ReCPU: a Parallel and Pipelined Architecture for Regular Expression Matching

Marco Paolieri[1], Ivano Bonesana[1], and Marco Domenico Santambrogio[2]

[1] ALaRI, Faculty of Informatics
University of Lugano, Lugano, Switzerland
{paolierm, bonesani}@alari.ch

[2] 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 tasks of applications such as Network Intrusion Detection Systems and DNA Sequencing Matching. Even though software solutions are available, they do not often satisfy the performance requirements, therefore specialized hardware designs can represent the right choice. This paper presents a novel hardware architecture for efficient regular expression matching: *ReCPU*.
This special-purpose processor is able to deal with the common regular expression semantics by treating the regular expressions as a programming language. With the parallelism exploited by the proposed solution a throughput of more than one character comparison per clock cycle (maximum performance of current state of the art solutions) is achieved and just $O(n)$ memory locations (where n is the length of the regular expression) are required.
In this paper we are going to expose our complete framework for efficient regular expression matching, both in its architecture and compiler. We present an evaluation of area, time and performance by synthesizing and simulating the configurable VHDL description of the proposed solution. Furthermore, we performed a design space exploration to find the optimal architecture configuration given some initial constraints. We present these results by explaining the idea behind the adopted cost-function.

1 Introduction

1.1 State of the Art

Searching for a set of strings that match a given pattern in a large input text - i.e. pattern matching - is a well known computation intensive task present in several application fields. Nowadays, there is an increasing demand for high performance pattern matching. In network security and QoS applications [1][2][3][4][5] it is required to detect multiple packets which payload matches a predefined set of patterns. In Network Intrusion Detection Systems (NIDS) regular expressions are

used to identify network attacks by predefined patterns. Software solutions are not feasible to perform this task without a sensible reduction of the throughput, therefore dedicated hardware architecture can overcome this (e.g. as described in [1]). Bioinformatics - as in case of the Human Genome project - requires DNA sequence matching [6][7]: searching DNA patterns among millions of sequences is a very computationally expensive task. Different alternative solutions to speedup software approaches have been proposed (e.g. like in [7] where the DNA sequences are compressed and a new research-algorithm is described).

For these application domains it is reasonable to move towards a full hardware implementation, overcoming the performance achievable with any software solution. Several research groups have been studying hardware architectures for regular expression matching. They are mostly based on Non-deterministic Finite Automaton (NFA) as described in [8] and [9]. In [5] a software that translates a RE into a circuit description has been developed. A Non-deterministic Finite Automaton is used to dynamically create efficient circuits for the pattern matching task.

FPGAs solutions have been presented: in the parallel implementation described in [4] multiple comparators are used to increase the throughput for parallel matching of multiple patterns. In [8] another FPGA implementation is proposed, it requires $O(n^2)$ memory space and processes one text character in $O(1)$ time (one clock cycle), it is based on an 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, therefore the overall execution time is not constant: it can be linear in best cases and exponential in worst ones. That is not the case for the solution proposed in this paper: regular expressions are stored using $O(n)$ memory locations. Furthermore, it does not require any additional time to start the regular expressions matching. In [9] an architecture that allows extracting and sharing common sub-regular expressions, with the goal of reducing the area of the circuit, is presented. In [6] a DNA sequence matching processor using FPGA and Java interface is addressed. 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. The work proposed in [3] focuses on pattern matching engines implemented with reconfigurable hardware. The implementation is based on Non-deterministic Finite Automaton and it includes a tool for automatic generation of the VHDL description.

These approaches require the re-generation of the HDL description, whenever a new regular expression needs to be executed. Each description is strictly dependent on the given pattern. The time needed for the re-generation increases the overall execution time and reduces the performance.

1.2 An Overview of ReCPU

This paper presents, to the best of our knowledge, a novel approach to solve the pattern matching problem. Regular expressions are considered the programming

language of a dedicated CPU. We do not build either Deterministic nor Non-deterministic Finite Automaton of the given regular expression: so we have the advantage that any modification on the regular expression does not require any change on the HDL description. This way any additional setup time is avoided and a considerable overall speed-up is achieved.

