



# The design and implementation of a reconfigurable processor for problems of combinatorial computation <sup>☆</sup>

Iouliia Skliarova <sup>\*</sup>, António B. Ferrari

*Department of Electronics and Telecommunications, University of Aveiro, IEETA, 3810-193 Aveiro, Portugal*

## Abstract

The paper analyses different techniques that might be employed in order to solve various problems of combinatorial optimization and argues that the best results can be achieved by the use of software running on a general-purpose computer together with an FPGA-based reconfigurable co-processor. It suggests an architecture for a combinatorial co-processor that is based on hardware templates and consists of reconfigurable functional and control units. Finally the paper demonstrates how the co-processor can be applied to two practical applications formulated over discrete matrices, the Boolean satisfiability and covering problems.

© 2003 Elsevier B.V. All rights reserved.

*Keywords:* Reconfigurable combinatorial processor; Hardware template; Combinatorial problems

## 1. Introduction

With the advent of reconfigurable computing it becomes possible to create rapidly custom hardware implementations of various algorithms that cannot be solved efficiently using other known approaches (such as ASIC-based implementations, realization on general-purpose computers (GPCs), etc.). The effectiveness of reconfigurable circuits in general, and FPGA-based circuits in particular, depends on the application, and for certain classes

of problems they provide many advantages from the point of view of performance, the resources required, etc. [14]. This potential has been realized by many different research machines, such as DECPeRLe [37], Splash 2 [6], PRISM [4], RENCO [13], Spyder [16], etc. [25]. There are many applications where an FPGA-based system offers a very high performance solution, including DNA pattern recognition, RSA cryptography [27], the traveling salesman problem [11], long integer arithmetic [37], signal processing [9], neural networks [17], image processing [15], and the Boolean satisfiability (SAT) problem [23].

Most of the current systems that are implemented in reconfigurable hardware are data processing computations based on relatively simple algorithms. The applications that are suitable for a configurable realization usually have a large number of bit-level manipulations, which is typical

<sup>☆</sup> A preliminary version of this paper was presented at the Euromicro Symposium on Digital System Design (DSD), Warsaw, Poland in 2001.

<sup>\*</sup> Corresponding author. Tel.: +351-234-370-355x23062; fax: +351-234-370-545.

*E-mail addresses:* [iouliia@det.ua.pt](mailto:iouliia@det.ua.pt) (I. Skliarova), [ferrari@ieeta.pt](mailto:ferrari@ieeta.pt) (A.B. Ferrari).

for applications in image processing, cryptography, etc. This paper suggests applying the configurable computing technique to problems that involve rather more complex flow of control operations, in particular, problems in the area of combinatorial optimization. There are a few reconfigurable engines available for such problems [1,11,22–24,36,41,42]. They are mainly based on the idea of *instance-specific* hardware, which assumes that a specific FPGA configuration is generated for each individual problem. In this paper we propose a *domain-specific* approach that enables a variety of problems in the area of combinatorial computation to be addressed. The proposed technique is based on a reconfigurable combinatorial processor (RCP) that is going to be used for solving combinatorial tasks formulated over discrete matrices [40]. The approach we have adopted is based on special *hardware templates* (HTs) that allow execution and control units with the same architecture to be used for a variety of problems, and which do not require to be changed from one task to another. In this case, it is only necessary to configure the basic computational operations and the corresponding control algorithms for each combinatorial problem.

The paper is divided into eight sections. Section 1 is this introduction. Section 2 presents typical models and methods used in the field of combinatorial optimization. Section 3 gives an overview of possible strategies for combinatorial computations. Section 4 discusses the architecture of the proposed RCP. Experiments are presented in Section 5. Section 6 summarizes the principal ideas of software/reconfigurable hardware partitioning. Section 7 surveys related work on reconfigurable accelerators for solving combinatorial optimization problems, providing the respective comparison with the RCP. Finally, the conclusion is given in Section 8.

## 2. Models and methods of combinatorial optimization

The problems of combinatorial optimization arise in many application areas such as logic design, technical diagnostics, artificial intelligence,

etc. [20,40]. Many of these problems are known to be NP-hard [10], which means that in general, the execution time for a solution grows exponentially with the size of a problem instance. Of course, with the proposed approach, we cannot cancel out this effect of exponential growth, but we are able to delay it by enabling the primary operations of the respective algorithms to be executed more efficiently.

The exact algorithms that are employed in the area of combinatorial optimization are usually based on the generation and exhaustive examination of all possible solutions until a solution with a desired quality is found. The primary decision to be taken in this approach is how to generate the candidate solutions effectively. A widely accepted answer to this question consists of constructing a decision tree [18], which enables all possible solutions to be generated in a well-structured and efficient way. The root of the tree is considered to be the starting point that corresponds to the initial situation. The other nodes represent various situations that can be reached during the search for results. The arcs of the tree specify steps of the algorithm that have been performed. Initially, the decision tree is unknown and it is constructed during the search process.

A distinctive feature of this approach is that at each node of the decision tree the same problem is being solved. The only thing that changes from node to node is the input data. This means that the whole problem reduces to the execution of a large number of repeated operations over a periodically modified set of data.

Of course, the method of exhaustively checking all possible solutions cannot be used for the majority of practical problems because it requires such a very long execution time. That is why it is necessary to apply some optimization techniques that reduce the number of the situations that need to be considered. In order to speed up the discovering of the results various tree-pruning techniques are applied. Usually the pruning process is based on erasing repeated variants and on avoiding feasible solutions that have a cost higher than the cost of any solution already found. Sometimes it is possible to apply problem-specific methods that allow big portions of the decision tree to be

pruned by exploiting instance-specific information that can be obtained during the search. A good example of this technique is non-chronological backtracking that is widely used in the state-of-the-art SAT solvers [21,28].

The other known method of improving the effectiveness of the search is a reduction [40], which permits the current situation to be replaced with some new simpler situation without sacrificing any feasible solution. As a result, the number of computations that are required for the analysis of situations resulting from the current algorithmic step can be reduced. Returning to the example of the Boolean satisfiability problem, the well-known unit-clause rule is a good illustration of this technique [12].

However, a reduction is not possible for all existing situations. In this case another method is used that relies on the divide-and-conquer strategy [19]. This applies to critical situations that have to be divided into several simpler situations such that each of them has to be examined. The objective is to find the minimum number of such variants. Very often these new situations can be ordered according to some criteria. Considering such preliminary ordered variants essentially increases the effectiveness of computations.

It is not always possible to find the optimal solution for a number of practical combinatorial problems in a reasonable time with the available computational resources. In these cases approximate algorithms are widely used [19]. It should be noted that many of the approximate algorithms that have been developed for problems of combinatorial optimization also rely on the construction of a decision tree [40]. These algorithms try to reduce the number of variants to be considered by eliminating less promising branches of the decision tree. However, in this procedure the optimal result can be lost. The quality of the results and the computation time define the effectiveness of approximate methods.

