Software/Configware Implementation of Combinatorial Algorithms

Iouliia Skliarova, Valery Sklyarov
University of Aveiro, Department of Electronics, Telecommunications and Informatics, IEETA, 3810-193 Aveiro, Portugal
iouliia@det.ua.pt, skl@det.ua.pt

Abstract

This paper discusses an approach for solving combinatorial problems by combining software and dynamically reconfigurable hardware (configware). The suggested technique avoids instance-specific hardware compilation and, as a result, allows obtaining higher performance than currently available pure software approaches as well as instance-specific reconfigurable solutions. Moreover, the technique permits problems to be solved that exceed the resources of the available reconfigurable hardware. The architecture of dynamically reconfigurable hardware problem solver is modeled in software allowing to estimate different characteristics, such as the time of reconfiguration, performance, etc., and to speed up the overall design process.

1. Introduction

Combinatorial problems play an important role in the computer-aided design and test of integrated circuits [1,2], artificial intelligence, embedded systems [3], etc. Because of the wide scope of practical applications, these problems have been studied extensively. Many of them are NP-complete and NP-hard, therefore the complexity of the respective algorithms makes it difficult (and sometimes even impossible) to solve a problem in a reasonable time, or with the available computational resources. As a result, many different techniques for accelerating the solution of combinatorial problems are being constantly proposed, either in software or in hardware (for example, in field-programmable gate arrays - FPGA).

Novel methods suggested in the scope of software aim at studying and applying optimized data structures, exploring alternative algorithmic approaches (such as proposing more efficient approximate algorithms based on heuristic search), etc. Accelerating the solution of combinatorial problems in reconfigurable hardware is based primarily on mapping to hardware the existing algorithms (which have already been tested exhaustively in software) and trying to execute as much work as possible in parallel. According to Amdhal's law, the gain attainable by the parallel execution is limited by the portion of the algorithm suitable for parallelization.

Let us consider backtracking search algorithms, whose general structure is depicted in Figure 1. In this case, the processes of initial problem simplification and decomposition in sub-problems can be executed by a number of processing elements working in parallel, whereas invoking the algorithm for each of the sub-problems has to be done sequentially, one by one.

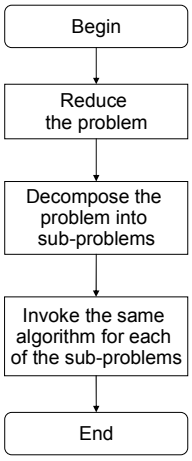

Figure 1. General structure of backtracking search algorithms

Another very significant bottleneck in executing combinatorial search algorithms, either in software or in hardware, is access to and management of problem data. In software, this issue is solved by both inventing more efficient data structures and using caches. What is attempted in reconfigurable hardware is to store as much data as possible in local memories, which are only accessed by a single processing element, and to adapt the dimensions of memory blocks to the particular requirements of each problem instance.

The majority of researches in the area of accelerating solutions with the aid of reconfigurable (based on FPGA) hardware apply an instance-specific approach, i.e. they generate a circuit for each particular problem instance to be solved [4-6]. In this case the total problem solving time is equal to "hardware circuit generation time" + "FPGA configuration time" + "actual execution time" +

"time required to communicate the results". The major advantage of this strategy is that a direct mapping of problem structure to functional components permits performance to be increased and provides a good utilization of the resources. However, the required time for hardware compilation is very significant, being frequently much higher than the actual time for the execution of combinatorial algorithms in hardware. Thus, this instance-specific method can only be used effectively for complicated problems with large volume of input data where the hardware compilation time is offset by the reduced execution time. More recent work in this direction is targeted at avoiding instance-specific layout compilation [7].

Another important issue affecting any algorithm implemented in reconfigurable hardware is related to the capacity of the hardware platform. If the circuit cannot be implemented in a single device it is possible to employ several components applying methods of multi-device partitioning. Nevertheless there is no guarantee that a given task can be efficiently solved on a given set of reconfigurable hardware resources [8]. Because of this, it was proposed to partition the problem solution between software and reconfigurable hardware. The partitioning process can be conducted in two possible ways: automated and manual.

