An adaptable FPGA-based System for Regular Expression Matching

Ivano Bonesana, Marco Paolieri
ALaRI, Faculty of Informatics
University of Lugano, Lugano, Switzerland
{bonesani, paolierm}@alari.ch

Marco D. Santambrogio
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Milano, Italy
marco.santambrogio@polimi.it

Abstract

In many applications string pattern matching is one of the most intensive tasks in terms of computation time and memory accesses. Network Intrusion Detection Systems and DNA Sequence Matching are two examples. Since software solutions are not able to satisfy the performance requirements, specialized hardware architectures are required. In this paper we propose a complete framework for regular expression matching, both in its architecture and compiler. This special-purpose processor is programmed using regular expressions as programming language. With the parallelism exploited in the design it is possible to achieve a throughput greater than one character per clock cycle, requiring $O(n)$ memory space. The VHDL description of the proposed architecture is fully configurable. A design space exploration to find the optimal architecture based on area and performance cost-function is presented.

1 Introduction

Pattern matching is a well known computational intensive task and represents the core of several application domains. A *regular expression* [6] (RE), often called a *pattern*, is an expression that represents a set of strings. In networking security and QoS applications [12] [1] [11] [4] it is required to detect packets which payload matches within a set of predefined patterns. Software solutions are not fast enough to perform this task at real time, therefore dedicated hardware designs are required (e.g. as specified in [12]). Bioinformatics requires DNA sequence matching [2] [3]; that is a very computationally expensive task. To speedup software programs, several solutions have been proposed, like in [3] where the DNA sequences are compressed and a new algorithm is used. Several research groups have been studying hardware solutions: mostly based on Nondeterministic Finite Automaton (NFA). In [8] an architecture that allows extracting and sharing common subexpressions to reduce the area of the circuit, is presented. To change the regular expression to match, it is necessary to re-generate the HDL description that depends on the pattern. In [4] a NFA has been used to dynamically build efficient circuits for pattern matching. The implementation is dependent on the pattern. In [10] an FPGA implementation is proposed: it requires $O(n^2)$ memory space and processes one text character in one clock cycle. The architecture is NFA-based and requires additional time and space to rebuild its structure. Therefore, the time required for a matching operation is not constant; it can be linear in best cases but exponential in worst ones. In [11] a parallel FPGA implementation allows to increase the throughput for simultaneous matching of multiple patterns. In [2] a DNA sequence matching processor for FPGA with a Java interface is presented: parallel comparators are used, but it supports only simple RE semantics. The work proposed in [1] focuses on REs pattern matching engines implemented with reconfigurable hardware: a NFA-based implementation and a tool for automatic generation of the VHDL description are presented.

To the best of our knowledge this paper presents a novel and different approach to the pattern matching problem: REs are considered the programming language for a dedicated CPU. We do not build either Deterministic nor Nondeterministic Finite Automaton of the RE, hence we do not require additional setup time as in [10]. ReCPU - the proposed architecture - is a processor able to fetch an RE from the instruction memory and perform the matching with the text stored in the data memory. The architecture is optimized to execute computations in a parallel and pipelined way. It involves several advantages: on average it compares more than one character per clock cycle as well as it requires linear memory occupation. The processor has been developed together with a compiler inspired from the VLIW design style. This has double benefits. First, one can configure the design for achieving a particular balance of performance, area, power, etc. Second, the compiler can automatically fit the high-level language to the suitable arbitrary level of parallelism.

978-3-9810801-3-1/DATE08 © 2008 EDAA

In Figure 1 the complete working flow is shown: the user defines the architecture parameters specifying the number of parallel units. These information are used to synthesize ReCPU on the hardware design flow, and to compile the RE - provided by the user - following the software design flow.

