," in *Proc. Design Automation Conf.*, 1998, pp. 194–199.
[16] M. Abramovici and D. Saab, "Satisfiability on reconfigurable hardware," in *Proc. 7th Int. Workshop Field-Programmable Logic Applications*, 1997, pp. 448–456.
[17] M. Platzner and G. De Micheli, "Acceleration of satisfiability algorithms by reconfigurable hardware," in *Field-Programmable Logic: From FPGA's to Computing Paradigm*, R. W. Hartenstein and A. Keevallik, Eds, Berlin, Germany: Springer-Verlag, Aug./Sept. 1998, pp. 69–78.
[18] O. Mencer and M. Platzner, "Dynamic circuit generation for Boolean satisfiability in an object-oriented design environment," in *Proc. 32nd Hawaii Int. Conf. System Sciences (HICSS)*, Maui, HI, Jan. 5–8, 1999.
[19] T. Suyama, M. Yokoo, and H. Sawada, "Solving satisfiability problems using logic synthesis and reconfigurable hardware," in *Proc. 31st Hawaii Int. Conf. System Sciences*, vol. 7, 1998, pp. 179–186.
[20] M. Boyd and T. Larrabee, "ELVIS—A scalable, loadable custom programmable logic device for solving Boolean satisfiability problems," in *Proc. 8th IEEE Int. Symp. Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 16–19, 2000.
[21] J. de Sousa, J. P. Marques-Silva, and M. Abramovici, "A configware/software approach to SAT solving," in *Proc. 9th IEEE Int. Symp. Field-Programmable Custom Computing Machines*, 2001.
[22] DIMACS challenge benchmarks [Online]. Available: http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html
[23] A. D. Zakrevski, *Logical Synthesis of Cascade Networks*, Moscow, Russia: Science, 1981. in Russian.
[24] Alpha Data [Online]. Available: http://www.alpha-data.com
[25] Xilinx [Online]. Available: http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp

Iouliia Skliarova received the M.Sc. degree (in computer engineering) from the Belorussian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus, in 1998, and the Ph.D. degree, in electrical engineering, from the University of Aveiro, Portugal.

She is currently an Assistant Professor in the Department of Electronics and Telecommunications, University of Aveiro. Her research interests include reconfigurable computing, application-specific architectures, computer-aided design and object-oriented programming.

António de Brito Ferrari (M'83) received the electrical engineering degree from Universidade do Porto, Portugal and Ecole Superieure d'Electricité, Paris, and the M.Sc. and Ph.D. degrees from Brunel University, Brunel, U.K.

Currently, he is a Professor of Computer Engineering, University of Aveiro, Portugal. His main research interests are in computer architecture, computer arithmetic and reconfigurable systems.