ReCPU - the proposed architecture - is a dedicated processor able to fetch a regular expression from the instruction memory and to perform the pattern matching with the text stored in the data memory. The architecture is optimized to execute comparisons in parallel by means of several parallel units and a two-stage pipeline. This architecture has several advantages: on average it compares more than one character per clock cycle and it requires linear memory occupation (i.e. for a given regular expression 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 any modification of the underlying hardware. Since it is based on a CPU-like approach a *compiler* is necessary to obtain the machine executable code from a given regular expression (i.e. a low-level operational description starting from a high-level representation). This guarantees much more flexibility than the other solutions described in Sect. 1.1.

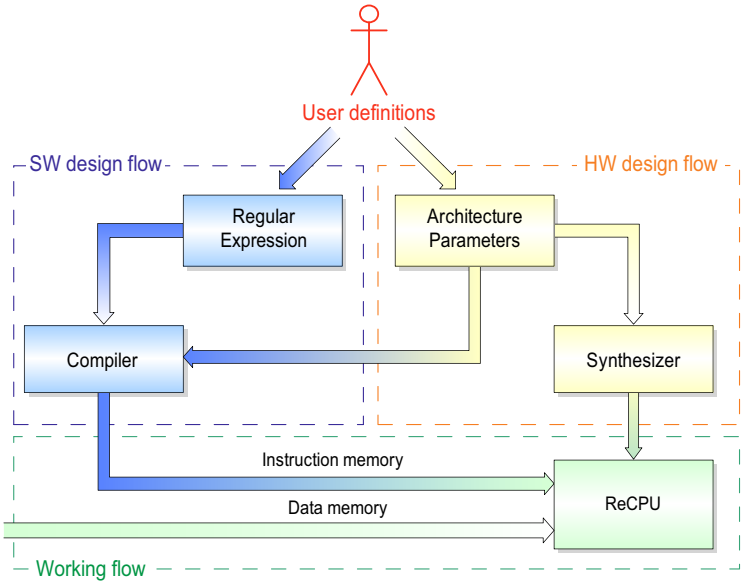

Fig. 1. ReCPU framework flows.

The ReCPU framework (i.e. CPU and compiler) has been inspired from the VLIW design style [10]. This has several advantages: it is easily possible to configure the design to achieve a particular trade-off between performance, area,

power, etc. Moreover since some architectural parameters are exposed to the compiler, it can automatically compile the portable high-level regular expression description to a set of instructions targeted for a custom design.

In Fig. 1 the complete working flow of the proposed framework is shown: the user defines the regular expression and the architectural parameters specifying the number of parallel units. These information are used to synthesize ReCPU on the hardware design flow and to compile the regular expression following the software design flow. ReCPU works on the content of instructions and data memory as it is visible in the bottom part of Fig. 1: the former is automatically generated by the compiler, while the latter is specified by the user.

1.3 Organization of the Paper

This paper is organized as follows: in Sect. 1.2 the general concepts of ReCPU design are described. A brief overview of regular expressions focusing first on a formal definition and then on the semantics that has been implemented in hardware, is addressed in Sect. 2.1. The idea behind considering regular expressions as a programming language is fully covered in Sect. 2.2, by means of some examples.

Section 3 provides a top-down detailed description of the hardware architecture: the *Data Path* in Sect. 3.1 and the *Control Unit* in 3.2.

Results of synthesis on ASIC technology are discussed in terms of critical path, maximum working frequency and area in Sect. 4.1, a comparison of the performance with other solutions is also proposed. In Sect. 4.2 a *Design Space Exploration* (DSE) for FPGA synthesis is provided. A possible cost function is defined and applied to the DSE.

Conclusions and future works are addressed in Sect. 5.

2 Proposed Approach

2.1 Regular Expressions Overview

Formal Definition. A *regular expression* [11] (RE), also known as *pattern*, is an expression that describes the elements of a set of strings. The theoretical concept of regular expression was introduced by Stephen C. Kleene in the 1950s as a model to describe and classify formal languages.

