Solving Sudoku in Reconfigurable Hardware

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

***Abstract*—In this paper we explore the effectiveness of solution of computationally intensive problems in FPGA (Field-Programmable Gate Array) on an example of Sudoku game. Three different Sudoku solvers have been fully implemented and tested on a low-cost FPGA of Xilinx Spartan-3E family. The first solver is only able to deal with simple puzzles with reasoning, i.e. without search. The second solver applies breadth-first search algorithm and therefore has virtually no limitation on the type of puzzles which are solvable. We prove that despite the serial nature of implemented backtracking search algorithms, parallelism can be used efficiently. Thus, the suggested third solver explores the possibility of parallel processing of search tree branches and boosts the performance of the second solver. The trade-offs of the designed solvers are analyzed, the results are compared to software and to other known implementations, and conclusions are drawn on how to improve the suggested architectures.**

Keywords-Sudoku; FPGA; breadth-first search; parallel processing

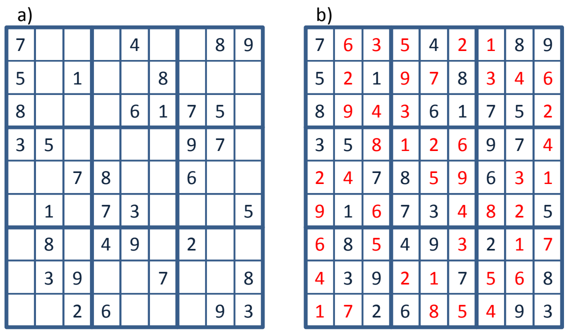
I. INTRODUCTION

Sudoku is a well known combinatorial number-placement puzzle which achieved international widespread popularity in 2005 [1]. The most common version of the game consists of 81 cells organized in a 9×9 grid (which in turn is divided in nine 3×3 regions) where some cells contain numbers from 1 to 9. The objective is to fill in the remaining empty cells with numbers 1..9 so that:

1. every number (1..9) appears in each row only once;
2. every number (1..9) appears in each column only once;
3. every number (1..9) appears in each 3×3 region only once.

The most common version of Sudoku (9×9 grid with 3×3 regions), can be generalized to other sizes. A Sudoku of order N is an $N^2 \times N^2$ grid divided in N^2 regions (each of size $N \times N$) which has to be filled with numbers from 1 to N^2. The generalized Sudoku problem is NP-complete [2].

An example of a Sudoku puzzle of order 3 is depicted in Fig. 1,a, where 3×3 regions are shown with thick lines. A solution to this puzzle is presented in Fig. 1,b.



a)

7				4			8	9
5		1			8			
8				6	1	7	5	
3	5					9	7	
		7	8			6		
	1		7	3				5
	8		4	9		2		
	3	9			7			8
		2	6				9	3

b)

7	6	3	5	4	2	1	8	9
5	2	1	9	7	8	3	4	6
8	9	4	3	6	1	7	5	2
3	5	8	1	2	6	9	7	4
2	4	7	8	5	9	6	3	1
9	1	6	7	3	4	8	2	5
6	8	5	4	9	3	2	1	7
4	3	9	2	1	7	5	6	8
1	7	2	6	8	5	4	9	3

Figure 1. a) An example of Sudoku puzzle of order 3; b) a solution to the puzzle.

Very recently (and especially after the Field-Programmable Technology conference launched design competition) an interest appeared to explore the possibility and effectiveness of solving Sudoku with FPGA (Field-Programmable Gate Array). In this paper, we also tackle the problem but only considering puzzles of order 3. In particular, we exploit parallelism to increase performance.

The remaining part of this paper is organized in five sections. An overview of the related work is done in section II. The implemented algorithms are described in section III. The proposed architectures are presented in section IV. Finally, section V discusses implementation details and the results of experiments. Conclusions are given in section VI.

II. RELATED WORK

In the recent years we witnessed a tremendous progress in the scope of reconfigurable systems. FPGAs appeared on the market in 1985 and now they have evolved to devices containing more than 6 billion transistors [3] and Extensible Processing Platforms incorporating multi-core processors [4] and targeting embedded designers working on market applications that require multi-functionality and real-time responsiveness. FPGAs, because of their processing throughput and inherent reconfigurability, are widely used and have proven to be effective for algorithm acceleration in high-performance computing in the areas of biological sequencing, searching, sorting, coding/decoding, signal processing, audio/video/image manipulation, image and speech recognition, encryption, and many others where significant speed and power savings over traditional microprocessors, graphics processing units, and digital signal processors were achieved [5-7].