Automated partitioning supposes that high-level general purpose programming languages are employed for problem description. Is this case the specification can be provided in standard ANSI C/C++ and the algorithm can be simulated directly using any available C/C++ debugger. After that the code portions, which have to be accelerated in an FPGA, are identified; software/hardware partitioning is executed, the respective interfaces are generated and, for the hardware part, the synthesizable HDL code is produced. The results of the last two steps are exported to implementation tools, which provide for future steps and ultimately generate the FPGA configuration file. This approach sounds to be very attractive but currently, to the best of our knowledge, there exist no efficient solutions of combinatorial problems using automated partitioning.

In the scope of manual software/configware partitioning two different methods can be envisioned: partitioning according to computational complexity and partitioning with respect to logic capacity [9]. The first of these methods is based on selecting and assigning the most computationally intensive tasks (which are well-suited to parallel execution) to hardware, while the sequential control-oriented tasks are mostly performed in software [10]. Reconfigurable systems of this type are based on the 90/10 rule, which states that 90% of the execution time of an application is spent by 10% of its code. Thus, in order to accelerate the solution of a particular combinatorial problem, a small portion of the algorithm is selected and executed in an FPGA.

The second method performs partitioning according to the available logic capacity of the FPGA [8]. In this case, if a problem instance does not "fit" to a chosen device, it has first to be decomposed by software into independent sub-problems and only the sub-problems satisfying the resource constraints are assigned to hardware being the remaining part of the problem processed in software.

Many combinatorial search problems can be efficiently formulated and solved over Boolean and ternary matrices that keep the initial and all the required intermediate data [11]. An analysis of these problems and the respective algorithms shows that each of them requires quite a limited number of different operations. However, various combinatorial problems require different sets of operations. It allows concluding the following:

- reconfigurable devices should be more profitable than the respective software solutions;
- the same computational device can be customized for solving different combinatorial problems through applying reconfiguration techniques.

This paper summarizes the recent work in the considered scope applying software/configware solutions and makes two following fundamental contributions:

- It describes the suggested architecture of a software/configware system for solving combinatorial problems and providing virtual partitioning between software and run-time reconfigurable hardware;
- It defines architecture of dynamically reconfigurable hardware problem solver and suggests tools for modeling this architecture in software, which makes possible to estimate the time of reconfiguration, performance and other important characteristics for solving a particular combinatorial problem.

The remainder of this paper is organized in 6 sections. Section 2 suggests software/configware partitioning. Section 3 discusses a reconfigurable architecture. Section 4 describes the dynamically reconfigurable units for the considered architecture, for which section 5 introduces different software models. Section 6 is devoted to experiments. The conclusion is given in section 7.

2. Software/configware partitioning

The most common approach to solving combinatorial problems is based on construction of a search tree [12]. The root of the tree represents an initial point, i.e. an initial situation that has to be considered and examined. All other nodes of the tree correspond to situations that can be reached during the search process. If the desired solution exists then it can be found in one or more nodes of the tree. Thus, if we construct the complete tree we

will be able to find out a solution or to conclude that the problem instance does not have any solution.

The proposed technique performs software/configware partitioning according to the available logic capacity of the FPGA and incorporates the following strategy:

- The root of tree is the starting point for software of the host computer;
- If the problem satisfies the pre-given constraints (for the maximum allowed complexity of the given hardware) it will be completely solved in hardware. Thus, software of the host computer transfers the problem data to the attached hardware;
- If the problem does not satisfy the pre-given constraints, the software will apply special decomposition and reduction methods trying to obtain a sub-problem that satisfies the pre-given constraints. After that the hardware will be responsible for the subsequent steps (i.e. the sub-problem is transferred to the attached hardware);
- If hardware finds a solution the problem is considered to be solved and the result will be dispatched to the host computer;
- If the considered branch of the tree does not allow to find a solution the control will be returned to software;
- If the size of the current intermediate sub-problem exceeds the constraints, the execution will be continued in software;
- The considered steps will be repeated until we receive either a positive or a negative result, i.e. we will get the solution or we will conclude that the problem does not have any solution.

2.1. Example of the Boolean satisfiability problem

It is known that product of sums (POS) form is composed of a conjunction of a number of clauses, where a clause is a disjunction of one or more variables or their negations. The satisfiability problem consists in determining if a formula in POS is satisfiable, i.e. whether there exists an assignment of values to variables that forces the formula to evaluate to 1.

