ReCPU: a Parallel and Pipelined Architecture for Regular Expression Matching

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

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

ABSTRACT

Text pattern matching is one of the main and most computation intensive parts of systems such as Network Intrusion Detection Systems and DNA Sequencing Matching. Software solutions to this are available but often they do not satisfy the requirements in terms of performance. This paper presents a new hardware approach for regular expression matching: ReCPU. The proposed solution is a parallel and pipelined architecture able to deal with the common regular expression semantics. This implementation based on several parallel units achieves a throughput of more than one character per clock cycle (maximum performance of current proposed solution) requiring just $O(n)$ memory locations (where n is the length of the regular expression). Performance has been evaluated synthesizing the VHDL description. Area and time constraints have been analyzed. Experimental results are obtained simulating the architecture.

1. INTRODUCTION

Searching for a set of strings that match a given pattern is a well known computation-intensive task, exploited in several different application fields. Software solutions cannot always meet the requirements in terms of speed. Nowadays there is an increasing need of high performance computing - as in the case of biological sciences. Matching a DNA pattern among millions of sequences is a very common and computationally expensive task in the Human Genome Project. In Network Intrusion Detection Systems - where regular expressions are used to identify network attack patterns - software solutions are not acceptable because they would slow down the entire system. Such applications require a different approach.

To move towards a full hardware implementation - overcoming the performance achievable with software - it is reasonable for these application domains.
Several research groups have been studying hardware architectures for regular expressions matching: mostly based on Non-deterministic Finite Automaton (NFA) as described in [1] and [2].

In [1] an FPGA implementation is proposed. It requires $O(n^2)$ memory space and processes a text character in $O(1)$ time (one clock cycle). The architecture is based on hardware implementation of Non-deterministic Finite Automaton (NFA); additional time and space are necessary to build the NFA structure starting from the given regular expression. The time required is not constant, it can be linear in best cases and exponential in worst ones. We do not face with these limitations because we are able to store regular expressions using $O(n)$ memory locations. We do not require any additional time to start to process the regular expressions (from now on RE). In [2] an architecture that allows extracting and sharing common sub-regular expressions, in order to reduce the area of the circuit, is presented. It is necessary to re-generate the HDL description to change the regular expression. It is clear that this approach generates an implementation dependent from the pattern. In [3] a software that translates a RE into a circuit description has been developed. A Non-deterministic Finite Automaton has been utilized to dynamically create efficient circuits for pattern matching (that have been specified with a standard rule language).

The work proposed in [4] focuses on REs pattern matching engines implemented with reconfigurable hardware. A Non-deterministic Finite Automaton based implementation is used, and a tool for automatic generation of the VHDL description has been developed. All these approaches - [2], [3], [4] - require a new generation of the HDL description whenever a new regular expression needs to be processed. In our solution we just require to update the instruction memory with the new RE. In [5] a parallel FPGA implementation is described: multiple comparators allow to increase the throughput for parallel matching of multiple patterns.

In [6] a DNA sequence matching processor using FPGA and Java interface is presented. Parallel comparators are used for the pattern matching. They do not implement the regular expression semantics (i.e. complex operators) but just simple text search based on exact string matching.

At the best of our knowledge this paper presents a different approach to the pattern matching problem: REs are considered the programming language for a dedicated CPU. We do not build either Deterministic or Non-deterministic Finite Automaton of the RE, hence not requiring additional setup time as in [1]. 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. This approach involves several advantages: on average it compares more than one character per clock cycle as well as it requires less memory occupation: for a given RE of size n the memory required is just $O(n)$. In our solution it is easily possible to change the pattern at run-time just updating the content of the instruction memory without modifying the underlaying hardware. Considering the CPU-like approach a small *compiler* is necessary to obtain the machine code from the given RE (i.e.

specified with a high-level description).

In Section 2 a brief overview of Regular Expressions focusing on the semantics - that have been implemented in hardware - is provided. The idea of considering regular expressions as a programming language is fully described in Section 3. Section 4 describes in a top-down manner the hardware architecture. Data Path is fully covered in 4.1 and Control Path in 4.2. Results of the synthesis process in terms of critical path and area are discussed in Section 5: a comparison of the performance with other solutions is proposed. Conclusions and future works are addressed in Section 6.