A number of research works explore the FPGA-based implementation of computationally intensive problems such as the Eigenvalues calculation [8], fast Fourier transform [9], data sort [10], Boolean Satisfiability problem (SAT) [11], Eternity II [12], and Sudoku [13-16].

In [13] a probabilistic Sudoku solver is presented based on simulated annealing methods. The solver was implemented on a Xilinx Virtex-II FPGA, ran at 53 MHz and is capable of handling puzzles up to 12th order. The obtained results were compared to the implementation of exactly the same algorithm in software and the authors reports that both systems behaved similarly in terms of performance (FPGA was faster for lower order puzzles, software performed better for higher order puzzles). The suggested technique is not applicable to hard instances (none of the tested "hard" puzzles was solved in [13]).

Hardware implementation of an exhaustive search algorithm for Sudoku (which does not use any search space pruning) is presented in [14]. The solver was implemented in a Xilinx Virtex-II FPGA, supported 50 MHz clock frequency and was able to tackle problems up to order 8. Once again the technique is not applicable to hard instances because the search is not directed.

Another FPGA-based Sudoku solver is proposed in [15]. In this case, the backtracking search algorithm is augmented with several heuristics that lead to search space reduction. The solver can process puzzles up to order 11.

In [16] depth-first search algorithm is implemented and a new memory scheme is proposed which allowed to reduce memory requirements of [13-15] and to solve puzzles up to order 14 in a Spartan-3E FPGA. Parallelization was introduced as well. The results were compared to a state-of-the-art SAT solver (by modeling Sudoku as the SAT problem as in [17]) and the authors found that their FPGA-based implementation can only speed up easy puzzles while the hard ones are processed slower than in software.

In this work we further explore the problem with particular emphasis on implementing: a) a backtracking search algorithm with search space pruning; b) a number of parallel Sudoku solvers (all the overviewed research works pointed that state-of-the-art software solving methods are more competitive and therefore parallelism must be the main target of hardware implementations).

III. Implemented Algorithms

For every empty cell in a puzzle a list of possibilities must be generated. This list includes all the numbers that theoretically could be placed in each cell (taking into account the current puzzle constraints).

The following search space pruning techniques have been implemented:

- *Singles*: when a cell has only one candidate number to be used, this number is assigned to that cell and removed from the lists of candidates in all other cells in the same row, column, and region;
- *Hidden Singles*: if a candidate number appears only once in a given row, column or region, it is assigned to the respective cell.

When further pruning is not possible, we recur to a backtracking algorithm which:

1. Finds a cell with the smallest number of candidates;
2. Tries those candidates one by one (performing problem simplification as described above);
3. If a cell without candidates appears it means that this trial was wrong and the algorithm backtracks to explore the remaining possibilities; otherwise go to point 1 above;
4. The algorithm stops when all the cells become filled with a valid number.

Basically, we implemented two algorithms: A_{simple}, which just performs pruning of the search space and therefore can only solve very simple puzzles and A_{search}, which applies breadth-first search strategy and thus has virtually no limitation on the type of puzzles which are solvable. The flowcharts of both algorithms are presented in Fig. 2.

IV. Proposed Architectures

Three different Sudoku solver architectures have been implemented. The first solver S_{simple} implements the algorithm A_{simple} and is therefore only able to solve simple puzzles with reasoning, i.e. without search. The second solver S_{search} applies breadth-first search algorithm A_{search} and can process any puzzle (provided memory is available). And, finally, the third solver $S_{parallel}$ implements the algorithm A_{search} and explores the possibility of parallel processing of nodes in the search tree. In particular, all the sub-puzzles which are instantiated at the same level in the search tree are solved in parallel. Moreover, if further branching occurs, the new lower level sub-puzzles are also executed in parallel with those already running.

A. Organization of Data in Memory

Table I gives information about data stored in memory in our approach.