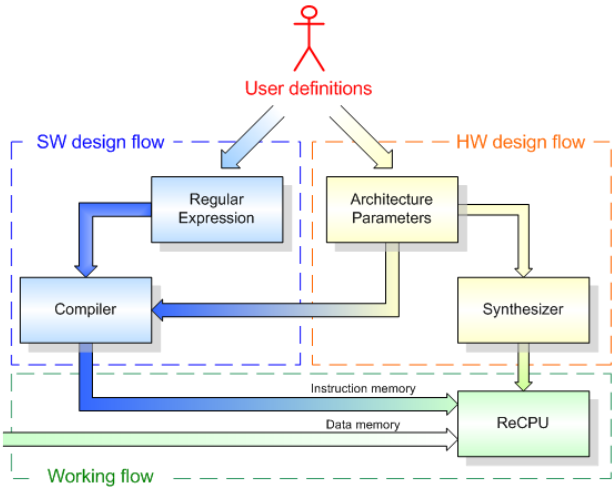

Figure 1. ReCPU Framework Flow.

In ReCPU it is possible to easily change the pattern at run-time just updating the content of the instruction memory, without modifying the underlying hardware. The state of the art approaches - [8], [4], [1] - require a re-generation of the HDL description whenever the processed regular expression changes. In comparison with the solution proposed in [8] the key advantage of our approach is the flexibility. Considering the CPU-like approach a small *compiler* is necessary to produce the binary code from the given RE, specified with the standard syntax.

The novelty introduced by our work can be found:

- in the RE programming language (Section 2);
- in the special-purpose ISA for conveying RE (Section 2);
- in the special-purpose CPU targeted for this ISA (Sections 3, 3.1, 3.2). The proposed architecture has been implemented on a reconfigurable device to be adapted to the different run-time working scenarios.

Results of synthesis on FPGA with the corresponding design space exploration for finding the optimal architecture are addressed in Section 4. Conclusions and future works are exposed in Section 5.

2 Compiler Description

ReCPU executes instructions stored in the program memory. We developed a compiler to translate REs into bitwise instructions. Section 2.1 focuses on the novel idea of considering REs an high-level programming language and Section 2.2 on the compiler structure and compilation phase.

2.1 RE as High-Level Programming Language

A program is composed by a sequence of instructions, each of those is a part of the original RE. Following this approach the final user is able to specify the RE with standard syntax [1] without the need of programming the special purpose processor at bit-level. In a generic RE single characters are considered RE that match themselves and additionally a set of operators are defined (given two REs: a and b):

- $a \cdot b$: matches strings matching a and b;
- $a|b$: strings matching either a or b ;
- $a*$: strings with zero or more occurrences of a;
- $a+$: strings with one or more occurrences of a;
- (a): define the scope and precedence of the operators.

2.2 The Compiler and the Compilation Phase

The *RE compiler* (REc) follows the VLIW approach [5], where architectural parameters are exposed to the compiler, that exploits the parallelism issuing the instructions to different parallel units. REc - similarly - is aware of the number of configurable units in the ReCPU architecture and based on this it splits the RE into different instructions. This approach corresponds to a *customizable VLIW architecture* where the designer can customize the whole framework: modifying the number of clusters in the ReCPU to obtain a different trade-off in terms of performance, power, area, etc. Once REc is updated with the new hardware parameters a different set of instructions will be generated starting from the given high-level RE. The binary code produced by the compiler is composed of *opcode* and *reference*, as shown in Figure 2. The opcode is divided into three slices: the MSB indicating an open parenthesis, the next 2-bits the internal operand (i.e. used within the characters of the *reference*), and the last bits for the external operand (i.e. loops and close parenthesis).

The compiler - written in Python - starting from the high level description of the RE generates the files to be loaded in the instruction memory and the data memory given the input text. Controls are performed to detect syntactical errors. Considering the stack-size REc computes the maximum level of nested parentheses and determines whether the architecture can execute the RE without overflowing the

[1] IEEE POSIX 1003.2, as described in [6] and [7].