Nowadays, REs are used to perform searches on text data and are commonly present in programming languages, text-editors and word processors for text editing. In this section we propose a brief review of the basic concepts of formal languages to be able to expose formally the concept of regular expression. This helps to understand the different features of ReCPU that we are going to describe in the next sections.

Given a finite alphabet Σ of elementary symbols, we define a *string* as an ordered combination of some elements of Σ. A particular string, belonging to any Σ, is the *empty string*, containing no elements and indicated with ϵ.

Let us define Σ^* as the set of all possible strings generated with the elements of Σ. A language L over Σ is a set of strings generated by the alphabet, thus it is a subset of Σ^*. In other words

$$L(\Sigma) \subset \Sigma^*$$

The simplest language L contains only strings composed by a single element of the alphabet and the empty string. It can be used to generate other languages, called *regular languages*, by applying three basic operators

- concatenation, denoting the set $\{\{a, b\} \,|a \in L_1 \wedge b \in L_2\}$;
- union (or alternation), denoting the set $\{L_1 \cup L_2\}$;
- star (or closure), denoting the set that can be made by concatenating zero or more strings of L.

Using the strings of the language and the operators, it is possible to write formulas representing regular languages (i.e. a new set of strings derived from the regular language). This formula is known as *regular expression*. A language can be titled as *regular* only if it exists a regular expression able to describe the whole set of strings composing it.

We can define formally regular expressions as follows [12]:

Definition 1. *A regular expression over the alphabet Σ is defined as:*

1. *$\emptyset$ is a regular expression corresponding to the empty language $\emptyset$.*
2. *ϵ is a regular expression corresponding to the language ϵ.*
3. *For each symbol $a \in \Sigma$, a is a regular expression corresponding to language a.*
4. *For any regular expression R and S over Σ, corresponding to the languages L_R and L_S respectively, each of the following is a regular expression corresponding to the indicated language:*
   (a) concatenation: $(RS) \Leftrightarrow L_R L_S$;
   (b) union: $(R|S) \Leftrightarrow L_R \cup L_S$;
   (c) star: $R^* \Leftrightarrow L_R^*$.
5. *Only the formulas produced applying rules 1-4 are regular expressions over Σ.*

Given an arbitrary set of strings T and a regular expression R, the pattern matching problem can be defined as follows:

Definition 2. *To find the elements of T, if there are any, which are also elements of the regular language described by R.*

The Syntax. In the IEEE POSIX 1003.2 document, a standard syntax for REs has been proposed. Even though it is a bit different from the formal definition, it recalls the same concepts. In an RE single characters are considered regular expressions that match themselves and additional operators are defined. Let us consider two REs: a and b, the operators that have been implemented in our architecture are:

- $a \cdot b$: 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[3];
- $a+$: matches all strings composed by one or more occurrences of a;
- (a): parenthesis 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: $(ab)*$).

2.2 Regular Expressions as a Programming Language

The novel idea we propose is to translate a given RE into a set of low-level instructions (i.e. *machine code*) part of a program stored in the instruction memory and executed by ReCPU. This approach is the same used in general purpose processors: a program is coded using a high-level language - that is more readable by programmers - and then compiled, optimized and linked to produce an efficient low-level set of instructions executed by the microprocessor. An example of this, is to code a program using C, then build it using *gcc* to obtain the compiled binary program that is executed by the CPU.

In our case the high-level representation is the RE defined according to the standard syntax described in [11] or in [13]. The compilation flow is inspired by the VLIW style [10], because some architectural parameters are exposed to the compiler, that is able to exploit an high level of parallelism issuing the instructions to different parallel units (i.e. the *clusters)*. Similarly, the *Regular Expression compiler* (REc) is aware of the number and the structure of configurable units in the ReCPU architecture and based on this it splits the RE into different low-level instructions, issuing as many character comparisons as the number of parallel comparators available in the architecture.