Let us formulate a SAT problem over a ternary matrix **M** by setting a correspondence between clauses and rows of **M**, and between variables and columns of **M** [8]. Each element m_{ij} of the matrix **M** is equal to:

- 1 – if clause c_i contains variable x_j;
- 0 – if clause c_i contains negated variable x_j;
- - (don't care) – if clause c_i does not contain variable x_j;

Note that the problem of satisfying a Boolean formula is equivalent to finding a ternary vector **v**, which is orthogonal to each row of the corresponding matrix **M** [8]. Orthogonality of two ternary vectors is defined as follows [12]: $(\mathbf{m}_i \; ort \; \mathbf{m}_j) \Rightarrow \{\mathbf{m}_i\} \cap \{\mathbf{m}_j\} = \varnothing$, where $\{\mathbf{m}_i\}$ ($\{\mathbf{m}_j\}$) is a set of Boolean vectors that correspond to the ternary vector $\mathbf{m}_i$ ($\mathbf{m}_j$) by replacing the don't care values ('-') with all possible combinations of 0s and 1s. If and only if two ternary vectors are not orthogonal (let us designate this as $\mathbf{m}_i$ ~~ort~~ $\mathbf{m}_j$) they do intersect in the Boolean space: $(\mathbf{m}_i$ ~~ort~~ $\mathbf{m}_j) \Leftrightarrow (\mathbf{m}_i \; ins \; \mathbf{m}_j)$.

If vector **v** cannot be found then the formula is unsatisfiable. On the other hand, if the vector **v** exists then the zeros and ones in it correspond to those variables that must receive values one and zero respectively in order to satisfy the formula.

For example, the following formula in POS:

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

can be presented by a ternary matrix **M**:

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

For this example, one possible solution is **v** = 10-, which means that x_1=0 and x_2=1. It is easy to check that such assignment of values to variables satisfies the formula (1).

In order to solve the SAT problem formulated over a ternary matrix we applied the algorithm proposed in [8]. The algorithm consists of a sequential application of various reduction and decomposition methods. The reduction methods allow the matrix **M** to be simplified by deleting rows and columns that cannot influence the final solution, and values 1 or 0 to be assigned to some components of vector **v** that initially is completely undetermined, i.e. **v**=[-…-]. The following reduction methods have been applied:

- All the columns that are completely undetermined (i.e. do not contain value 0 nor 1) are deleted from the matrix **M**.
- All the rows that are orthogonal to the vector **v** are deleted from the matrix **M**.
- All the columns that correspond to determined components of vector **v** are deleted from the matrix **M**.
- If there exists a row in the matrix **M** that has only one component with the value 0 or 1 and all the other components are equal to -, then the corresponding element of vector **v** is set to the inverse value.
- If there exists a column in the matrix **M** that does not contain the value 0 (or 1) then this value is assigned to the corresponding component of vector **v**.

When further reduction becomes impossible decomposition is applied. The adopted method selects an undetermined component of vector **v** and tries to assign a value to it. For this purpose a component is chosen that corresponds to the more determined column of the matrix **M**, i.e. to a column that has a minimal number of values -.

The selected vector component represents a decision variable, for which first the value 1 is tried and after that the value 0.

If after deleting a row the matrix becomes empty then the current value of the vector **v** represents the solution. 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 of values to variables will not lead to the solution. In this case the algorithm backtracks to the most-recently assigned component of vector **v** 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.

Figure 2 demonstrates software/configware partitioning for the SAT problem formulated over a ternary matrix. Figure 3 illustrates processing the search tree in software and in configware.