2. REGULAR EXPRESSION OVERVIEW

A *regular expression* [7] (RE), often called a *pattern*, is an expression that matches a set of strings. REs are used to perform searches over text data, and are commonly present in programming languages and text editors for text manipulation. In an RE single characters are considered regular expressions that match themselves and additionally several operators are defined. Let us consider two REs: a and b, the operators that have been implemented in our architecture follow:

- $a \cdot b$: it matches all the strings that match a and b;
- $a|b$: matches all strings that match either a or b ;
- $a*$: matches all strings composed by zero or more occurrences of a;
- $a+$: matches all strings composed by one or more occurrences of a;
- (a): parentheses are used to define the scope and precedence of the operators (e.g. to match zero or more occurrences of a and b it is necessary to define the following RE: $(ab)*$).

3. PROPOSED APPROACH

The innovative idea we developed is to use REs as instructions for the ReCPU processor. ReCPU executes a program stored in the instruction memory. The program is composed by a set of instructions, each of those is part of the original RE. The approach followed is the same used in general purpose processors: a program is coded in a high-level language (e.g. C) and compiled (e.g. using gcc) into low level language (i.e. machine code). In our case an RE is defined using the common high-level language - as described in [7] or in [8] - it is compiled[1] into machine code and executed by ReCPU processor. Given a RE a *compiler* is used to generate the instructions sequence that are executed by the core of the architecture on the text stored in the data memory. The compiler program does not need to perform many optimizations due to the simplicity of the RE programming language. The binary code produced by the compiler and composed of *opcode* and *operands* instructs the *execution unit* using the information about the number of parallel units of the architecture. Our compiler uses the idea behind the VLIW architecture design style: the parallel units are exposed to the back-end (i.e. the compiler issues as many character comparisons as the number of parallel comparators present in the architecture).

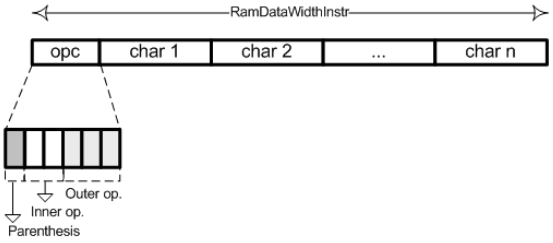

Figure 1: Instruction Structure

Let us analyze some examples to clarify the mapping of complex RE operators into the programming language: operators like $*$ and $+$ correspond to *loop* instructions. Such operators find more occurrences of the same pattern (i.e. looping on the same RE instruction). This technique guarantees the possibility to handle complex REs looping on more than one instruction. The loop terminates whenever the pattern matching fails. In case of $+$ at least one valid iteration of the loop is required to validate the RE, while for $*$ there is no limitation on the minimum number of iterations.

Another feature of complex REs that can be perfectly managed considering the RE as a programming language is the use of nested parentheses (e.g. $(((ab)*(c|d))|(abc))$). We mapped this to the *function call* paradigm of all the common programming languages. We treat an open parenthesis in an RE as a function call and - as in common processors - once the context (all the control path internal registers) is saved in a stack data structure the execution can continue normally. Whenever a close parenthesis is found a stack-pop operation is performed and the validity of the RE is checked combining the current operator, the current context and the context popped from the stack. This way this architecture can tackle very complex nested REs using a well and widely known function paradigm approach.

ReCPU binary instructions generated by the compiler are stored into the instruction memory. An instruction is composed by opcode and operands (i.e. the characters present in the pattern) as shown in Figure 1. The opcode is composed by three different parts: the MSB represents the use of a parenthesis (i.e. a function call), the next 2-bits represent the internal operands (i.e. *and* or *or* used to match the characters present in the current instruction) and the last bits select the external operand used to describe loops and close parenthesis (i.e. a *return* after a function call). A RE is completly matched whenever a NOP instruction is fetched from the instruction memory. A complete list of opcodes is shown in Table 1[2].

4. ARCHITECTURE DESCRIPTION

The ReCPU has been designed applying some well known computer architectural paradigms to provide a high throughput by limiting the number of stall conditions (that are the

[1]We developed a compiler that translates the high level description of the RE to the ReCPU machine code. As explained the compiler takes advantage of some well known VLIW techniques.

[2]Please notice that *don't care* values are expressed as "-".

Table 1: Bitwise representation of the opcodes.

opcode			RE
0	00	000	nop
1	--	---	(
0	01	---	and
0	10	---	or
0	--	001	)*
0	--	010	)+
0	--	011	)|
0	--	1--	)

largest waste of computation time during the execution). This section overviews the structure of ReCPU - shown in the block diagram of Figure 2 - focusing on the microarchitectural implementation. A more detailed description of the two main blocks the *Data Path* and the *Control Path*, is provided.