TABLE I. Memory requirements

Name	Description
puzzle	puzzle itself
auxiliary puzzle(s)	copies of the puzzle which are required by the solvers S_{search} and $S_{parallel}$
candidates	lists of candidates for every cell
map of columns	column index for every cell
map of regions	region index for every cell
map of minimals	ordered (by the number of candidates) list of cells
map of indices	indicates which memories are free (to be used by the solvers S_{search} and $S_{parallel}$)

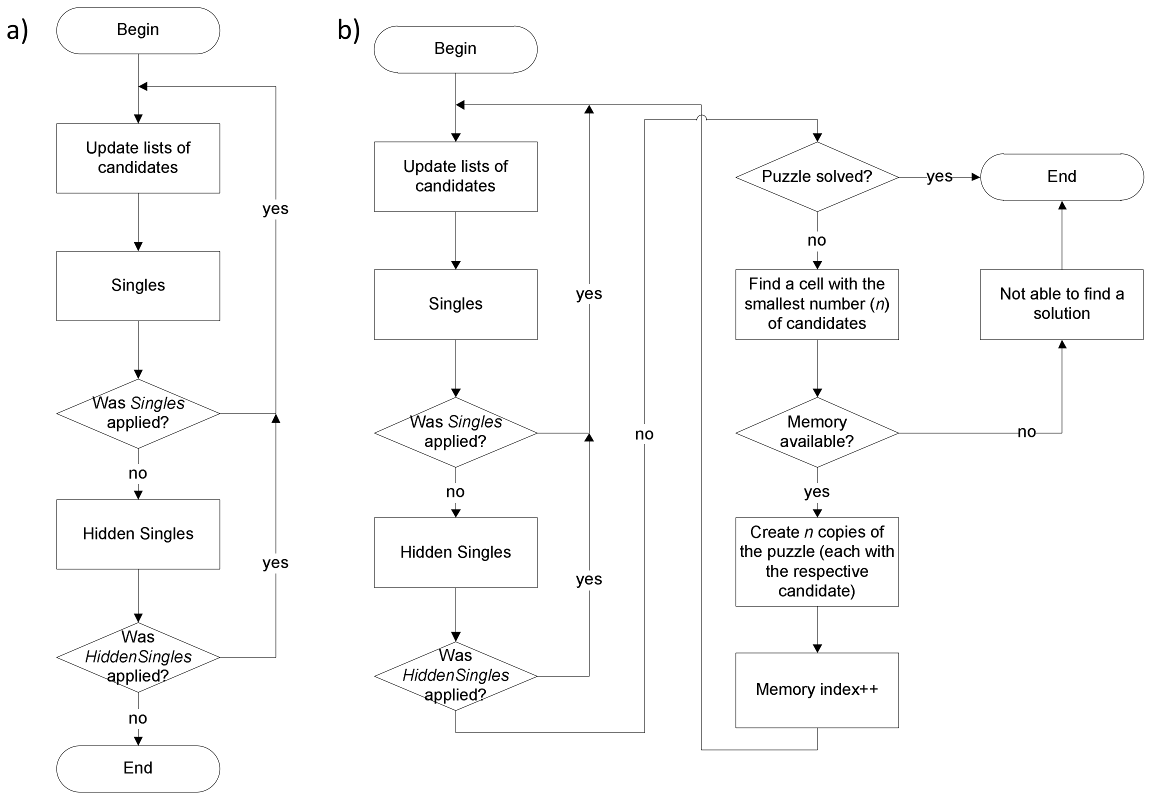

a)
Begin
Update lists of candidates
Singles
Was Singles applied?
no
Hidden Singles
Was HiddenSingles applied?
no
End
yes
yes
b)
Begin
Update lists of candidates
Singles
Was Singles applied?
no
Hidden Singles
Was HiddenSingles applied?
yes
yes
no
Puzzle solved?
yes
End
no
Find a cell with the smallest number (n) of candidates
Memory available?
no
Not able to find a solution
yes
Create n copies of the puzzle (each with the respective candidate)
Memory index++

Figure 2. a) Flowchart of the algorithm A_{simple} which is only able to solve simple puzzles by pruning the search space; b) Flowchart of the algorithm A_{search} which applies breadth-first search strategy.