There are many formal mathematical models such as sets, graphs, matrices, logic functions, etc. that are typically used in order to describe combinatorial tasks. As a rule any of these representations can be converted into any other [31]. 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 combinatorial processor that we are going to suggest needs to be reconfigurable (in order to be able to handle a variety of problems and problem instances) and the matrix model is more suited to this purpose because it significantly eases the process of reconfiguration and helps to minimize the reconfiguration time.

### 3. Strategies of combinatorial computations

Let us analyze the various possible ways in which combinatorial computations might be carried out. Basically, there are three different approaches that can be adopted. The first is based on the design and implementation of an ASIC that is able to solve a chosen combinatorial problem. In this case we are likely to achieve excellent results (in terms of performance) because the functional and control units that are required can be optimized for the selected problem. However, if it becomes necessary to solve a different problem, or even to make a small change in the algorithm employed, all the original design steps have to be repeated. Thus this approach is totally inflexible. Moreover, ASICs have very high development costs that would only be recovered in the case of large volume production.

The second possible approach is to use a GPC and to develop a software program that solves the required combinatorial problems. Compared to the first approach this is very flexible: any algorithmic change can easily be incorporated into the program code. The problem here is performance. The architecture of a GPC is not tailored to the specific domain of combinatorial computations and applications must always be programmed using the instructions from the fixed set that the machine provides.

The last approach to be analyzed relies on reconfigurable hardware. This is considered to be more advantageous because it allows the benefits, such as flexibility and speed, of both ASICs and GPCs to be combined, and their weaknesses to be eliminated. Of course, in similar technology,

FPGAs (reconfigurable hardware is usually constructed with the aid of SRAM-based FPGAs) suffer a speed penalty of at least one order of magnitude compared to ASICs. However, FPGAs possess the flexibility and low development cost of software implementations. To enable an implementation based on reconfigurable hardware to outperform the equivalent software implementation, the following techniques are usually employed. First, primary functional units are constructed in such a way that they are optimized for particular operations, thus requiring fewer clock cycles [2]. Second, the techniques of parallel processing and pipelining are employed. And finally, the memory organization is tailored to specific data sizes, thus speeding up data transfer.

It should be noted that in general it is difficult to realize practical combinatorial algorithms efficiently entirely in an FPGA. This is because reconfigurable logic is not so well suited for some computations. For instance, floating point arithmetic can be realized more efficiently in custom mathematical co-processors. In addition, some fragments of combinatorial algorithms are activated rarely and we can predict that the effectiveness of an FPGA-based solution will be low for such fragments. A GPC is more appropriate for realizing such irregular computations. On the other hand, FPGAs are more suited to regular (repeated) processing of a large volume of data [14]. Thus the best result can be achieved by a combination of GPC and FPGA resources. Sub-routines that can benefit from a hardware implementation are mapped to the FPGA, while others are computed by the GPC. Besides, in general it is not possible to solve any problem instance in FPGA because the reconfigurable hardware resources are always limited. A more detailed consideration of this matter is given in Section 6.

Combinatorial optimization problems have a huge number of varieties so it is difficult to design a universal accelerator that would allow for their efficient solution. Indeed it would be very problematical to find a reasonable compromise between the complexity of such an accelerator and the redundancy of its components [30]. The architecture of the RCP has to be *reconfigurable* to allow a wide variety of combinatorial problems to be solved.

The reconfigurability feature is of great importance because the number of different primary operations required to support all relevant algorithms is huge, but any particular algorithm involves just a very limited number of them [22,30].

Within the domain of configurable computing, we can distinguish between two modes of configurability: static and dynamic [26]. Static reconfiguration assumes permanent functionality after the configuration has been loaded. In fact it does not provide much flexibility but permits performance to be increased by using hardware that is optimized for a given application. Dynamic reconfiguration allows the functionality of the system to be changed during the execution of an application. Dynamic reconfiguration can in turn be *partial or global*. Global reconfiguration reserves all the hardware resources for each step of execution. After a step has been concluded, the device may be reprogrammed for the next step. Partial reconfiguration implies the selective modification of hardware resources. This opportunity allows the hardware to be adapted to better suit the actual needs of the application. Since only selected portions are reconfigured, the configuration overhead is smaller than in the previous case.

A variety of reprogrammable devices can be used to carry out dynamic reconfiguration. Single-context devices require complete reprogramming in order to introduce even a small change. *Multi-context* devices possess various planes of configuration information with just one of them active at any given moment. The main advantage of such devices is the ability to switch the context very quickly. They also support background configuration thus allowing a selected context to be reprogrammed while another one is active. *Partially reconfigurable* devices permit small portions of their resources to be modified without disturbing the remaining parts. Although this kind of device (such as the XC6200 family of Xilinx) was used in some architectures intended to accelerate the solution of combinatorial problems [1], the potential for partial reconfigurability has not been explored.

It should be noted that although the majority of commercially available FPGAs are single-context, there is a technique (based on HTs) that allows partial dynamic reconfiguration to take place. The

primary idea of a HT is to construct a parameterizable computational unit in such a way that it includes components that have both changeable and non-changeable functionality with fixed connections between them [30]. Customizing the unit is achieved by configuring the components that have alterable functionality, and the number of these components is kept to a minimum. This allows partial reconfigurability to be used and fundamentally reduces the configuration overhead.

#### 4. Design and implementation of RCP

On the basis of the analysis of the mathematical models that are used in combinatorial optimization [31], and the primary operations and basic techniques for combinatorial algorithms over discrete matrices [30], the following requirements for the RCP have been formulated.

Due to the heterogeneity of combinatorial tasks, the RCP must be dynamically reconfigurable. In other words we must be able to modify the processor's functionality at run time. In order to reduce the reconfiguration time, the RCP has to be based on a HT. In this case only the basic computational operations and the corresponding control algorithms can be altered. All the other components and connections between them will not be changed. In order to do this we propose using a RAM-based HT in such a way that customizing the functionality of the RCP is achieved by reloading RAM-based blocks.

Consequently, the RCP has to be built on the basis of an FPGA with distributed memory cells (e.g. LUTs) or embedded memory blocks, such as the Xilinx XC4000 and Virtex families. The distributed cells grouped in blocks of required sizes can be used to store matrices, much like the way the general-purpose registers of a GPC store operands and the results of intermediate computations. The embedded memory blocks can also be used to implement the components of a HT with modifiable functionality. The reconfiguration is supported by auxiliary circuits that control the reloading of the RAM-based blocks.