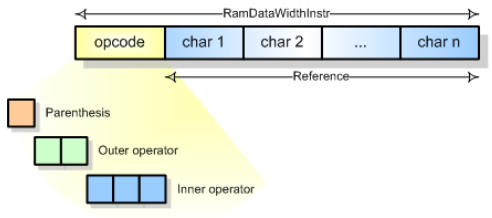

Figure 2. ReCPU Instruction Structure.

stack. The novel idea of considering REs as a programming language is clarified by the following examples: operators like $*$ and $+$ correspond to *loop* instructions. Such operators find more occurrences of the same pattern (i.e. a loop on the same RE instruction). Complex nested REs are loops on more than one instruction. The loop terminates whenever the matching of the current instruction fails.

Nested parentheses (e.g. $(((ab)*(c|d))|(abc))$) are managed as *function calls*. An open parenthesis inside an RE is a *call* instruction while the closed one is a *return*: in the former the status of the processor is pushed into the stack while in the latter is returned. Using such approaches we can tackle very complex REs by means of well and widely known processor techniques.

3 Architecture Description

The block diagram of ReCPU is shown in Figure 3. This section describes the microarchitecture: *Data Path* in Section 3.1 and *Control Unit* in Section 3.2. ReCPU uses two separate memories: data for the input text and instruction for the RE. One of the main features of ReCPU is the execution of more than one character comparison per clock cycle. The architecture is adaptable, i.e. the designer can specify the number of parallel units: to adapt it to the requirements it is necessary to perform a trade-off in terms of performance, area, power, etc. To find the optimal architecture a *cost-function* has been defined and a *Design Space Exploration* has been carried out (see Section 4).

3.1 Data Path

In the Data Path are placed all the different parallel comparators organized in *Clusters*. Each comparator unit compares a character from the input text with one from the pattern. The total number of elements in a cluster - indicated as *ClusterWidth* - is the number of characters that can be compared every clock cycle if a sub-RE is matching. This parameter influences the throughput in case the pattern starts matching the input text. The processor is composed of several *Clusters* - the total number is indicated as *NCluster* - that are used to compare a sub-RE starting by shifted position of the input text. This influences the throughput in case the pattern is not matching. Some architectural techniques - pipelining, data and instructions prefetching, multiple memory ports - are used to increase the parallelism achieving throughput of more than one character per clock cycle. The pipeline, controlled by the *Control Unit*, is composed by two stages: *Fetch/Decode* and *Execute*. Due to the regular control flow of the instructions a good prediction technique - i.e. duplicated instruction-fetching units - avoids stalls resulting in a reduction of the execution latency with a consequent performance improvement. In *Fetch/Decode* stage, the two instruction buffers load two sequential instructions: when an RE starts matching, one buffer prefetches the next instruction and the other keeps a *backup* of the first one. If the RE fails the *backup* instruction is used without stalling the pipeline. Parallel data buffers and duplicated inner registers reduce the latency to access the data memory. The second stage of the pipeline - fully combinatorial - is the *Execute*. The reference forwarded by the previous stage is compared with the data read from the RAM. The configurable number of comparator clusters is shown in Figure 4. Each cluster is shifted by one character from the previous to process a wider set of data in a single clock cycle. The results of each cluster are collected and evaluated by the *Engine*, that produces a match/not match signal that is analyzed by the *Control Unit*.

We designed a fully configurable VHDL implementation that allows to modify some architectural parameters. This way it is possible to define the best architecture fullfilling the user requirements, to find a good trade-off between timing, area constraints and desired performance.

3.2 Control Unit

A RE is defined as a sequence of instructions (i.e. a set of conditions to be satisfied). Whenever all the instructions of an RE are matched, the whole RE is satisfied. The *Data Path* cannot identify the result of the entire RE and it cannot request data or instructions to the external memories: the execution is managed by the *Control Unit* (see Figure 3): the core part is the *Finite State Machine* (FSM) shown in Figure 5.

In a not-matching condition, the same instruction address is fetched while the data address is incremented every clock cycle. The comparisons are exploited by all the clusters in the *Data Path*. Thus, several characters are compared simultaneously. If an RE starts matching, ReCPU switches to the matching mode.