ReCPU has a Harvard based architecture that uses two separate memory banks: one storing the text and the other one the instructions (i.e. the RE). Both RAMs must be dual port to allow parallel accesses. In the *Data Path* subsection the use of parallel buffers is described.

The main idea proposed by this paper is the execution of more than one character comparison per clock cycle. To achieve this goal several parallel comparators - grouped in units called *Clusters* - are placed in the *Data Path*. Each comparator compares an input text character with a different one from the pattern. The number of elements of the cluster is indicated as *ClusterWidth* and it represents the number of characters that can be computed every clock cycle whenever a sub-RE is matching. This figure is influencing the throughput whenever a part of the pattern starts matching the input text. The architecture is composed by several *Clusters* - the total number is indicated as *NCluster* - used to compare a sub-RE starting by shifted position of the input text. This influences the throughput whenever the pattern is not matching.

4.1 Data Path

In order to process more than one character per clock cycle, we applied some architectural techniques to increase the parallelism of ReCPU: pipelining, data and instructions prefetching, and use of parallel memory ports.

The pipeline composed by two stages: *Fetch/Decode* and *Execute*. The *Control Path*, as explained in the next section, spends one cycle to precharge the pipeline and then it starts performing the prefetching mechanism. In each stage we introduced duplicated buffers to avoid stalls. This approach was advantageous because the parts we replicated and the corresponding control logic are not so complex, leading to an acceptable increase in terms of area, while no overhead in terms of time constraints is present since they work in parallel. Hence, we have a reduction of the execution latency with a consequent performance improvement.

Due to the regular instruction control flow a good prediction technique with duplicated instruction fetching structures is able to avoid stalls. Indeed, considering the *Fetch/Decode* stages, the two instruction buffers load two sequential instructions: when an RE starts matching, one buffer is used to prefetch the next instruction and the other is used as *backup* of the first one. In case that the matching process fails (i.e. prefetching is useless) the first instruction (i.e. the backup one) can be used without stalling the pipeline. Similarly, the parallel data buffers reduce the latency of the access to the data memory.

According to this design methodology in the *Fetch/Decode* stage the decoder and the pipeline registers are duplicated. By means of a multiplexer, just one set of pipeline register values are forwarded to the *Execution* stage. As shown in the diagram, the multiplexer is controlled by the *Control Path*. The decode process extracts from the instruction the reference string (i.e. the characters of the pattern that must be compared with the text), its length (indicated as `valid_ref` and necessary because the number of characters composing the sub-RE can be lower than the width of the cluster) and the operators used.

The second stage of the pipeline is the *Execute*. It is a fully combinatorial circuit. The reference coming from the previous stage is compared with the data read from the RAM and previously stored in one of the two parallel buffers. Like in case of *Fetch/Decode* stage this technique (see Figure 2) reduces the latency of the access to the memory avoiding the need of a stall if a jump in the data memory is required[3].

We implemented the comparison using a configurable number of arrays of comparators. This is shown in details in Figure 3. Each cluster is shifted of one character from the previous in order to cover a wider set of data in a single clock cycle. The results of each comparator cluster are collected and evaluated by the block called *Engine*. It produces a match/not match signal going into the *Control Path*.

Our approach is based on a fully-configurable VHDL implementation. It is possible to modify some architectural parameters such as: number and dimensions of the parallel comparator units (*ClusterWidth* and *NCluster*), width of buffer registers and memory addresses. This way it is possible to define the best architecture according to the user requirements, finding a good trade-off between timing, area constraints and desired performance.

4.2 Control Path

We define an RE as a sequence of instructions that actually represent a set of conditions to be satisfied. If all the instructions of an RE are matched, then the RE itself is matched. The ReCPU *Data Path* fetches the instruction, decodes it and verifies whether it matches the current part of the text or not. But it cannot identify the result of the whole RE. Moreover the *Data Path* does not have the possibility to request data or instructions from the external memories.

To manage the execution of the RE we designed a *Control Path* block containing some specific hardware structures that we are going to describe in the current section. The core of the *Control Path* is the *Finite State Machine* (FSM) shown in Figure 4. The execution of an RE requires two input addresses: the RE start address and the text start

[3]A jump in the data memory is required whenever one or more instructions are matching the text and then the matching fails (because the current instruction is not satisfied). In this case a jump in the data memory address restarts the search from the address where the first match occurred.