Fig. 1 depicts the structure of a RCP that satisfies the requirements considered above. The RCP

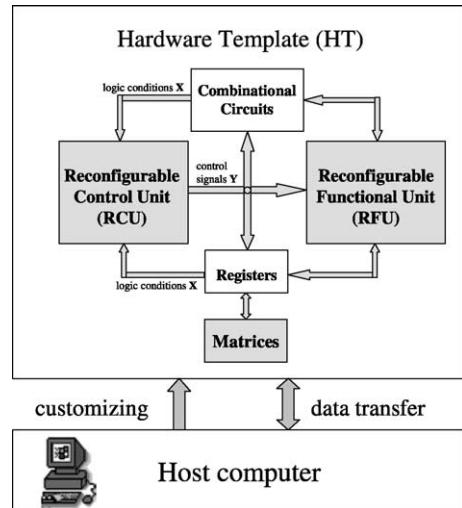


Fig. 1. The structure of RCP.

in Fig. 1 consists of two major parts: a reconfigurable control unit (RCU), and a reconfigurable functional unit (RFU). The shaded blocks in Fig. 1 can be customized for a particular application. The remaining blocks possess fixed functionality and perform common tasks that are shared by many different combinatorial problems.

Let us describe each reconfigurable block from Fig. 1 in more detail.

##### 4.1. Reprogrammable matrices

The *Matrices* block can store up to three logical matrices with dimensions  $m \times n$ . For each logical matrix,  $\mathbf{U}$ , two physical copies are constructed, the first of which stores the original matrix,  $\mathbf{U}$ , and the second stores its transpose,  $\mathbf{U}^T$ . Of course, such an approach requires double the resources needed to store just the matrix data. However, the RCP performance is improved significantly because each row and column can be read in just one clock cycle. Each physical matrix is composed of two blocks of equal dimensions:  $U\_ones$  and  $U\_zeros$ . The blocks  $U\_ones$  ( $U^T\_ones$ ) contain 1s in the positions where the original logical matrix  $\mathbf{U}$  ( $\mathbf{U}^T$ ) has 1s and contain 0s in all the other positions. Correspondingly, the blocks  $U\_zeros$  ( $U^T\_zeros$ ) hold 1s only in the positions in which matrix  $\mathbf{U}$  ( $\mathbf{U}^T$ ) has 0s. As a result, each element  $u_{ij}$ ,

$i = 1, \dots, m, j = 1, \dots, n$ , of the matrix  $\mathbf{U}$  is encoded as follows:

- if  $u_{ij} = '1'$ , then  $U\_ones[i][j] = '1'$ ,  $U\_zeros[i][j] = '0'$ ,  $U^T\_ones[j][i] = '1'$ , and  $U^T\_zeros[j][i] = '0'$ .
- if  $u_{ij} = '0'$ , then  $U\_ones[i][j] = '0'$ ,  $U\_zeros[i][j] = '1'$ ,  $U^T\_ones[j][i] = '0'$ , and  $U^T\_zeros[j][i] = '1'$ .
- if  $u_{ij} = '-'$ , then  $U\_ones[i][j] = '0'$ ,  $U\_zeros[i][j] = '0'$ ,  $U^T\_ones[j][i] = '0'$ , and  $U^T\_zeros[j][i] = '0'$ .

Fig. 2 illustrates the encoding for the following matrix:

$$\mathbf{U} = \begin{bmatrix} - & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & - & \mathbf{1} \end{bmatrix}$$

Before execution the relevant matrix data are transferred to the *Matrices* block. When some other problem instance is to be solved, the block can easily be loaded with different data.

#### 4.2. Reconfigurable control unit

The RCU implements the control algorithms that are required. The unit is modeled by a finite state machine (FSM) with dynamically modifiable behavior that generates the sequence of operations for the combinatorial algorithm being executed. An FSM might be presented at the structural level as a composition of a combinational circuit that calculates the next states and outputs, and a register that stores the current state (see Fig. 3).

We are considering a so-called RAM-based FSM, i.e. an FSM for which the combinational circuit is constructed from RAM-based blocks

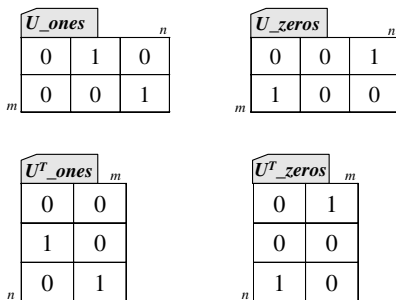


Fig. 2. Example of representing a ternary matrix in FPGA memory blocks.

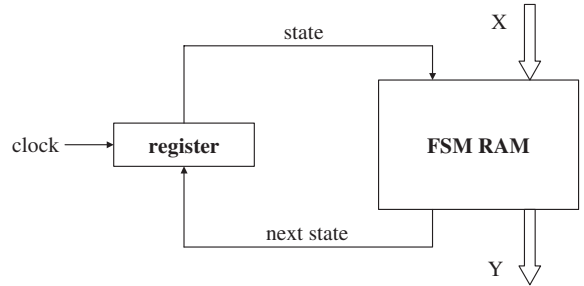


Fig. 3. A trivial structure of RAM-based FSM.

[32]. Any state code from the register is combined with input variables (i.e. logic conditions from the set  $X$ ) and the result forms an address in the FSM RAM. Each address accesses a word in the FSM RAM that contains both the code for the corresponding next state and the outputs (i.e. the control signals from the set  $Y$ ). Such an architecture is easily reprogrammable since reloading the contents of the FSM RAM changes the functionality of the RCU. If  $R$  is the minimum number of bits needed for state codes, then the size of the FSM RAM is  $2^{(R+L)} \times (R + N)$ , where  $L$  is the number of inputs from the set  $X$  and  $N$  is the number of outputs from the set  $Y$ . For sufficiently large  $L$  the size of the FSM RAM becomes quite large. Thus the principal drawback of this approach is that it is resource consuming and only allows very simple control algorithms to be implemented.

In order to reduce the size requirements of the RCU, a special state encoding technique was employed that allows the functional dependency of outputs on inputs to be reduced [35]. The technique relies on combining the inputs from the set  $X$  with state codes, enabling the depth of the FSM RAM to be reduced to  $2^{R'}$  where  $R \leq R' \leq (R + L)$ . As a result, the size of the FSM RAM becomes  $2^{R'} \times (R' + N)$ . Experiments have shown that for many practical applications  $R' \rightarrow R$  and usually  $R \leq R' \leq (R + 1)$  [35].