In this scenario, a single cluster is used. Still more than one character comparison per clock cycle is achieved by means of the different comparators of the cluster. In this state if one of the instructions of the RE fails, the exe-

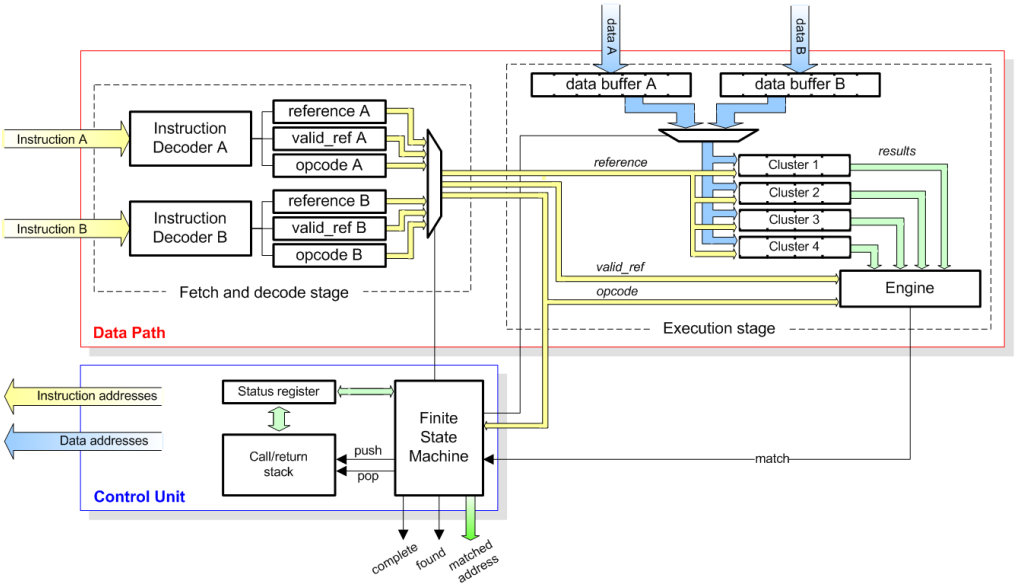

Figure 3. Block diagram of ReCPU with 4 Clusters, each of those has a ClusterWidth of 4.

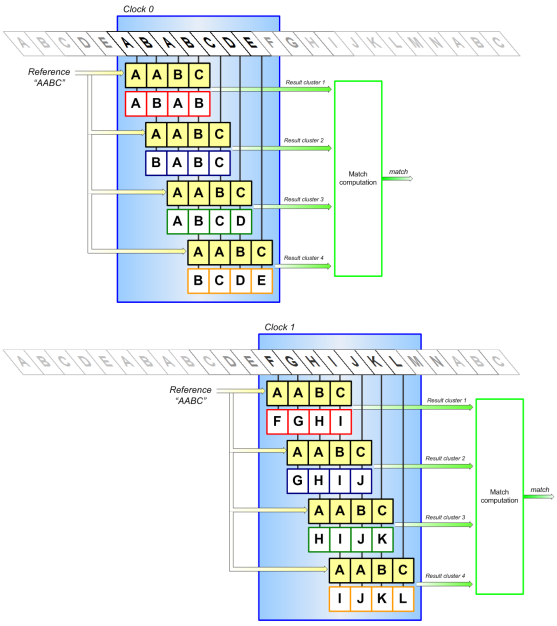

Figure 4. Comparator clusters working on an input text. The top and bottom pictures correspond to two subsequent clock cycles.