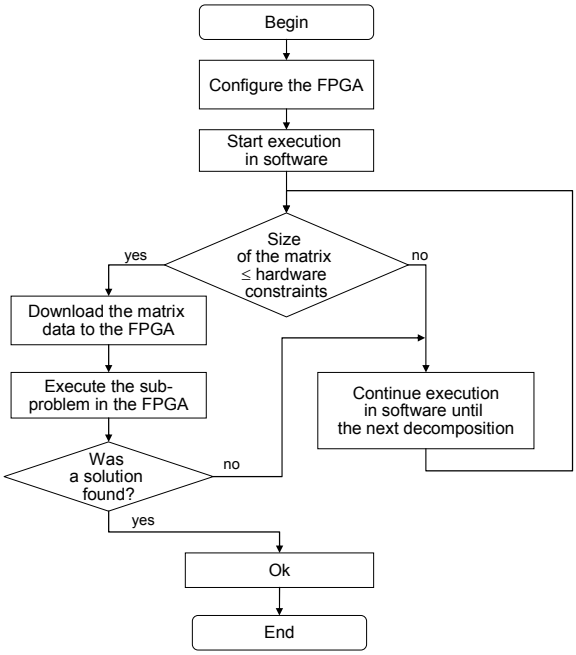

Figure 2. Software/configware partitioning

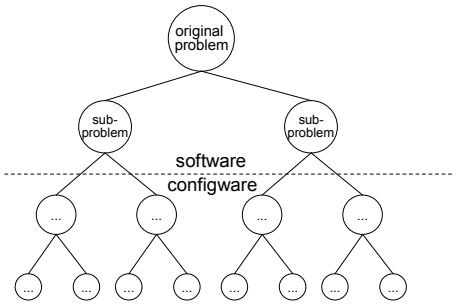

Figure 3. Processing the search tree in software and in configware

2.2. Example of the covering problem

This subsection shows how to construct a search tree for the exact method [12] that permits a minimal column cover of a binary matrix to be found, i.e. a minimal subset of matrix's columns containing at least one value 1 in each row. The method will be demonstrated through an example of the following matrix (columns D, F, G represent the minimal column cover):

	A	B	C	*D*	E	*F*	*G*	H	I
1	1	0	0	*0*	0	*1*	*0*	1	1
2	0	1	1	*0*	0	*0*	*1*	0	0
3	0	0	1	*1*	0	*1*	*0*	0	1
4	1	0	0	*1*	1	*0*	*0*	0	0
5	0	0	1	*0*	0	*0*	*1*	1	1
6	0	1	0	*0*	0	*1*	*0*	0	0
7	0	1	0	*0*	1	*0*	*1*	0	1
8	1	0	1	*1*	0	*0*	*0*	1	0
9	0	0	0	*0*	0	*1*	*0*	1	0
10	0	0	0	*1*	0	*0*	*0*	0	1

(2)

The following set of rules [12], permitting to simplify the matrix, will be used:

- If for $i \neq j$ row_i & row_j = row_j then row_i can be removed from the matrix, for example, row_1 = 100001011, row_9 = 000001010, row_1 & row_9 = row_9 and row_1 have to be removed from the matrix;
- If for $i \neq j$ $column_i$ & $column_j$ = $column_i$ then $column_i$ can be removed from the matrix, for example, after deleting rows 1 and 3 using the first rule $column_A$ = 01000100, $column_D$ = 01000101, $column_A$ & $column_D$ = $column_A$ and $column_A$ has to be removed from the matrix.
- If any column contains just values 0 it has to be removed from the matrix;
- If there is a row, which does not have values 1 then covering cannot be found.

The first two operations are called subsumption operations. For decomposition purposes the following rules will be used:

- If a row has just one value 1 then the respective column (in which this value 1 appears) must be included into the covering;
- If all rows have more than one value 1 then the first row from the top of the matrix that contains the minimum number of ones has to be selected. For this row it is necessary to analyze all possible sub-problems and the number of such sub-problems is equal to the number of values 1 in the row.

Obviously, any branch has to be examined until the step where an intermediate result becomes worse than any previously discovered covering. Figure 4 shows all the steps that are required in order to find out a minimal column cover of the matrix (2). The way that leads to the minimal cover (columns D, G, F) is shown with the aid of

double arrows. There are three branching points in Figure 4: D-E, G-H and D-I. After getting the first solution (D, G, F) we are interested just in coverings that contain 2 or less columns. Thus, it is not necessary to traverse all branches and the search process can be stopped at any point that gives a 2-component incomplete solution.

Like the considered above SAT problem, the search algorithm in Figure 4 can be partitioned between software and configware. Indeed, any intermediate matrix, constructed in the branching points, is smaller than the initial matrix. As soon as all the established hardware constraints for an intermediate matrix are satisfied the problem can be solved in hardware.