To support reprogrammability, the RCU is decomposed into several RAM-based sub-blocks corresponding to the architecture presented in Fig. 4 (which implements the Moore FSM model). The block *Y RAM* produces outputs on the basis of state codes. The block *A RAM* provides con-

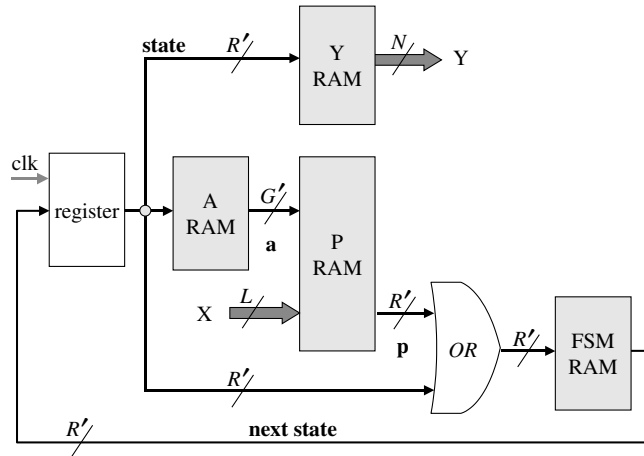


Fig. 4. The proposed architecture of the RCU.

ditional state transitions, i.e. transitions that are caused by some input variables from the set  $X$ . For all unconditional state transitions the output vector  $\mathbf{a}$  is identically equal to 0. The block  $P$  RAM calculates the vector  $\mathbf{p}$  that depends on inputs from the set  $X$  and on the vector  $\mathbf{a}$ . Finally, the block  $FSM$  RAM produces the next state of the FSM. A more detailed description of the decomposition and encoding can be found in [32,35].

As a result, the RCU is based on a parameterizable HT that has some predefined constraints. These constraints restrict the sizes of the respective RAM-based blocks, and consequently determine the maximum number of FSM states, inputs, outputs, and conditional state transitions that can be accommodated [32]. It is very easy to reprogram this device. For such purposes it is sufficient to reload the contents of the RAM-based blocks. We have developed special software tools [32] that allow a given behavioral specification of the control algorithms to be translated into the RAM contents, assuming that the RCU is based on the predefined HT. The results of experiments [32] have shown that the proposed realization of the RCU requires less area than circuits generated by the Xilinx Foundation Software from the same specification.

### 4.3. Reconfigurable functional unit

The RFU is composed of memory elements and circuits needed to store and process the variables

of the algorithm. Basic computations over columns and rows of discrete matrices are executed in the reconfigurable core shown in Fig. 5. The core is composed of a number of RAM-based blocks (the number of blocks,  $k$ , is equal to  $\max(m, n)$ ). Each block performs an operation over one or two 2-bit values and calculates a 2-bit result. The first bit comes from one of the blocks  $U_{ones}$  or  $U^T_{ones}$ , and the second bit is from the blocks  $U_{zeros}$  or  $U^T_{zeros}$ .

In order to implement different operations, it is necessary to reload the appropriate group of RAM-based blocks. Three groups of operations

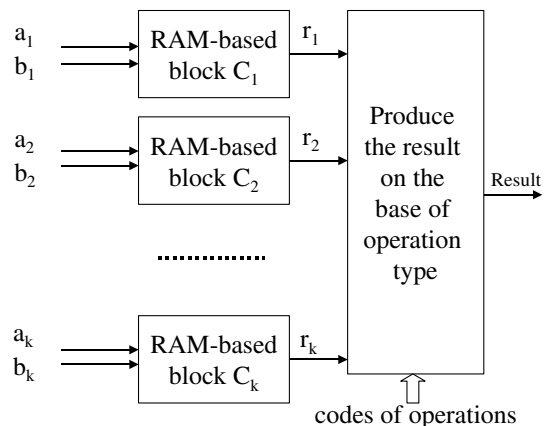


Fig. 5. The reconfigurable core performs operations over rows and columns of matrices.

have been proposed: Boolean operations, such as  $\mathbf{a} \wedge \mathbf{b}$ ; operations that require an answer in the form YES/NO, for example “test if  $\mathbf{a}$  is orthogonal to  $\mathbf{b}$ ”; and counting operations, such as calculating the number of zeros in a Boolean vector [29,30].

#### 4.4. Implementation of the RCP

Two variants of the RCP have been designed, implemented, and tested. The first variant was implemented based on the XStend board from XESS [38] containing one Xilinx XC4010XL FPGA. The reprogrammable blocks of the RCP were constructed from LUTs available in this FPGA. The reconfiguration was carried out through the parallel port. Since the FPGA employed has very restricted resources and reconfiguration via a parallel interface does not provide much flexibility, this implementation has only been used for verification purposes and for some experiments. In this section we present the results achieved with this approach.

The second variant of RCP architecture that is more application-oriented was implemented subsequently using the ADM-XRC PCI board [3]. This board contains one XCV812E Virtex Extended Memory FPGA with approximately 254K logic gates and embedded memory blocks that provide for a total capacity of more than 1 Mbits [39]. 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. The respective implementation is tailored to the Boolean satisfiability problem and all the details can be found in [33,34].

For configuring the FPGA, the following model has been proposed (see Fig. 6). Basic functions of combinatorial algorithms (for instance, *find-max-column*, *find-ort-row*, etc.) are included in a parameterized library. The system level specification is prepared in the C++ programming language, i.e. a combinatorial algorithm is described in C++ using the library mentioned above. The assisting software tools extract the corresponding configuration from the library and download it to the FPGA when required. Note that the basic HT is loaded from the very beginning, so it is only necessary to reprogram the alterable components in the RCU and RFU. Finally the matrix data are passed to the FPGA for processing. When the computation has been completed, the assisting software tools will store the intermediate results and program execution will proceed, eventually reaching another hardware library function forcing a similar sequence of actions, i.e. the process considered above will be repeated.

## 5. Experiments

Two main applications were used to test the RCP. These are the Boolean satisfiability problem and the covering problem.

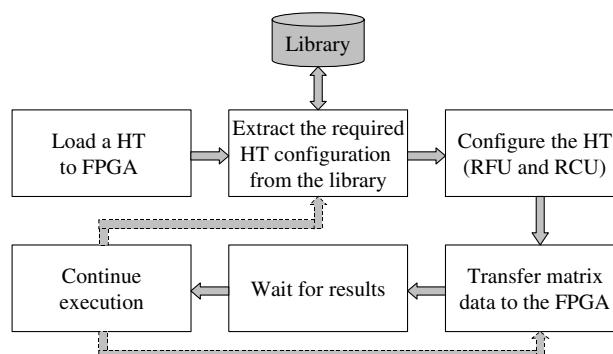


Fig. 6. Configuring the RCP.

5.1. The Boolean satisfiability problem

The Boolean satisfiability problem involves determining if a formula presented in conjunctive normal form (CNF) is satisfied by some truth assignment. The search variant of this problem requires at least one satisfying assignment to be found. A CNF consists of a conjunction of a number of clauses, where a clause is a disjunction of one or more variables or their negations. The SAT problem has great importance in the area of computer-aided hardware optimization.