The original puzzle as well as auxiliary puzzles are stored in memory as 81 (9×9) 4-bit words (see Fig. 3,a), where every cell value is represented by the respective binary code (code “0000” is reserved for empty cells).

Each cell might have up to 9 candidate numbers (in the worst case). Therefore we store the list of candidates as 81 36(9×4)-bit words, where every word is organized as illustrated in Fig. 4. For example, the cell A1 from Fig. 3 already has a value. Therefore this cell does not have any candidate and all 36 bits are assigned to zero. The cell A2 has two possible candidate numbers: 2 and 6. So, these numbers are coded in binary and occupy 4 bits each as illustrated in Fig. 4; the remaining bits are assigned to zero. The cell A4 has three possible candidate numbers: 2, 3, and 5, which are coded in binary with 4 bits each as shown in Fig. 4.

The maps of columns and regions are used to find easily which column or region does a particular cell belong to. These memories are organized as 81 (9×9) 4-bit words (see Fig. 3,b,c). The maps of columns and regions are implemented as ROMs whose contents is defined statically and does not change during the solution of a problem (as well as when we switch between different problem instances). These maps are not required in software but we found that they facilitate memory addressing in hardware.

The map of minimals keeps ordered (by the number of candidates) addresses of cells and is used in branching (in order to split the search on the cell that has the smallest number of candidates). This memory is organized as 81 7-bit words.

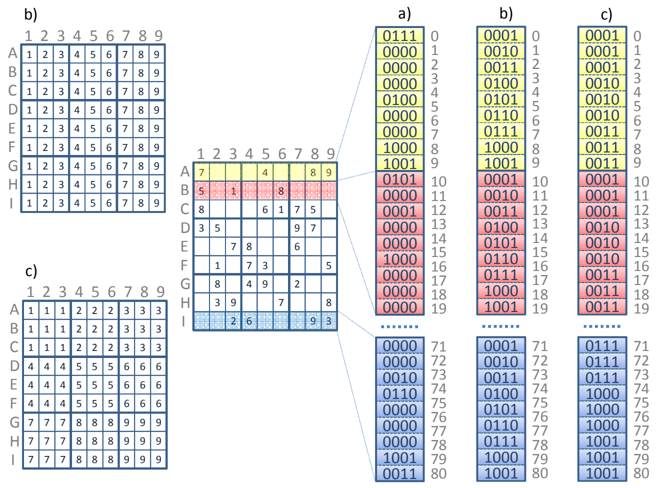


Figure 3. Representation of data in memory: a) storing the original and auxiliary puzzles; b) map of columns; c) map of regions.

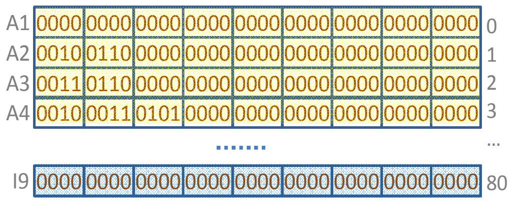

A1 0000 0000 0000 0000 0000 0000 0000 0000 0000 0
A2 0010 0110 0000 0000 0000 0000 0000 0000 0000 1
A3 0011 0110 0000 0000 0000 0000 0000 0000 0000 2
A4 0010 0011 0101 0000 0000 0000 0000 0000 0000 3
...
I9 0000 0000 0000 0000 0000 0000 0000 0000 0000 80

Figure 4. Representation of the lists of candidates in memory.

Finally, the map of indices points out which memories are free and can therefore be used for storing auxiliary puzzles in the solvers S_{search} and $S_{parallel}$. This memory is composed of 32 1-bit flags and can be used for managing at most 32 auxiliary puzzles (but is easily scalable to deal with more puzzles). At the beginning of execution, the map is filled in with zeros.

B. *Solvers Architectures*

The block diagrams of the solvers S_{simple} and S_{search} are illustrated in Fig. 5 and Fig. 6. The first solver S_{simple} (see Fig. 5) is able to process very simple puzzles that can be solved without branching by applying two reduction techniques: *Singles* and *Hidden Singles*. The *Control Unit* manages access to memory blocks (which hold the original puzzle, the list of candidate numbers, the map of columns, and the map of regions) with the aid of a priority encoder and multiplexers for addresses, data and write enable signals. The access is arbitrated between external interface (for example, USB interface to a host computer) and three VHDL processes, which are responsible for application of the two reduction techniques (*singles* and *hidden singles* processes) and updating the list of candidate numbers (*list of candidates* process) according to the algorithm A_{simple} (see Fig. 2,a). The processes are implemented as communicating finite state machines [18].