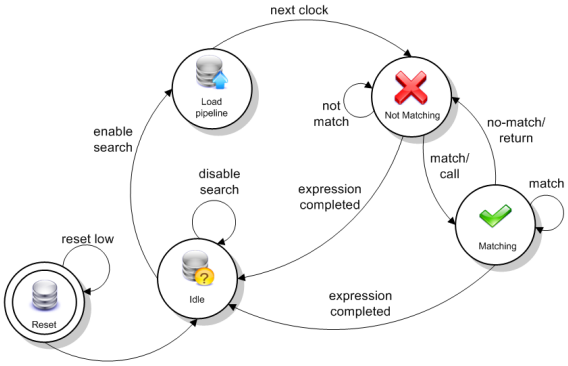

Figure 5. Finite State Machine of the *Control Unit*.

cution has to be restarted from the address where the RE started to match. Whenever a `NOP` instruction is detected the RE is considered complete. ReCPU outputs indicate if and at which address of the data memory the RE matched successfully. Loops (i.e. + or * operators) are a special cases, treated with a `call/return` function-paradigm. As soon as an open parenthesis (i.e. a *call*) is decoded, the *Control Unit* pushes the current matching status, the program counter and the current internal operator in the stack. The computation continues normally until a return instruction (i.e. a closed parenthesis followed by an operator) is fetched. The previous context is restored and the overall matching value is updated. If the not matching condition is verified while processing a *call*, the stack is flushed, the whole RE is considered as failed and the computation restarts.

4 Experimental Results

In [9], the proposed architecture has been synthesized using Synopsys Design Compiler on STMicroelectronics HCMOS8 ASIC technology library featuring 0.18μm silicon process: experimental results (area and time constraints) have been shown for an architecture with *NCluster* and *ClusterWidth* equal to 4. In [9] the proposed architecture has been compared with other state of the art solutions, showing an average performance increase of 9.1 times respect to the most commonly used software tool. Table 1 shows the speedup of the architecture with respect to *grep*[2].

Table 1. Performance comparison between *grep* and ReCPU on a text file of 65K characters.

Pattern	*grep*	ReCPU	Speedup
$E\|F\|G\|HAA$	19.1 ms	32.7 μs	584.8
$ABCD$	14.01 ms	32.8 μs	426.65
$(ABCD)+$	26.2 ms	393.1 μs	66.74

A ReCPU constant performance index of the time required to process a character cannot be determined due to the dependence on the input data and the RE. Different scenarios are possible (each of those has its performance figure) - where the input text is:

1. not matching current instruction with $\cdot$ operator;
2. not matching current instruction with $|$ operator;
3. matching current instruction with any operator.

The corresponding time per character (expressed in $ns/char$) figures are computed by the following formulas:

1. $$T_{cnm} = \frac{T_{cp}}{NCluster + ClusterWidth - 1} \quad (1)$$
2. $$T_{onm} = \frac{T_{cp}}{NCluster} \quad (2)$$
3. $$T_m = \frac{T_{cp}}{ClusterWidth} \quad (3)$$

The time per character can be used to compute the bit-rate[3] B_x:

$$B_x = \frac{1}{T_x} \cdot 8 \cdot 10^9 \quad (4)$$

where T_x is either (1), (2) or (3).

In this paper we present a complete Design Space Exploration (DSE) to define the optimal architecture (synthesized on different Xilinx FPGAs). This represents an outgoing work towards the exploration of a reconfigurable solution. We changed the structure modifying the number of parallel units - i.e. `NCluster` and `ClusterWidth` - in the completely adaptable VHDL description. Finally we analyzed how area and performance were scaling. We exploited the DSE using an `NCluster` between $\{2, 4, 8, 16, 32, 64\}$ and `ClusterWidth` equal to $\{8\}$. Increasing the number of `NCluster` more characters are checked in parallel, and so ReCPU results to be faster whenever the pattern is not matching the input text. Due to the higher hardware complexity the *critical path* increases and the maximum possible clocking frequency decreases. On the other side, a bigger `ClusterWidth` corresponds to much better performance whenever the input string starts matching the RE because a wider sub-expression (i.e. an instruction) is processed in a single clock cycle. The FPGA synthesis results are shown in Table 2.