For example the following formula contains four variables and three clauses, and is satisfied when  $x_1 = '0'$ ,  $x_2 = '0'$ ,  $x_3 = '1'$  and  $x_4 = '1'$ :

$$(\bar{x}_1 \vee x_4)(x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

In order to solve this problem with the aid of the RCP, we first have to formulate it over a ternary matrix  $U$ . Let us set a correspondence between the variables and clauses of the formula, and the columns and rows of  $U$ . Each element  $u_{ij}$ ,  $i = 1, \dots, m, j = 1, \dots, n$ , of the matrix is equal to:

- '0'—if variable  $x_j$  is included in clause  $c_i$  with negation;
- '1'—if variable  $x_j$  is included in clause  $c_i$  without negation;
- '-' (don't care)—if variable  $x_j$  is not included in clause  $c_i$ .

Taking into account these rules, the matrix  $U$  for the formula considered above can be presented in the following form:

$$U = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ \mathbf{0} & - & - & \mathbf{1} \\ - & - & \mathbf{1} & - \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & - \end{bmatrix} \begin{matrix} c_1 \\ c_2 \\ c_3 \end{matrix}$$

We have implemented a deterministic solution to the SAT problem that exhaustively generates and examines all possible assignments of values to variables. It should be noted that such a brute-force approach is not very efficient. However, it can be used to estimate the effectiveness of the proposed architecture. The more competent solution to this problem can be found in [33,34].

The algorithm employed has been described by the graph-scheme (GS) [5] depicted in Fig. 7. The problem can be solved using the following sequence of actions. First we have to construct the matrix  $U$ , load the control algorithm from Fig. 7 into the RCU, and configure the RFU to implement the orthogonality checking operation.

According to the algorithm we have to find a Boolean vector  $w$  that is orthogonal to all rows of the matrix  $U$ . Two ternary vectors  $a = [a_1 a_2 \dots a_n]$  and  $b = [b_1 b_2 \dots b_n]$  are orthogonal if there exist  $j = 1, \dots, n$ , such that either  $a_j = '0'$  and  $b_j = '1'$ , or  $a_j = '1'$  and  $b_j = '0'$ . If such vector cannot be found then the problem is unsatisfiable. In the opposite case the negated vector  $w$  gives the solution, i.e. all its elements having value '0' point to variables, which must be equal to '1', and all its elements with value '1' point to variables, which must be equal to '0'.

The overall runtime for solving this problem in hardware includes in our case the time for configuring the matrix, the RCU, and the RFU, and the actual hardware execution time.

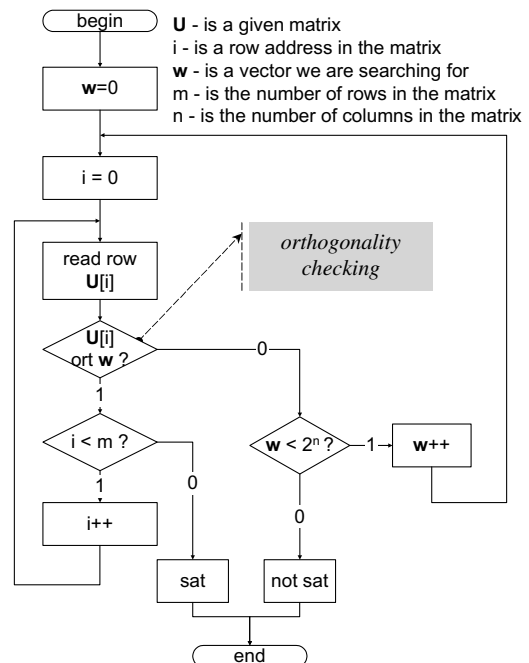


Fig. 7. A GS of the algorithm for solving the Boolean satisfiability problem.

5.2. The covering problem

Covering is a well-known combinatorial problem that has many practical applications. For a given set it requires a subset to be found that has certain properties. The optimization variant of this problem involves finding a subset of minimal cardinality. For example, let us suppose that it is necessary to determine a minimum-size vertex cover of an undirected graph. It is known that a vertex cover of a graph  $G = (V, E)$  is a subset  $V' \subseteq V$  such that if  $(u, v) \in E$ , then  $u \in V'$  or  $v \in V'$  (or both) [7].

In order to solve this problem in the RCP, we must reformulate it over a matrix. For this purpose let us build the incidence matrix,  $\mathbf{I}$ , whose columns correspond to edges of the graph and rows represent the vertices. Now in order to solve the problem, we have to find a cover of the matrix,  $\mathbf{I}$ , that is composed of the minimal number of rows having in conjunction at least one ‘1’ in each column of  $\mathbf{I}$ . The selected rows correspond to the vertices that must be included into the minimum-size vertex cover. Let us consider an example. For the graph in Fig. 8a the incidence matrix is presented in the form depicted in Fig. 8b.

In order to find a minimum-size row cover of a Boolean matrix we used the approximate algorithm described in Fig. 9 [40]. As can be seen from Fig. 9, the primary operation to be executed by the algorithm is to count the number of ones in different rows and columns of the matrix. Thus the problem can be solved by applying the following basic steps. First the matrix  $\mathbf{I}$  is constructed, then

the control algorithm is loaded into the RCU. Finally, the RFU is configured to implement a *count-number-of-ones* operation.

The discovered subset is shaded in the graph in Fig. 8a and in the matrix  $\mathbf{I}$  in Fig. 8b. In this example we have indeed found the minimum-size vertex cover of the graph but in the general case the algorithm does not guarantee that the best solution will be found.

5.3. Performance results

Table 1 contains the results obtained when we solved the two combinatorial problems considered with the aid of software running under Windows 2000 on a Pentium III-800 MHz/256 MB PC and with the RCP implemented in an XC4010XL FPGA. The rows *Cov1*...*Cov6* show results for randomly generated instances of the covering problem, and the rows *SAT1*...*SAT6*—for the satisfiability problem.

The first line in Table 1 presents an example of a trivial operation that counts the number of ones in a Boolean vector. This operation requires several clock cycles in a GPC. In the RCP we can realize it using much fewer clock cycles.

As it can be seen from Table 1, the improvement in results obtained for the covering problem compared to the software implementation of the same algorithm is quite considerable. The primary operation of the algorithm for finding the minimal row cover of a Boolean matrix is to count the number of ones in various rows and columns (see Fig. 9). As we mentioned above, this operation

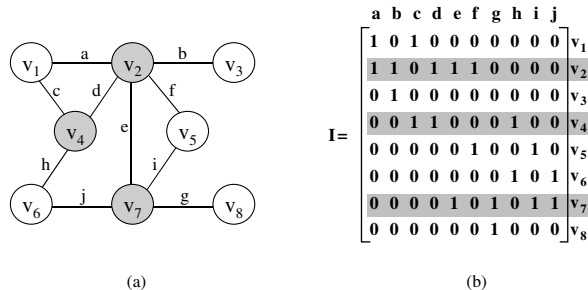


Fig. 8. A minimum-size vertex cover of a graph (a) and the corresponding incidence matrix (b).