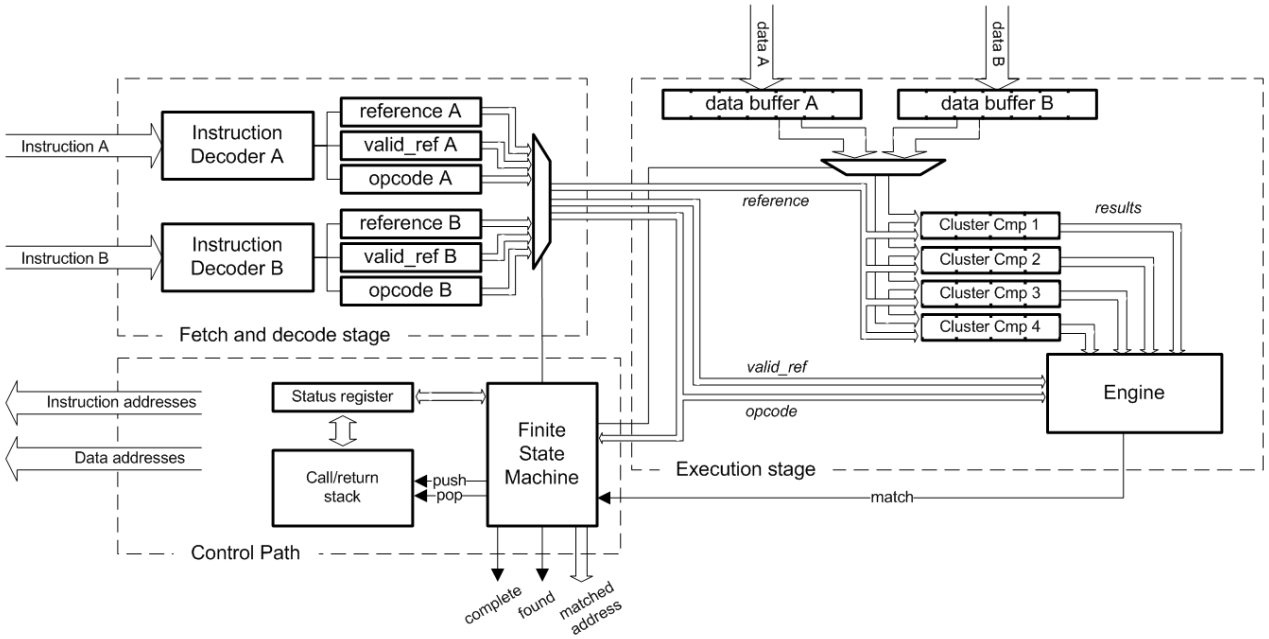

Figure 2: Block diagram of ReCPU with 4 Clusters, each of those has a ClusterWidth of 4. The main blocks are: Control Path and Data Path (composed by a Pipeline of Fetch/Decode and Execution stages).

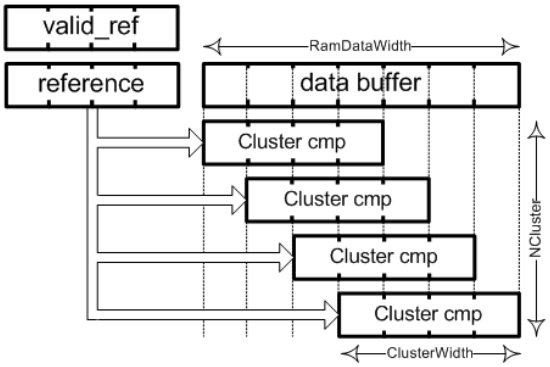

Figure 3: Detail of comparator clusters.

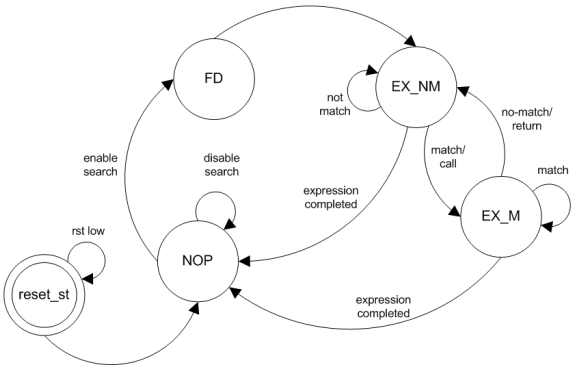

Figure 4: Finite state machine of the *Control Path*.

address. The FSM is designed in such a way that after the preload of the pipeline, two execution cases can occur. When the first instruction of an RE does not match the text, the FSM loops in the `EX_NM` state, as soon as a match is detected the FSM goes into the `EX_M` state.