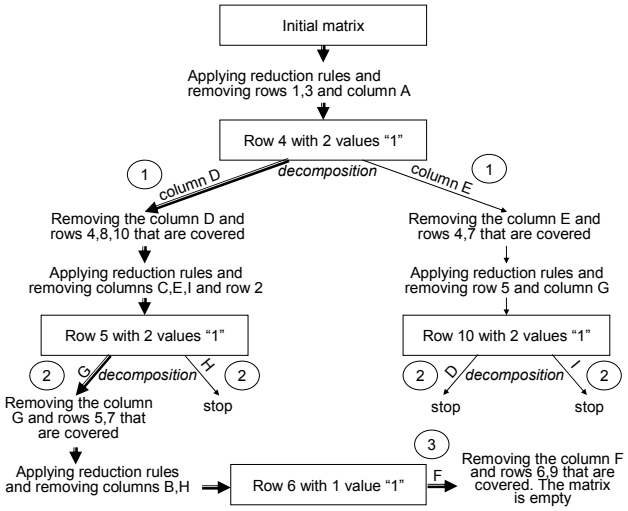

Figure 4. Search algorithm to find out a minimal column cover of the matrix (2)

Similar search algorithms can be used for solving many other combinatorial problems. For example, in [13] it is shown how a graph coloring problem can be formulated over a discrete matrix and an algorithm is proposed allowing the problem to be solved by backtracking search. Therefore the suggested software/configware partitioning method shown in Figure 2 can be applied directly.

3. Configware architecture

The basic components of the proposed configware architecture are shown in Figure 5. The dynamically reconfigurable units (DRU) run the algorithms and execute operations over matrix rows/columns (such as that described in section 2). Stack memory provides support for the backtracking process; in particular they permit to construct sub-matrices (for each sub-problem) sequentially and to return back to any intermediate sub-matrix if required. General-purpose registers store the intermediate data.

There are two blocks of memory storing matrices in Figure 5. The first one corresponds directly to the matrix **M** received from the host computer and the second one represents a transpose of **M**. As a result, we can access any row and column of **M** in 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 (*Row mask* and *Column mask* in Figure 5). Thus, we avoid the need to store the intermediate matrices in the stack.

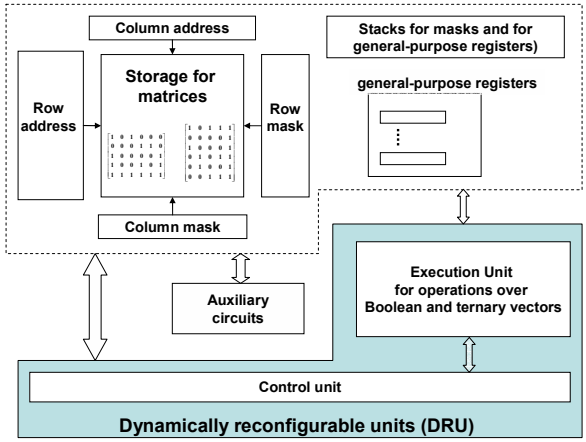

Figure 5. The proposed architecture

4. Dynamically reconfigurable units

DRU is the basic component of Figure 5 and altering the functionality of the DRU makes possible to configure (i.e. to adapt) the proposed architecture to the concrete problem. The proposed architecture for the DRU is shown in Figure 6 [14]. There are 8 registers in Figure 6, which are the following:

- operand registers R_1 and R_2 that store given vectors (row/column of the matrix) V_1 and V_2 and/or intermediate result(s) if required;
- temporary register R_t, which is useful for swapping vectors V_1, V_2 and similar operations;
- mask register R_m for masking some positions of the vectors V_1, V_2 and V_t;
- an index register "index" for selecting a desired element of the vectors V_1 and V_2;
- size register "size" for storing the size of the vectors;
- counter "count" for performing counting operations;
- temporary register "temp" for keeping intermediate results of counting operations.

Any ternary vector is composed of two Boolean vectors BV0 and BV1. The vector BV0 (BV1) contains ones in the positions that have zeros (ones) in the respective ternary vector. The component "=" (see Figure 6) permits values in the registers "index" and "size" to be compared.

Run time changes to the functionality can be achieved with the aid of 4 dynamically reconfigurable components, which are:

- RVCC - reconfigurable combinational circuit that performs operations logical over entire vectors V_1 or/and V_2;
- RECC - reconfigurable combinational circuit that performs operations over individual elements of V_1 and V_2 with the same index I;
- RFSM - reconfigurable finite state machine that permits to carry out sequential operations over V_1 or/and V_2;
- RC - reconfigurable comparator that permits the selection of different criteria for comparison of the registers "temp" and "count".