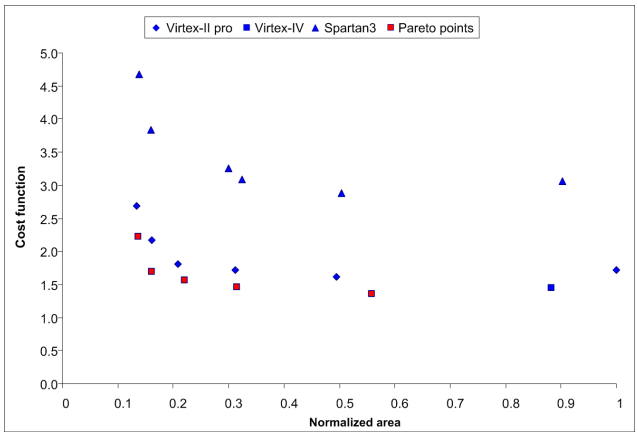

Figure 6. Area and performance cost function values for different ReCPU configurations.

To evaluate different configurations that have been synthesized and listed in Table 2, we defined a *cost function* that takes into account the previous scenarios and considers the consequences in terms of area and performance:

$$costf = p_1 \cdot T_{cnm} + p_2 \cdot T_{onm} + p_3 \cdot T_m \quad (5)$$

The function $costf(\cdot)$ evaluates the different performance indexes with a corresponding probability for different cases. We consider that the probability of having an *and* operator in the current instruction is 0.5 as well as the one of having an *or*. Among these cases there is respectively the 0.25 probability of matching the pattern and 0.25 of not matching. We consider all the cases equiprobable. The $costf$ is the resulting average time per character based on the previous probabilities. p_1 is the probability of having an *and* operator and the pattern is not matching (0.25), p_2 an

[2] www.gnu.org/grep

[3] The bit-rate represents the number of bits processed in a second, it is computed considering 1 char = 8 bits.

Ncluster	ClusterWidth	RamWidthData	RamWidthInstr	FPGA	Technology	Critical Path ns	Max Freq MHz	Area EU	Norm. Area	Tcnm	Tonm	Tm	cost
2	8	10	8	Virtex-II pro	xc2vp30-fg676-7	8.9	111.9	6082	0.133	0.993	4.469	1.1	1.92
4	8	12	8	Virtex-II pro	xc2vp30-fg676-7	9.7	102.9	7379	0.162	0.884	2.431	1.2	1.44
8	8	16	8	Virtex-II pro	xc2vp30-fg676-7	10.1	99.2	9508	0.209	0.672	1.26	1.3	1.11
16	8	24	8	Virtex-II pro	xc2vp30-fg676-7	11.2	89.6	14254	0.313	0.485	0.697	1.4	0.99
32	8	40	8	Virtex-II pro	xc2vp30-fg676-7	11.6	86.3	22532	0.494	0.297	0.362	1.4	0.89
64	8	72	8	Virtex-II pro	xc2vp30-fg676-7	12.9	77.1	45573	1	0.183	0.203	1.6	0.91
2	8	10	8	Virtex-IV	xc4vsx35-ff668-12	7.42	134.778	6218	0.136	0.824	3.71	0.9	1.6
4	8	12	8	Virtex-IV	xc4vsx35-ff668-12	7.595	131.658	7327	0.161	0.69	1.899	0.9	1.12
8	8	16	8	Virtex-IV	xc4vsx35-ff668-12	8.709	114.818	10030	0.220	0.581	1.089	1.1	0.96
16	8	24	8	Virtex-IV	xc4vsx35-ff668-12	9.494	105.329	14335	0.315	0.413	0.593	1.2	0.84
32	8	40	8	Virtex-IV	xc4vsx35-ff668-12	9.66	103.52	25470	0.559	0.248	0.302	1.2	0.74
64	8	72	8	Virtex-IV	xc4vsx35-ff668-12	10.878	91.932	40220	0.883	0.153	0.17	1.4	0.76
2	8	10	8	Spartan3	xc35400-fg456-5	15.602	64.096	6283	0.138	1.734	7.801	2	3.36
4	8	12	8	Spartan3	xc35400-fg456-5	17.182	58.199	7287	0.160	1.562	4.296	2.1	2.54
8	8	16	8	Spartan3	xc35400-fg456-5	18.198	54.951	10078	0.221	1.213	2.275	2.3	2.01
16	8	24	8	Spartan3	xc35400-fg456-5	20.096	49.762	14776	0.324	0.874	1.256	2.5	1.79
32	8	40	8	Spartan3	xc35400-fg456-5	20.595	48.556	22965	0.504	0.528	0.644	2.6	1.58
64	8	72	8	Spartan3	xc35400-fg456-5	23.086	43.316	41131	0.903	0.325	0.361	2.9	1.61