The second solver S_{search} (see Fig. 6) applies breadth-first search algorithm as soon as it concludes that further reduction is not possible. The process *find minimal* is responsible for finding an empty cell *C* which has the smallest number of candidates *n*. Then *n* copies of the puzzle are done and each of these has the cell *C* filled with one of the candidates. After that the copied auxiliary puzzles are processed by the solver one by one sequentially according to the algorithm A_{search} (see Fig. 2,b) eventually creating more copies of the sub-puzzles which are added to the end of the queue and processed in the order of their instantiation. The process *check solution* is responsible for verifying if the current puzzle is completely filled in and a solution was found, a case in which the *Control Unit* has to stop running the remaining puzzles.

With this architecture any Sudoku puzzle can be solved provided there is memory available for keeping auxiliary copies as described above. Access to memories (which store the list of the candidates and different maps) is arbitrated by the *Control Unit* in a way similar to the solver S_{simple}. Access to *K* puzzles is controlled with the aid of a demultiplexer (not shown in Fig. 6 for the sake of simplicity) activated by *index* supplied by the *Control Unit*. As soon as a solution is found, data from the respective puzzle are sent to the external interface via a multiplexer (also not shown in Fig. 6) controlled by *index* from the *Control Unit*.

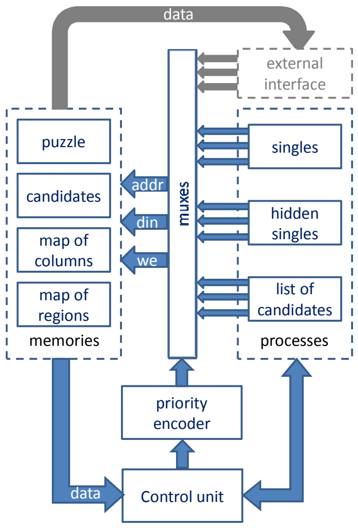


Figure 5. Block diagrams of the solver S_{simple}.

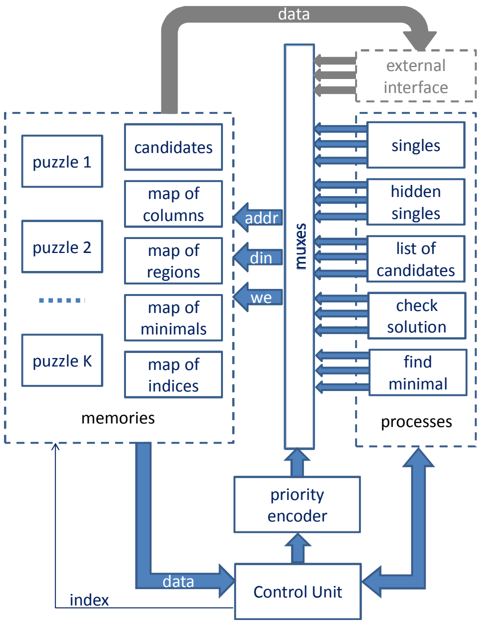


Figure 6. Block diagrams of the solver S_{search}.

Finally, the block diagram of the third solver $S_{parallel}$ is illustrated in Fig. 7 and Fig. 8. The solver is started by activating the first engine *Solver 1*. Each solving engine (see Fig. 7) is only responsible for applying *Singles* and *Hidden Singles* reduction techniques and for finding a cell *C* with the minimum number of candidate numbers *n*. Information about the cell *C* is then sent to the *Parallel Control Unit* (see Fig. 8) which makes *n* copies of the puzzle and activates *n* next available solving engines which all execute in parallel. As soon as one of these engines needs to branch the search space the process repeats by activating n_{new} more solving engines. As soon as one of them reports that a solution is found the *Parallel*

Control Unit aborts all the remaining solving engines and notifies the respective external entity and data from the relevant solving engine are sent to external interface via a multiplexer. As in the previous case, with this architecture any Sudoku puzzle can be solved provided resources are available for implementing the required number of solving engines. Access to different solving engines is controlled with the aid of demultiplexers activated by the respective *index* supplied by the *Parallel Control Unit*.