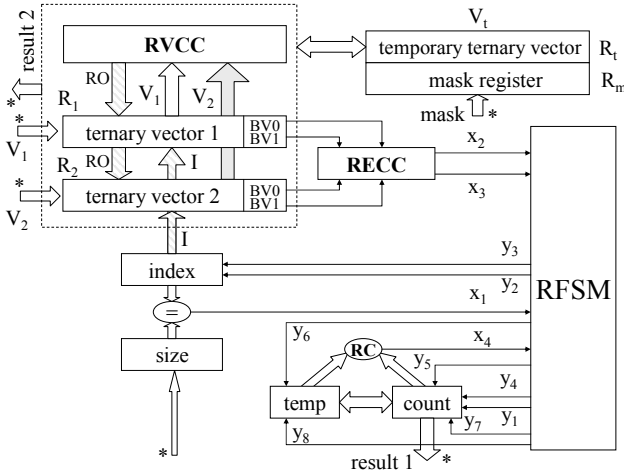

Figure 6. The basic architecture of the DRU

The variables $x_1,x_2,...,x_4$ and also others that are not shown in Figure 6 inform RFSM about the states of the DRU, such as "index"="size", "index"≤"size", etc. The variables $y_1,y_2,...,y_8$ and some others that are not shown in Figure 6 enable us to change the states of the DRU. Inputs and outputs of the DRU that are marked with asterisks are external and they permit loading the vectors, setting the mask register R_m and retrieving the result of the operation.

Reconfiguration makes it possible to alter the functionality of the blocks RVCC, RECC, RC and RFSM and it can be done from the host computer using the technique [14, 15].

5. Modeling the DRU functionality in software

We have already mentioned that architecture depicted in Figures 5 and 6 can be reused for solving different combinatorial problems and the functionality can be customized through changes in the reprogrammable blocks. Prior to the implementation of a particular circuit in hardware, it is important to model and to test it in software. This technique is also very helpful for debugging purposes and for examining and comparing various alternative algorithms.

A software model of a DRU is organized as a set of communicating objects, which are instances of classes described in C++. Basic relationships between the classes are expressed through aggregates and dependencies (see the simplified class diagram in Figure 7). The classes enclosed in single-line rectangles (shown at the bottom of Figure 7) describe the functionality of a cascaded RFSM model and all these classes were considered in [16] with detailed examples. The classes enclosed in double-line rectangles (shown at the top of Figure 7) express the functionality of the DRU without the RFSM.

The *accelerator* class models the architecture shown in Figure 6 and contains data (objects) of both user-defined and predefined C++ types. The objects of the latter type are *size*, *count*, *index* and *temp* and their names are the same as the names of the respective components in Figure 6. There are four objects of user-defined types in the *accelerator* class, namely *TV1*, *TV2*, *TVt* (which are instances of the class *Ternary_vector*) and *mask_register* (which is an instance of the class *Boolean_vector*). These objects model the components R_1, R_2, R_t and R_m accordingly (see Figure 6). The class *Ternary_vector* contains two objects *BV1* and *BV0* of the type *Boolean_vector* that use the encoding method considered in section 4. An object of type *FSM_template* is invoked in the class *accelerator* in order to provide the required sequential functionality. Thus, the behavior of the class *accelerator* depends on the behavior of the RFSM, which is indicated by the dependency relationship (see a dashed line shown in Figure 7). Rhomboidal symbols indicate aggregate relationships.

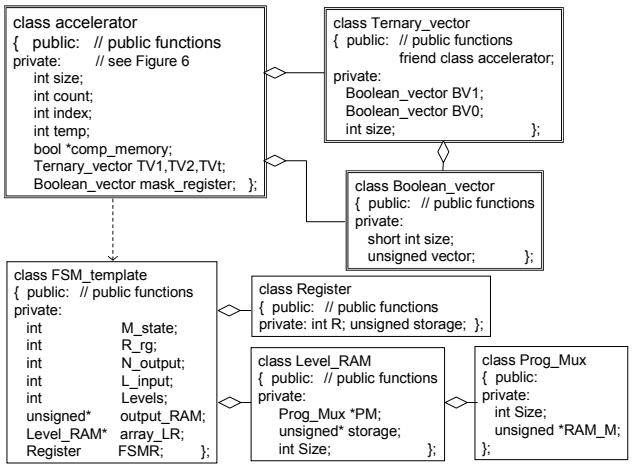

Figure 7. Modeling the architecture in Figure 6 through a set of C++ classes

An RFSM object is created with the aid of the following class constructor:

```
FSM_template::FSM_template (int L, int R, int M, int N, int levels, unsigned RAMI[][16], unsigned *tableI, unsigned *tableOUTI);
```