Table 2. Results after synthesizing ReCPU with different parameters on Xilinx FPGAs.

or operator that does not match the pattern (0.25) and p_3 a matching with any operator (0.5). The optimal cost function can be different according to the characteristics of the text and the RE. The goal of this work is to define a complete framework, not the best cost function.

The plot of Figure 6 has on the X-axis the area occupied while on Y-axis the cost function $costf(\cdot)$. It allows to select the best architecture according to area and performance requirements. The three sets of points correspond to different FPGA architectures. It is easily possible to identify which are the optimal Pareto points and which are the dominated ones.

5 Conclusions and Future Works

Nowadays pattern matching is one of the main computational and memory intensive tasks. Software solutions do not meet the performance requirements, so special hardware architectures are proposed. We propose a novel architecture: a special-purpose processor for regular expression matching with the definition of a regular expression programming language. This approach enables the possibility of overcoming the current state of the art performance limit of one character comparison per clock cycle. The architecture is based on a configurable number of comparators and the compiler is aware of the underlying architecture (following the VLIW paradigm). We implemented ReCPU on reconfigurable devices, such as Xilinx FPGAs, and we carried out a Design Space Exploration to determine the best architecture according to the defined cost function: a trade-off in terms of performance, area, etc. We have shown in Table 2 the results of the synthesis on three different FPGAs: *Virtex-II pro*, *Virtex-IV*, *Spartan3*. In the plot of Figure 6 it is possible to identify - according to the defined cost function - which are the optimal configurations.

Future works are focused on the definition of a reconfigurable version of the ReCPU based on FPGA-devices. This way, we could dynamically reconfigure the architecture at run-time achieving the best performance for the given RE. We would also like to explore some optimizations for the REc compiler.

References

[1] J. C. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Proc. FPT*, pages 119–126, Dec. 2006.

[2] B. O. Brown, M.-L. Yin, and Y. Cheng. DNA sequence matching processor using FPGA and JAVA interface. In *Proc. IEEE EMBS*, Sep. 2004.

[3] L. Chen, S. Lu, and J. Ram. Compressed pattern matching in DNA sequences. In *Proc. CSB*, 2004.

[4] Y. H. Cho and W. H. Mangione-Smith. A pattern matching coprocessor for network security. In *Proc. DAC*, pages 234–239, San Diego, CA, USA, 2005.

[5] J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, Dec. 2004.

[6] J. Friedl. *Mastering Regular Expressions*. O'Reilly Media, 3 edition, Aug. 2006.

[7] GNU, USA. *Grep Manual*, Jan 2002.

[8] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang. Optimization of regular expression pattern matching circuits on FPGA. In *DATE*, pages 12–17, 2006.

[9] M. Paolieri, I. Bonesana, and M. D. Santambrogio. ReCPU: a parallel and pipelined architecture for regular expression matching. In *Proc. IFIP-VLSI*, Oct. 2007.

[10] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *Proc. FCCM*, Apr. 2001.

[11] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion. In *Proc. ICFLA*, Lisbon, Portugal, Sep. 2003.

[12] M. Yadav, A. Venkatachaliah, and P. Franzon. Hardware architecture of a parallel pattern matching engine. In *Proc. ISCAS*, Apr. 2007.