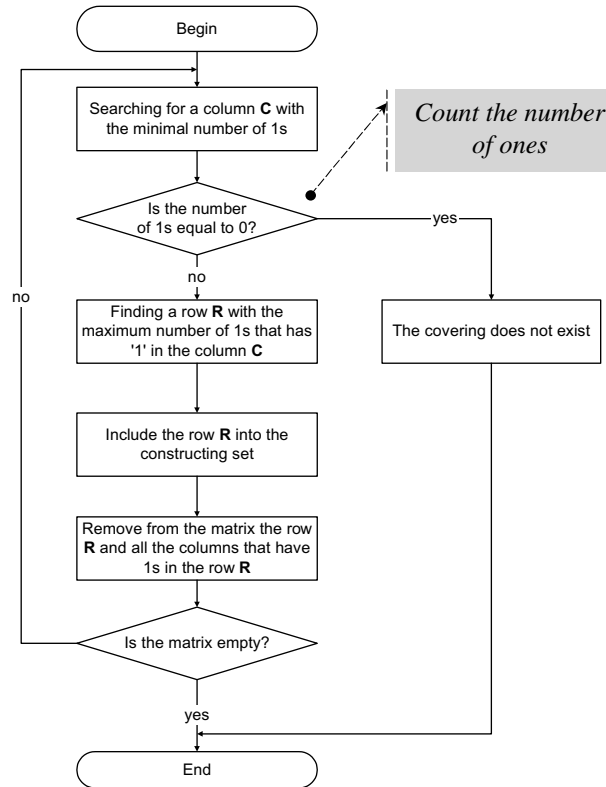


Fig. 9. An algorithm for finding the minimal row cover of a Boolean matrix.

Table 1  
Experimental results with RCP implemented in XC4010XL FPGA

Problem	Matrix/vector dimensions	Software execution time (in ms)	Hardware execution time (in ms)	Speedup
Count the number of ones in a vector	8	0.0059	0.00002487	237
Cov1		0.284	0.0110	25.82
Cov2		0.310515	0.0144	24.56
Cov3		0.217905	0.0065	33.52
Cov4		0.17684	0.00423	41.81
Cov5	8 × 8	0.292075	0.00924	31.61
Cov6		0.227125	0.00274	82.89
SAT1		4.54248	0.02971	152.89
SAT2		5.22301	0.01508	346.35
SAT3		0.24891	0.000588	423.32
SAT4		11.75764	0.07485	157.08
SAT5	8 × 8	2.72465	0.008059	338.09
SAT6		0.51543	0.002647	194.72

executes in the RCP much faster than in software. However, the problem is quite control-oriented, so

the acceleration in the basic operation is not achieved overall because the other parts of the

algorithm are not executed much faster than in software. We guess that this circumstance impedes the achievement of a more significant speedup compared to the software implementation.

For the satisfiability problem we have obtained more impressive speedups. This is because the algorithm (see Fig. 7) only requires sequential reading of different rows of the matrix and checking the respective vectors for orthogonality. Each of these operations needs just one clock cycle in the RCP. In software the matrix has been constructed as an array of integers. Thus checking whether the vector  $\mathbf{w}$  (see Fig. 7) is orthogonal to any row of the matrix requires many memory accesses and calculations.

## 6. Software/reconfigurable hardware partitioning

For the examples considered (see Section 5), each task was completely solved in FPGA. Note that the proposed RCP limits the maximum dimensions of the matrix so the RCP cannot be used for dealing with an arbitrary task. With respect to the maximum matrix dimensions that are allowed, three different kinds of situations can occur:

- The problem instance is very small and simple. In this case the use of FPGA becomes unreasonable because it takes a significant time to reconfigure the HT (in our case), or to generate the hardware circuit (in an instance-specific approach). Either way, in this instance we will lose all the advantages of fast hardware and a software solution will work much faster.
- The problem instance is hard and its dimensions fit within the matrix dimensions allowed in FPGA. In this case we might possibly achieve a sufficiently good performance to offset the hardware configuration time. However, real-world problems rarely match the limited dimensions of the HT.
- The problem size is very large, far and away exceeding the supported hardware capacity. Within the domain of an instance-specific approach [1,41], the following solution is usually adopted. If the circuit cannot be implemented in a single FPGA, several FPGAs are employed by

applying special methods for multi-FPGA partitioning. Nevertheless, there is still no guarantee that a given task will be solved efficiently on the available reconfigurable hardware resources.

Because of that, we suggest the following strategy should be applied. As we have already discussed in Section 2, a common approach to solving combinatorial problems is based on a decision tree. When we construct the tree, various splitting and reduction methods are used. This enables the initial matrix dimensions to be decreased gradually (traversing one of the tree branches).

The RCP is based on a HT with predefined constraints on the maximum number of rows and columns of the matrix. We can utilize the following technique to accommodate this. First of all, the assisting software tools will configure the RFU and the RCU to execute the required algorithm. Next, if the initial matrix satisfies the predefined constraints, the matrix data will be transferred to FPGA and the problem will be completely solved in the RCP. Otherwise, the software will try to solve the problem. During this process, it will apply special splitting and reduction methods until an intermediate matrix, constructed during the current search step, is arrived at that does not exceed the capacity restrictions. From this point on, the RCP will be responsible for subsequent steps. If the reconfigurable hardware finds a solution, the problem is considered to be solved and the result will be dispatched to the host computer. On the other hand, if the current branch of the decision tree does not allow a solution to be found, control will be returned to the software. The software will then continue to traverse the decision tree, eventually reaching some other point where the matrix dimensions will fall within the constraints. The matrix data will then be transferred to FPGA and the RCP will try to solve the sub-problem. These steps will be repeated until we arrive at either a negative or positive result, i.e. we will either obtain a solution or we will conclude that the problem does not have a solution. Thus the decision tree will be treated in software and in hardware in the way that is shown in Fig. 10.

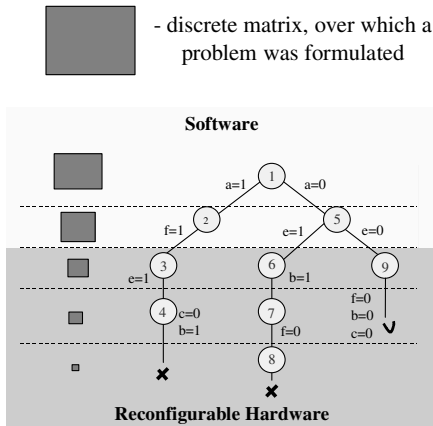


Fig. 10. Processing of the decision tree in software and in reconfigurable hardware.

## 7. Discussion and related work

Recently, several research groups have explored the possibility of accelerating the solution of combinatorial optimization problems with the aid of reconfigurable hardware. The best-investigated problems were the SAT problem [1,23,36,41,42] and the covering problem [24]. As we mentioned in the introduction, practically all proposed architectures apply an instance-specific approach.