where *L* is the number of RFSM inputs, *R* is the size (number of bits) of the RFSM register, *M* is the number of RFSM states, *N* is the number of RFSM outputs, *levels* is the number of levels in the cascaded combinational circuit of the RFSM [16]. The last three parameters are arrays that model reloadable RAM blocks permitting RFSM behavior to be altered. The two-dimensional array *RAMI* contains data for the RAM blocks that are required for each level (the first dimension corresponds to the number of levels and the second dimension to the depth of the RAM block for the respective level). The parameter *tableI* is an array that is supplied to the constructor and contains the data needed for the RAM block of a programmable multiplexer (see [16] for details). The parameter *tableOUTI* is an array that consists of data for generating the RFSM outputs [16]. The particular arrays can be modeled in software and then used in the host computer for reconfiguration of the DRU.

There is a special function *Reload* in the *FSM_template* class, which has the following prototype:

```
void FSM_template::Reload (unsigned RAM[][16], unsigned *table, unsigned *tableOUT);
```

This function allows the primary arrays mentioned above to be reloaded thus enabling the RFSM functionality to be changed.

The behavior of the blocks *RC*, *RECC* and *RVCC* (see Figure 6) is modeled through the *accelerator* class functions and overloaded operators.

Sequential operations are executed by calling special functions that establish links between the selected vector in the accelerator and the RFSM. These functions have the following general prototype:

```
unsigned Solve_TVe_BVf (FSM_template &FSM_t);
```

where *FSM_t* is a reference to an object (RFSM) that controls the desired sequence of steps. A C++ program, which models the RFSM functionality, allows RAM blocks to be reloaded or switched enabling this functionality to be changed. This program was described in detail in [16].

Let us now consider how the designed C++ classes can be used for validating, debugging and verification of various operations that are allowed for the proposed DRU. Figure 8 with a fragment of C++ *main* function demonstrates how to specify the functionality of the RFSM and how to change it during execution time. The object *A* of type *accelerator* models the architecture depicted in Figure 6. Two external objects *TV1* and *TV2* are copied to the accelerator *A* using the functions *write_TV1* and *write_TV2*. The *FSM_template* class constructor creates an object *FSMT*, which will specify the first type of functionality.

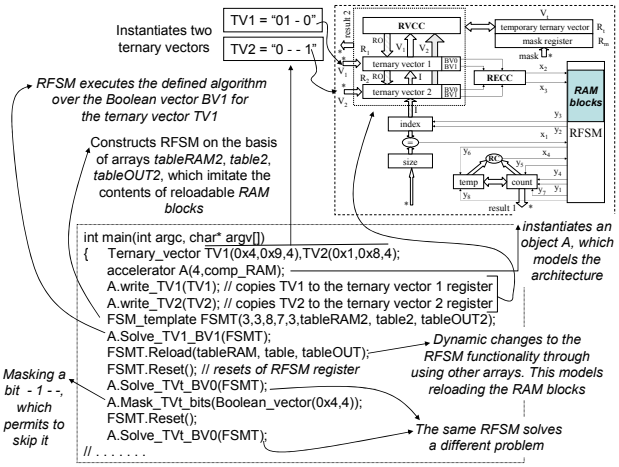

Figure 8. An example of modeling

The function *Solve* permits the defined RFSM behavior to be applied to a vector, which is indicated by the function name. Thus, calling *Solve_TVt_BV0* (FSMT) establishes an interaction between the RFSM and the Boolean vector *BV0* of the temporary ternary vector (*TVt*). The function *Reload* makes changes to the RFSM functionality and then the new functionality can be associated with any vector (see the first and the last functions *Solve* in Figure 8).

6. Experimental results

In order to estimate the effectiveness of the proposed approach, a number of experiments have been conducted for Boolean satisfiability, matrix covering and graph coloring problems. The problems were formulated over discrete matrices and were solved using backtracking algorithms. For this, all the individual reprogrammable components of the circuit in Figure 5 have been modeled in software and after that implemented in a PCI FPGA-based ADM-XRC board [17] with the Xilinx XCV812E FPGA. Various modes of static and dynamic reconfiguration have been examined. In particular, two kinds of implementation have been evaluated:

- components constructed through programming look-up tables - LUTs, i.e. on the basis of CLBs configured as ROM/RAM.
- components implemented on the basis of embedded memory blocks.