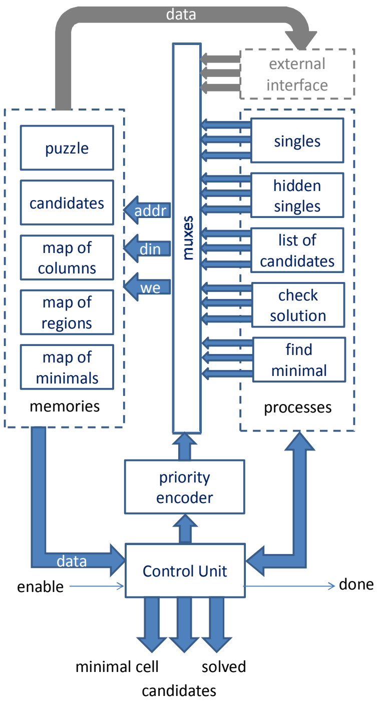


Figure 7. Block diagrams of the solver $S_{parallel}$: structure of the individual solving engine which does not perform branching.

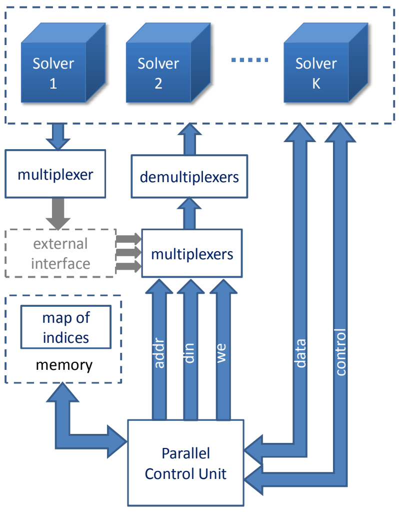


Figure 8. Block diagrams of the solver $S_{parallel}$: *K* solvers controlled by a single parallel control unit.

V. THE RESULTS OF EXPERIMENTS

All three proposed architectures were described in VHDL, synthesized with Xilinx ISE 12.1, and tested on a Nexys-2 prototyping board containing XC3S1200E FPGA. The FPGA has 28 18-Kbit embedded memory blocks (BlockRAMs). We configured each BlockRAM as a single-port memory and made so that every entity described in Section IV.A occupies one BlockRAM. This configuration permitted to implement easily the solver S_{simple}, to keep up to 21 auxiliary puzzles in the solver S_{search}, and to instantiate at most 5 parallel solving units in the solver $S_{parallel}$.

The occupied FPGA resources are summarized in Table II. Please note that although it is possible to keep up to 21 auxiliary puzzles in the solver S_{search}, only 12 were instantiated because these were sufficient for the experiments. That is why the this solver needs just 19 BlockRAMs.

Experiments were done with benchmarks available at [19-21]. The results are summarized in Tables III-IV. We have designed a software program (in C) which implements the algorithms A_{simple} and A_{search}. The respective software execution times t_{sw} (measured on a HP EliteBook 2730p with Inter Core Duo CPU 1.87 GHz) are presented in the rightmost columns of Tables III-IV. The column "*# copied puzzles*" gives information on how many search tree branches it was necessary to explore (and, as a consequence, how many puzzle copies were executed, or how many solving engines were activated – in case of the solver $S_{parallel}$). The solver $S_{parallel}$ supports at most 5 parallel solving units and therefore is not able to process the last 3 instances from the Table IV.

From the Tables you can see that the solvers S_{simple} and S_{search} are less efficient than the respective software implementations for all the selected benchmarks. This is an expected result because these two solvers implement very control-oriented algorithms and only explore low-level parallelism (at the level of operations). On the other hand, the solver $S_{parallel}$, which explores parallel traversal of the search tree branches, leads to execution times more closely related to software solution. The solver $S_{parallel}$ is on average 2.4 times faster than the solver S_{search}.