Not matching the text, the same instruction address is fetched and the data address advances performing the comparison by means of the clusters inside of the *Data Path*. If no match is detected the data memory address is incremented by the number of clusters. This way several characters are compared every single clock cycle leading to a throughput i.e. clearly more than one character/cc.

When an RE starts matching, the FSM goes into *EX_M* state and the ReCPU switches to the matching mode by using a single cluster comparator to perform the pattern matching task on the data memory. As for the previous case more than one character per clock cycle is checked by the different comparators of a cluster. When the FSM is in this state and one of the instructions composing the RE fails the whole process has to be restarted from the point where RE started to match.

In both cases (matching or not matching), whenever a `NOP` instruction is detected the RE is considered complete, so the FSM goes into the NOP state and the result is computed. The ReCPU returns a signal that indicates the match of the RE and the address of the memory location containing the first character of the matched string.

A particular case is represented by loops (i.e. + or * operators). We treat these operators with a `call` and `return` paradigm. When an open parenthesis is detected a call is performed: the *Control Path* saves the content of the status

register (i.e. the actual matching value, the current program counter for instruction memory and the current internal operator) in a stack. The RE is then computed normally until a return operand is detected. A return is basically a closed parenthesis followed by +, * or |. It restores the old context and updates the value of the global matching. If a not matching condition is verified while the FSM is processing a call, the stack is erased and the whole RE is considered not matching. The process is restarted as in the simple not matching case.

Problems of overflow in the number of elements stored in the stack are avoided by the compiler. It knows the stack-size and computing the maximum level of nested parentheses is able to determine if the architecture can execute the RE or not.

5. EXPERIMENTAL RESULTS

ReCPU has been synthesized using Synopsys Design Compiler[4] on the STMicroelectronics HCMOS8 ASIC technology library featuring 0.18μm silicon process. Validation of the proposed architecture has been exploited setting *NCluster* and *ClusterWidth* equal to 4. The synthesis results are presented in Table 2:

Table 2: Synthesis results for the ReCPU architecture with NCluster and ClusterWidth set to 4.

Critical Path (ns)	Area (μm^2)	Max Clock Frequency (MHz)
3.14	51082	318.47

Papers described in Section 1 show a maximum clock frequency between 100MHz and 300MHz. The results of the table show how our solution is competitive with the others having the advantage of processing in average more than one character per clock cycle (i.e. the case for all the other solutions like [1] and [2]).

We analyze different scenarios to figure out the performance of our implementation: whenever the input text is not matching the current instruction and the opcode represents a $\cdot$ operator, the maximum level of parallelism is exploited and the performance in terms of time required to process a character are up to:

$$T_{cnm} = \frac{T_{cp}}{NCluster + ClusterWidth - 1} \quad (1)$$

where the T_{cnm}, expressed in $ns/char$, depends on the number of clusters, the width of the clusters and the critical path delay T_{cp}. If the input text is not matching the current instruction and the opcode is a | then the performance are given by the following formula:

$$T_{onm} = \frac{T_{cp}}{NCluster} \quad (2)$$

If the input text is matching the current instruction then the performance depends on the width of one cluster (all the other clusters are not used):

$$T_m = \frac{T_{cp}}{ClusterWidth} \quad (3)$$

For each different scenarios, using the time per character computed using the formulas (1), (2) and (3) it possible to compute the corresponding bit-rate evaluating the achievable performance. The bit-rate B_x represents the number of bits[5] processed in one second and can be computed as follows:

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

where T_x represents one of the quantities resulting from (1), (2) and (3).

The numerical results for the implementation we have synthesized are shown in Table 3:

Table 3: Time necessary to process one character and corresponding bit-rate for the synthesized architecture.

T_{cnm} ns/char	T_{onm} ns/char	T_m ns/char	B_{cnm} GBit/s	B_{onm} GBit/s	B_m GBit/s
0.44	0.78	0.78	18.18	10.19	10.19

The results summarized in Table 3 represent the throughput achievable in different scenarios. Whenever there is a function call (i.e. nested parentheses) one additional clock cycle of latency is required. The throughput of the proposed architecture really depends on the RE as well as on the input text so it is not possible to compute a fixed throughput but just provide the performance achievable in different cases.