For example, Plessl et al. proposed an architecture for an instance-specific accelerator for the minimum covering problem [24]. The suggested architecture implements a branch-and-bound algorithm in 3-valued logic. For each problem instance a VHDL description of the respective circuit is automatically generated by a specially designed software application. The resulting VHDL code is used for synthesis and implementation of the circuit with the aid of commercially available tools.

The generation of a specific FPGA configuration for each individual problem instance permits performance to be increased and provides a good utilization of available resources. The total problem solving time in this case is equal to “hardware circuit generation time” + “FPGA configuration time” + “execution time”. It should be noted that the time required to generate an instance-specific circuit is quite significant, frequently exceeding the

actual execution time, and thus canceling out all the advantages of the fast hardware implementation. For example, with the accelerator developed by Plessl et al. the hardware execution time for some problem instances that were presented in [24] ranged from 1 ms to 2 s while the synthesis and implementation of the circuit required several minutes. Consequently this method can only be used efficiently for very difficult problems for which the hardware compilation and configuration time is negligible compared to the execution time. That is why all recent efforts have been focused on avoiding instance-specific placement and routing. For such purposes, special techniques are usually employed that enable the generation of the FPGA configurations to be significantly accelerated and higher clock rates to be achieved. These techniques rely on modular design styles [1] and HTs that are automatically customized for each problem instance [42].

For example, Dandalis et al. [8] proposed an application-specific mapping approach for solving graph problems. The main objective was to eliminate an excessive hardware compilation time by reducing the need for CAD tools at the mapping stage. For each problem instance an individual circuit is generated. However, the authors [8] have designed an algorithm-specific template, referred to as skeleton, which consists of modules corresponding to basic graph elements. The skeleton can be adapted to different graph instances at run-time.

However, the HTs employed in reconfigurable accelerators are usually oriented towards just one problem, and only allow customization for various instances of the same problem. In order to address a different task, even one that is very similar, it is necessary to design a new circuit and to modify the software tools that are employed for automatic customization. As opposed to this approach, the RCP is domain-oriented and thus can be used for solving various combinatorial problems. It is a partially reconfigurable accelerator since only small portions of the control and functional units need to be programmed to match a certain algorithm. The total problem solving time for the RCP consists of three components: “the HT configuration time” + “the time spent in communications

between software and FPGA” + “the execution time”. If the problem is partitioned between software and hardware then the latter value includes two parts: the software execution time and the hardware execution time. Since the reconfiguration is partial, the configuration time is negligible compared to the execution time.

A similar approach was followed in a cube calculus machine (CCM) proposed in [22]. The CCM is a hardware accelerator reconfigurable for specific multiple-valued cube calculus operations required by a certain algorithm. For each algorithm an appropriate structure of CCM must be instantiated. The CCM is targeted at execution of the inner loop of algorithms, i.e. a loop that performs operations on cubes. For each operation the host processor loads the respective complex instruction to the CCM and then sends the relevant data cubes to the CCM and receives back the resultant cubes. In such a model of execution, the communication between the host processor and the CCM is quite intensive and can only be acceptable for tightly coupled systems. In contrast to this, the RCP executes the complete algorithm, not just the inner loop, thereby reducing the communication overhead.

Another important issue affecting any algorithm implemented in reconfigurable hardware is related to the logic capacity of the device employed, which is always limited. Thus, efficient techniques are needed to deal with the situation when a problem instance exceeds the available hardware resources. In the domain of combinatorial accelerators the following four possibilities have been explored.

The first is the expansion of the logic capacity by interconnecting a number of FPGAs and partitioning the circuit between them. It should be noted that fast and efficient multi-device partitioning and routing is quite a difficult task (of course modular and scalable design styles [22,42] can alleviate it).

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 problem instance into a set of independent sub-problems [1]. Each sub-problem 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 problem instance. As a result, for some problem instances the partitioning time may increase to unacceptable levels.

The third method is based on a virtual hardware scheme proposed in [36] for solving the SAT problem, which relies on dividing the circuit into a series of hardware pages that are successively run being the intermediate results stored in external memory blocks. Since all the hardware pages have the same structure with only a number of registers being reconfigured, the page switching is performed very fast.

The last method, described in Section 6, is based on software/hardware partitioning according to the available logic capacity of the hardware that is employed [33,34]. Of course, the efficiency of such a partitioning depends on both the problem structure and the scale of the respective implementation of the RCP. In the worst case the time spent in communications can be a significant portion of the total problem solving time. However, recent FPGAs such as Stratix from Altera and Virtex-II/Virtex-II Pro from Xilinx provide an adequate platform for full-scale implementation of the RCP and assure a more efficient software/hardware partitioning. Moreover, it is not even necessary to implement the RCP on a SRAM-based FPGA since the reconfigurability of the accelerator is limited to the number of RAM-based blocks available in the RCU and RFU. Thus the RCP can be implemented as an ASIC equipped with embedded memory blocks [22]. Nevertheless, an implementation based on an in-circuit programmable FPGA is in most cases more cost-effective since the FPGA can be used for other tasks that may be required by different applications.

## 8. Conclusion

The paper presents the results of the design and implementation of a reconfigurable processor for problems of combinatorial optimization. It proposes an architecture for a RCP based on HTs that are composed of fixed components and repro-

grammable blocks. The latter include functional and control units with dynamically modifiable behavior. The advantages of the RCP have been shown by experiments with two important combinatorial applications, the Boolean satisfiability and covering problems. Finally, we propose a computational model that allows efficient collaboration between software and reconfigurable hardware and permits the hardware capacity problem inherent to all instance-specific implementations to be partially solved.

### Acknowledgements

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

This work was supported by the Portuguese Foundation of Science and Technology under Grant No. FCT-PRAXIS XXI/BD/21353/99.