TABLE II. FPGA RESOURCES OCCUPIED BY THE IMPLEMENTED SUDOKU SOLVERS AND THE MAXIMUM ACHIEVABLE CLOCK FREQUENCY

	S_{simple}	S_{search}	$S_{parallel}$
slices	529 (6%)	961 (11%)	4297 (49%)
BlockRAMs	4 (14%)	19 (67%)*	27 (96%)
freq (MHz)	55.6	43.4	51.7

TABLE III. THE RESULTS OF EXPERIMENTS WITH THE SOLVER S_{SIMPLE}

puzzle	**S_{simple} (ms)**	**t_{sw} (ms)**
Easy Puzzle 1	1.43	0.93
Easy Puzzle 2	0.79	0.59
Easy Puzzle 3	1.20	0.77
Easy Puzzle 4	1.68	0.98
Medium Puzzle 1	3.92	2.04
Medium Puzzle 2	1.57	3.76
Medium Puzzle 3	2.03	1.73
Medium Puzzle 4	3.36	2.87

TABLE IV. THE RESULTS OF EXPERIMENTS WITH THE SOLVERS S_{SEARCH} AND $S_{PARALLEL}$

puzzle	S_{search} (ms)	$S_{parallel}$ (ms)	# copied puzzles	t_{sw} (ms)
Hard Puzzle 5	14.19	5.65	4	5.63
Hard Puzzle 9	20.64	5.43	4	6.94
Hard Puzzle 12	8.16	4.91	2	4.34
Expert Puzzle 1	6.69	4.15	3	3.99
Expert Puzzle 2	22.09	4.98	2	4.78
Extreme Puzzle 5	11.03	5.15	1	3.39
Evil Puzzle 1	22.18	9.57	1	5.53
Evil Puzzle 2	10.56	7.22	3	5.39
Extra Chall 13	18.89	7.26	1	4.92
Evil Puzzle 17	6.22	5.22	1	4.15
Evil Puzzle 5	9.26	5.64	2	5.22
Hard Puzzle 10	7.36	5.09	3	3.36
Expert Puzzle 10	15.24	3.97	1	2.75
Hard Puzzle 10a	11.04	na	8	4.86
Evil Puzzle 5a	24.62	na	7	11.52
Evil Puzzle 17a	18.19	na	7	10.33

We would like to underline that the experiments were done on a low-cost FPGA which supports very limited clock frequency (about 37 times slower that the PC used). If we migrate the same designs to a more advanced FPGA chip, obviously better results will be received. Besides, we used a number of auxiliary test circuits that check the validity of puzzles frequently. These circuits slow down the solvers and will be removed in future implementations.

We have also compared our results with those of [13-16] using a benchmark *3a* from the Field-Programmable Technology conference [22]. The solver S_{simple} was 5 times faster than [14], 2 times faster than [15] but slower than [16] (simulated annealing–based solver [13] was not able to process *3a* instance).

VI. CONCLUSIONS

In this paper three different architectures were presented that allow a Sudoku puzzle to be solved on an FPGA. We proved that despite the serial nature of the implemented algorithms, parallelism can be applied efficiently allowing performance to be increased in 2.4 times. Nevertheless, even the parallel FPGA-based solver is slower than software solution. This leads to the conclusion that parallelism should be explored more aggressively. There is room for such exploration even with the used low-cost FPGA through more efficient data structures and BlockRAM management.

ACKNOWLEDGMENT

This research was supported by HP grant "Use of HP Mobile Technology to Enhance Teaching Reconfigurable Systems", FEDER through the Operational Program Competitiveness Factors - COMPETE and by National Funds through FCT - Foundation for Science and Technology in the context of project FCOMP-01-0124-FEDER-022682 (FCT reference PEst-C/EEI/UI0127/2011).

REFERENCES

[1] J.P. Delahaye, "The Science behind Sudoku", Scientific American Magazine, pp. 80-87, June 2006.

[2] T. Yato, T. Seta, "Complexity and Completeness of Finding Another Solution and its Application to Puzzles", IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, vol. E86-A, no. 5, pp. 1052-1060, 2003.

[3] Xilinx products, World's Highest Capacity FPGA - Now Shipping (25 October, 2011), available at: http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/.