The results of experiments have shown that the proposed technique makes possible to shorten essentially the design time of combinatorial processors. The hardware compilation time was completely eliminated and significant speedups (up to 100) were achieved for solution of a number of problem instances (for example, available from DIMACS [18]) compared to the equivalent solution in software [8, 13].

7. Conclusion

The paper presents an architecture for combinatorial problem solvers that is based on partitioning the problem solution between software and dynamically reconfigurable hardware. The technique allows instance-specific hardware compilation to be avoided and permits problem instances to be solved that are larger than the available capacity of the reconfigurable hardware platform. As a result, it is possible to achieve a significant speedup (even taking into account the FPGA configuration time) compared to the equivalent solutions in software.

Modeling the architecture in software allows to estimate the time of reconfiguration, performance and other important characteristics for solving a particular combinatorial problem and ultimately leads to a significant speedup in the design process.

8. References

[1] G. Micheli, Synthesis and optimization of digital circuits. McGraw-Hill, Inc., 1994, 570 p.

[2] P. R. Stephan et al., "Combinational Test Generation Using Satisfiability," IEEE Trans. on CAD, vol. 15, no. 9, pp. 1167-1176, Sept. 1996.

[3] R. Feldman, C. Haubelt, B. Monien, J. Teich, “Fault Tolerance Analysis of Distributed Reconfigurable Systems Using SAT-Based Techniques”, Proceeding of FPL’2003, Lisbon, Portugal, 2003, pp. 478-487.

[4] M. Platzner, “Reconfigurable Accelerators for Combinatorial Problems”, IEEE Computer, pp. 58-60, April 2000.

[5] P. Zhong, P. Ashar, S. Malik, M. Martonosi, “Using Reconfigurable Computing Techniques to Accelerate Problems in the CAD Domain: A Case Study with Boolean Satisfiability”, Proc. of the 34th Design Automation Conference, 1998, pp. 194-199.

[6] M. Abramovici, J.T. de Sousa, “A SAT solver using reconfigurable hardware and virtual logic”, Journal of Automated Reasoning, vol. 24, nos. 1-2, Feb. 2000, pp. 5-36.

[7] M. Boyd, T. Larrabee, “ELVIS – A Scalable, Loadable Custom Programmable Logic Device for Solving Boolean Satisfiability Problems”, Proc. of the IEEE FCCM’2000, Napa, California.

[8] I. Skliarova, A.B. Ferrari. “A Software/Reconfigurable Hardware SAT Solver”, IEEE Trans. on VLSI, Apr., vol. 12, Nº 4, 2004, pp. 408-419.

[9] I. Skliarova, A.B. Ferrari, "Reconfigurable Hardware SAT Solvers: a Survey of Systems", IEEE Trans on Computers, Nov., vol. 53, Nº 11, 2004, pp. 1449-1461.

[10] J. de Sousa, J.P. Marques-Silva, M. Abramovici, “A Configware/Software Approach to SAT Solving”, Proc. of the IEEE FCCM’2001.

[11] A. Zakrevskij, “Combinatorial Problems over Logical Matrices in Logic Design and Artificial Intelligence”, Electrónica e Telecomunicações, vol. 2, no. 2, pp. 261-268.

[12] A.D. Zakrevski, “Logical Synthesis of Cascade Networks”, Moscow: Science, 1981.

[13] V. Sklyarov, I. Skliarova, B. Pimentel, "Modeling and FPGA-based implementation of graph coloring algorithms", Proceedings of the 3rd International Conference on Autonomous Robots and Agents - ICARA'2006, Palmerston North, New Zealand, December 2006, pp. 443-448.

[14] V. Sklyarov, I. Skliarova, “Design of Digital Circuits on the Basis of Hardware Templates”, Proceedings of the International Conference on Embedded Systems and Applications – ESA’03, Las Vegas, USA, CSREA Press, pp. 56-62, Jun. 2003.

[15] V. Sklyarov, I. Skliarova, A. Oliveira, A. Ferrari, “A Dynamically Reconfigurable Accelerator for Operations over Boolean and Ternary Vectors”, Euromicro Symposium on Digital System Design, Belek, Turkey, pp. 222-229, Sept. 2003.

[16] V. Sklyarov, “Reconfigurable models of finite state machines and their implementation in FPGAs”, Journal of Systems Architecture, 2002, 47, pp. 1043-1064.

[17] Available: http://www.alpha-data.com/.

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