### References

- [1] M. Abramovici, J.T. de Sousa, A SAT solver using reconfigurable hardware and virtual logic, *Journal of Automated Reasoning* 24 (1–2) (2000) 5–36.
- [2] D. Abramson, A. Postula, M. Randall, FPGA based custom computing machines for irregular problems, in: *Proc. of the Fourth Int. Symposium on High-Performance Computer Architecture—HPCA98*, Las Vegas, NV, 1–4 February 1998.
- [3] Alpha Data (Online). Available from <<http://www.alpha-data.com>>.
- [4] P.M. Athanas, H.F. Silverman, Processor reconfiguration through instruction-set metamorphosis, *IEEE Computer* 26 (3) (1993) 11–18.
- [5] S. Baranov, *Logic Synthesis for Control Automata*, Kluwer Academic Publishers, 1994.
- [6] D.A. Bell, J.M. Arnold, W.J. Kleinfelder, Splash 2—FPGAs in a Custom Computing Machine, *IEEE Computer Society Press*, 1996.
- [7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1997.
- [8] A. Dandalis, A. Mei, V.K. Prasanna, Domain specific mapping for solving graph problems on reconfigurable devices, in: *Reconfigurable Architectures Workshop—RAW '99*, April 1999.
- [9] C. Dick, F. Harris, Virtual signal processors, *Microprocessors and Microsystems* 22 (1998) 135–148.
- [10] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.
- [11] P. Graham, B. Nelson, A hardware genetic algorithm for the traveling salesman problem on Splash 2, in: *Proc. 5th International Workshop on Field Programmable Logic and Applications*, Oxford, England, 1995, pp. 352–361.
- [12] J. Gu, P.W. Purdom, J. Franco, B.W. Wah, Algorithms for the satisfiability (SAT) problem: a survey, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 35 (1997) 19–151.
- [13] J.O. Haenni, J.L. Beuchat, E. Sanchez, RENCO: a reconfigurable network computer, in: *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, Napa, California, 1998, pp. 288–289.
- [14] S. Hauck, The Roles of FPGAs in Reprogrammable Systems, *Proceedings of the IEEE* 86 (4) (1998) 615–638.
- [15] S.D. Haynes, J. Stone, W. Luk, Video image processing with the Sonic architecture, *IEEE Computer* (2000) 50–57.
- [16] C. Iseli, E. Sanchez, Spyder: a reconfigurable VLIW processor using FPGAs, in: *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993, pp. 17–24.
- [17] P. Kolinummi, P. Hamalainen, T. Hamalainen, J. Saari-nen, PARNEU: general-purpose partial tree computer, *Microprocessors and Microsystems* 24 (2000) 23–42.
- [18] D.L. Kreher, D.R. Stinson, *Combinatorial Algorithms: Generation, Enumeration, and Search*, CRC Press, 1999.
- [19] Z. Michalewicz, D.B. Fogel, *How to Solve It: Modern Heuristics*, Springer-Verlag, 2000.
- [20] G. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.
- [21] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: *Proc. of the 38th Design Automation Conference*, June 2001, pp. 530–535.
- [22] M. Perkowski, D. Foote, Q. Chen, A. Al-Rabadi, L. Jozwiak, Learning hardware using multiple-valued logic—part 2: cube calculus and architecture, in: *IEEE Micro*, vol. 22, no. 3, *IEEE Computer Society Press*, Los Alamitos, CA, USA, May/June 2002, pp. 52–61.
- [23] M. Platzner, G. Micheli, Acceleration of satisfiability algorithms by reconfigurable hardware, in: *Proc. 8th Int. Workshop on Field Programmable Logic and Applications FPL'98*, Springer-Verlag, Tallin, Estonia, 1998, pp. 69–78.
- [24] C. Plessl, M. Platzner, Instance-specific accelerators for minimum covering, in: *Proc. 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, June 2001, pp. 85–91.
- [25] RC Bibliography (Online). Available from <<http://www.ife.ee.ethz.ch/~enzler/rc/bib.html>>.
- [26] E. Sanchez, M. Sipper, J.O. Haenni, J.L. Beuchat, A. Stauffer, A. Perez-Urbe, Static and dynamic configurable systems, *IEEE Transactions on Computers* 48 (6) (1999) 556–564.
- [27] M. Shand, J. Vuillemin, Fast implementation of RSA cryptography, in: *Proc. 11th IEEE Symp. Computer Arithmetic*, Canada, 1993, pp. 252–259.

- [28] J.M. Silva, K.A. Sakallah, GRASP: a search algorithm for propositional satisfiability, *IEEE Transactions on Computers* 48 (5) (1999) 506–521.
- [29] I. Skliarova, A.B. Ferrari, Exploiting FPGA-based architectures and design tools for problems of reconfigurable computations, in: *Proc. XIII Symposium on Integrated Circuits and System Design SBCCI'2000*, Brazil, September 2000, pp. 347–352.
- [30] I. Skliarova, A.B. Ferrari, Development tools for problems of combinatorial optimization, in: *Proc. 4th Portuguese Conference on Automatic Control—CONTROLO'2000*, Portugal, October 2000, pp. 552–557.
- [31] I. Skliarova, A.B. Ferrari, Modelos matemáticos e problemas de optimização combinatoria, *Electrónica e Telecomunicações* 3 (3) (2001) 202–208 (in Portuguese).
- [32] I. Skliarova, A.B. Ferrari, Synthesis of reprogrammable control unit for combinatorial processor, in: *Proc. of the 4th Int. Workshop on IEEE Design and Diagnostics of Electronic Circuits and Systems—DDECS 2001*, Gyor, Hungary, April 2001, pp. 179–186.
- [33] I. Skliarova, A.B. Ferrari, A SAT solver using software and reconfigurable hardware, in: *Proc. of the Design, Automation and Test in Europe Conference DATE'2002*, Paris, France, March 2002, p. 1094.
- [34] I. Skliarova, A.B. Ferrari, A hardware/software approach to accelerate Boolean satisfiability, in: *Proc. of IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems IEEE DDECS'2002*, Brno, Czech Republic, April 2002, pp. 270–277.
- [35] V. Sklyarov, Synthesis and implementation of RAM-based finite state machines in FPGAs, in: *Proc. of FPL'2000*, Villach, Austria, August 2000, pp. 718–728.
- [36] J. de Sousa, J.P. Marques-Silva, M. Abramovici, A configware/software approach to SAT solving, in: *Proc. of 9th IEEE Int. Symp. on Field-Programmable Custom Computing Machines*, 2001.
- [37] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, P. Boucard, Programmable active memories: reconfigurable systems come of age, *IEEE Transactions on VLSI Systems* 4 (1) (1996) 56–69.
- [38] XESS Corp. (Online). Available from <<http://www.xess.com/>>.
- [39] Xilinx, *The Programmable Logic Data Book*, Xilinx, San Jose, 2000.
- [40] A.D. Zakrevski, *Logical Synthesis of Cascade Networks*, Moscow, Science, 1981 (in Russian).
- [41] P. Zhong, M. Martonosi, P. Ashar, S. Malik, Accelerating Boolean satisfiability with configurable hardware, in: *Proc. IEEE Symposium on FPGAs for Custom Computing Machines—FCCM*, April 1998, pp. 186–195.
- [42] P. Zhong, *Using Configurable Computing to Accelerate Boolean Satisfiability*, Ph.D. dissertation, Department of Electrical Engineering, Princeton University, June 1999.



**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. She is currently a Ph.D. student at the Department of Electronics and Telecommunications of the University of Aveiro, Portugal. Her research interests include reconfigurable computing, application-specific architectures, computer-aided design and object-oriented programming.



**António de Brito Ferrari** received Electrical Engineering degrees from Universidade do Porto, Portugal and Ecole Supérieure d'Electricité, Paris, and M.Sc. and Ph.D. degrees from Brunel University, UK. 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.