In our experiments we compared ReCPU with the popular software *grep*[6] using three different text files of 65K characters each. For those files we chose a different content trying to stress the behavior of ReCPU. We ran *grep* on a Linux Fedora Core 4.0 PC with Intel Pentium 4@2.80GHz, 512MB RAM measuring the execution time with Linux *time* command and taking as result the *real* value. The results are presented in Table 4.

Table 4: Performance comparison between grep software and ReCPU on a text file of 65K characters.

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

We notice that if loop operator are not present our solution performs equal either with more than one instruction and OR operator or with a single AND instruction (see the first two entries of the table). In these cases the speedup is more than 400 times, achieving extremely good results with respect to software solution. In case of loop operator it is

[4] www.synopsys.com

[5] It is computed considering that 1 char = 8 bits.

[6] www.gnu.org/software/grep

possible to notice a slow-down in the performance but still achieving a speedup of more than 60.

To prove performance improvements of our approach with respect to the other published solutions, we compare the bit-rates as described in the Table 5. It was not possible to compare the bit-rate for [6], [2] because this quantity was not published in the papers.

Table 5: Bit-Rate comparison between literature solutions and ReCPU.

Solution published in	bit-rate GBit/s	ReCPU GBit/s	Speedup factor (x)
[4]	(2.0, 2.9)	(10.19, 18.18)	(5.09, 6.26)
[3]	(1.53, 2.0)	(10.19, 18.18)	(6.66, 9.09)
[5]	(5.47, 8.06)	(10.19, 18.18)	(1.82, 2.25)
[1]	(0.45, 0.71)	(10.19, 18.18)	(22, 25)

In Table 5 the bit-rate range for different solutions is shown. We compared it with the one of ReCPU computing a speedup factor that underlines the speedup of our approach. It is shown that the performance achievable with our solution is n times faster than the other published research works. Our solution guarantees several advantages apart from the bit-rate improvement: $O(n)$ memory locations are necessary to store the RE and it is possible to modify the pattern at run-time just updating the program memory. It is interesting to notice - analyzing the results in the table - that in the worst case we are performing pattern matching almost two times faster.

6. CONCLUSIONS AND FUTURE WORKS

Nowadays the need of high performance computing is growing up. An example of this is represented by biological sciences (e.g. Humane Genome Project) where DNA sequence matching is one of the main applications. To increase the performance it is better to get advantage of hardware solutions for the pattern matching problem. In this paper we presented a novel approach for hardware implementation of regular expression matching. Our contribution regards a completely different approach of dealing with the regular expressions. REs are considered as a programming language for a parallel and pipelined architecture. This guarantees the possibility of changing the RE at run-time just modifying the content of the instruction memory and it involves a high improvement in terms of performance. Some features, like the multiple characters checking, instructions prefetching and parallelism exposure to the compiler level are inspired to the VLIW design style.

The current state of the art guarantees a fixed performance of one character per clock cycle. Our goal was to figure out a way of extract some parallelism to achieve in average much better performance. We proposed a solution that has a bit-rate of at least 10.19 GBit/s with a peak of 18.18 GBit/s.

Future works are focused on the definition of a reconfigurable version of the proposed architecture based on FPGA-devices. We could, this way, exploit the possibility to dynamically reconfigure the architecture at run-time. The study of possible optimizations of the *Data Path* to reduce the critical path and increase the maximum possible clock frequency is an alternative. We would also like to explore the possibility of adding some optimization in the compiler side.

7. REFERENCES

[1] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01)*, April 2001.

[2] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of regular expression pattern matching circuits on FPGA," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 12–17.

[3] Y. H. Cho and W. H. Mangione-Smith, "A pattern matching coprocessor for network security," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*. New York, NY, USA: ACM Press, 2005, pp. 234–239.

[4] J. C. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *IEEE International Conference on Field Programmable Technology (FPT)*, December 2006, pp. 119–126.

[5] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10gbps fpga-based network intrusion," in *International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, September 2003.

[6] B. O. Brown, M.-L. Yin, and Y. Cheng, "DNA sequence matching processor using FPGA and JAVA interface," in *Annual International Conference of the IEEE EMBS*, September 2004.

[7] J. Friedl, *Mastering Regular Expressions*, 3rd ed. O'Reilly Media, August 2006.

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