[4] M. Santarini, "Zynq-7000 EPP Sets Stage for New Era of Innovations", Xcell journal, issue 75, second quarter, available at: http://www.eetimes.com/design/programmable-logic/4217069/Zynq-7000-EPP-sets-stage-for-new-era-of-innovations, 2011.

[5] B. Cope, P.Y.K. Cheung, W. Luk, L. Howes, "Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study", IEEE Transactions on computers, vol. 59, no. 4, pp. 433-448, 2010.

[6] S. Chey, J. Liz, J.W. Sheaffery, K. Skadrony, J. Lach, " Accelerating Compute-Intensive Applications with GPUs and FPGAs", Symposium on Application Specific Processors, pp. 101-107, Anaheim, California, USA, June 2008.

[7] M. Lakka, G. Chrysos, I. Papaefstathiou, A. Dollas, "Architecture, Design, and Experimental Evaluation of a Lightfield Descriptor Depth Buffer Algorithm on Reconfigurable Logic and on a GPU", IEEE International Symposium on Field-Programmable Custom Computing Machines, pp. 57-64, Salt Lake City, Utah, USA, May 2011.

[8] I.A. Sajid, M.M. Ahmed, M. Sagheer, "FPGA-Based Householder Implementation for Efficient Eigenvalues Calculation", International Journal of Innovative Computing, Information and Control, vol. 7, no. 10, pp. 5939-5946, 2011.

[9] S.T. Pan, C.C. Lai, B.Y. Tsai, "The Implementation of Speech Recognition Systems on FPGA-Based Embedded Systems with SOC Architecture", International Journal of Innovative Computing, Information and Control, vol. 7, no. 11, pp. 6161-6175, 2011.

[10] V. Sklyarov, I. Skliarova, D. Mihhailov, and A. Sudnitson, "Implementation in FPGA of Address-based Data Sorting", Proc. 21st Int. Conf. on Field Programmable Logic and Applications, pp. 405-410, Crete, Greece, 2011.

[11] I. Skliarova and A.B. Ferrari, "A SAT Solver Using Software and Reconfigurable Hardware", Proc. Design, Automation and Test in Europe Conference, p. 1094, Paris, France, 2002.

[12] P. Malakonakis, A. Dollas, "Exploitation of Parallel Search Space Evaluation with FPGAs in Combinatorial Problems: The Eternity II Case", Proc. 21st Int. Conf. on Field Programmable Logic and Applications, pp. 264-268, Crete, Greece, September 2011.

[13] P. Malakonakis, M. Smerdis, E. Sotiriades, A. Dollas, "An FPGA-Based Sudoku Solver based on Simulated Annealing Methods", Proc. 2009 Int. Conf. on Field-Programmable Technology, pp. 522-525, Australia, 2009.

[14] K. van der Bok, M. Taouil, P. Afratis, I. Sourdis, "The TU Delft Sudoku Solver on FPGA", Proc. 2009 Int. Conf. on Field-Programmable Technology, pp. 526-529, Australia, 2009.

[15] C. González, J. Olivito, J. Resano, "An Initial Specific Processor for Sudoku Solving", Proc. 2009 Int. Conf. on Field-Programmable Technology, pp. 530-533, Australia, 2009.

[16] M. Dittrich, T.B. Preußer, R.G. Spallek, "Solving Sudokus through an Incidence Matrix on an FPGA", Proc. 2010 Int. Conf. on Field-Programmable Technology, pp. 465-469, Beijing, China, 2010.

[17] I. Lynce, J. Ouaknine, "Sudoku as a SAT Problem", 9th Symposium on Artificial Intelligence and Mathematics, Florida, USA, January 2006.

[18] I. Skliarova, V. Sklyarov, and A. Sudnitson, Design of FPGA-based Circuits using Hierarchical Finite State Machines, TUT Press, 2012.

[19] Web Sudoku benchmarks, available at http://www.websudoku.com/.

[20] M. Feenstra, Sudoku puzzles collection, available at http://www.sudoku.ws/.

[21] Extra Challenging (Very Hard) Sudoku Puzzles, available at http://puzzles.about.com/library/sudoku/blprsudokux04.htm.

[22] 2009 Int. Conf. on Field-Programmable Technology benchmarks, available at: http://fpt09.cse.unsw.edu.au/comp/benchmarks.html.