Given a regular expression, REc[4] generates the sequence of instructions that are executed by the core of the architecture on the text stored in the data memory. REc provides in output two binary images: one for the instruction memory and the other one for the data memory, with the input text adapted to the configuration of the architecture. The compiler does not need to perform many optimizations due to the simplicity of the *RE programming language*. However, some controls are performed to detect syntactical mistakes and possible problems of stack-overflow. Given the stack-size (see Sect. 3 for more details) REc computes the maximum level of nested parenthesis allowed and determines whether the architecture can execute the specified RE or not. A RE is completely matched whenever a NOP instruction is fetched from the instruction memory. If any instruction fails during the execution, the RE is considered completely failed and it is restarted. The binary code of a low-level instruction produced by the compiler is composed by an *opcode* and a *reference text* as shown in Fig. 2. The *opcode* is divided in three different parts:

[3] Please notice that this operator is different from the formal *star* operator previously defined.

[4] REc is the Regular Expression compiler written in Python

– the most significant bit (MSB) used to indicate an open parenthesis;
– the next 2-bits for the internal operand used within the reference;
– the last bits for the external operand for describing loops and close parenthesis.

The complete list of the opcodes is shown[5] in Table 1.

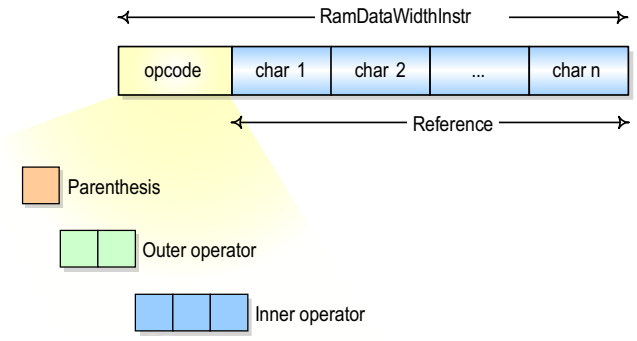

Fig. 2. Instruction Structure.

Table 1. Bitwise representation of the opcodes.

Opcode	Associated Operator
0 00 000	nop
1 -- ---	(
0 01 ---	and
0 10 ---	or
0 -- 001	)*
0 -- 010	)+
0 -- 011	)|
0 -- 1--	)

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). 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 to the minimum number of iterations.

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

Another characteristic of complex REs that can be perfectly managed considering REs as a programming language is the use of nested parenthesis (e.g. $(((ab) * (c|d))|(abc))$). This can be handled with the *function call paradigm* of common programming languages, so that we can deal with it as in the majority of processors. We consider an open parenthesis as a *call* instruction and a closed one as a *return*. Whenever an open parenthesis is encountered, the current context is pushed into an entry of a stack data structure and the execution continues normally. Whenever a close parenthesis is found, a pop operation is performed on the stack and the overall validity of the RE is checked by combining the outer operator of the the current instruction, the current context and the previous one popped from the stack. The context is composed by the internal registers of the *Control Unit* that contain the memory location and the partial matching result (see Sect. 3.2 for further details). This way, our architecture can tackle very complex nested REs using a well and widely known approach.

A simple example of translating a RE into a sequence of instructions is listed in Table 2 (where it has been hypothesized to have a ReCPU that compares 4 characters per comparison unit - see Sect. 3 for further details). The given RE is *(ABCD)|(aacde)*, the open parenthesis are translated into *call*s, while the closed parenthesis are translated into *return*s. The original RE is split into several sub-expressions. This mechanism allows us to exploit the maximum possible capacity of the internal parallel comparators. The overall RE result is computed by combining the partial results of each sub-expression. If during the execution the RE does not match, the program is restarted from the first instruction with a different starting address in the data memory. Otherwise, the execution continues until the NOP instruction is fetched. At this point the RE is considered completed.

Table 2. Translation of the RE=(ABCD)|(aacde), into ReCPU instructions using 4 cluster units.

SubRE	Translated Instructions
`(`	call
`ABCD`	compare text with "ABCD"
`)|`	return: process OR
`(`	call
`aacd`	compare text with "aacd"
`e)`	compare text with "e" and return, overall evaluation
`NOP`	end of RE

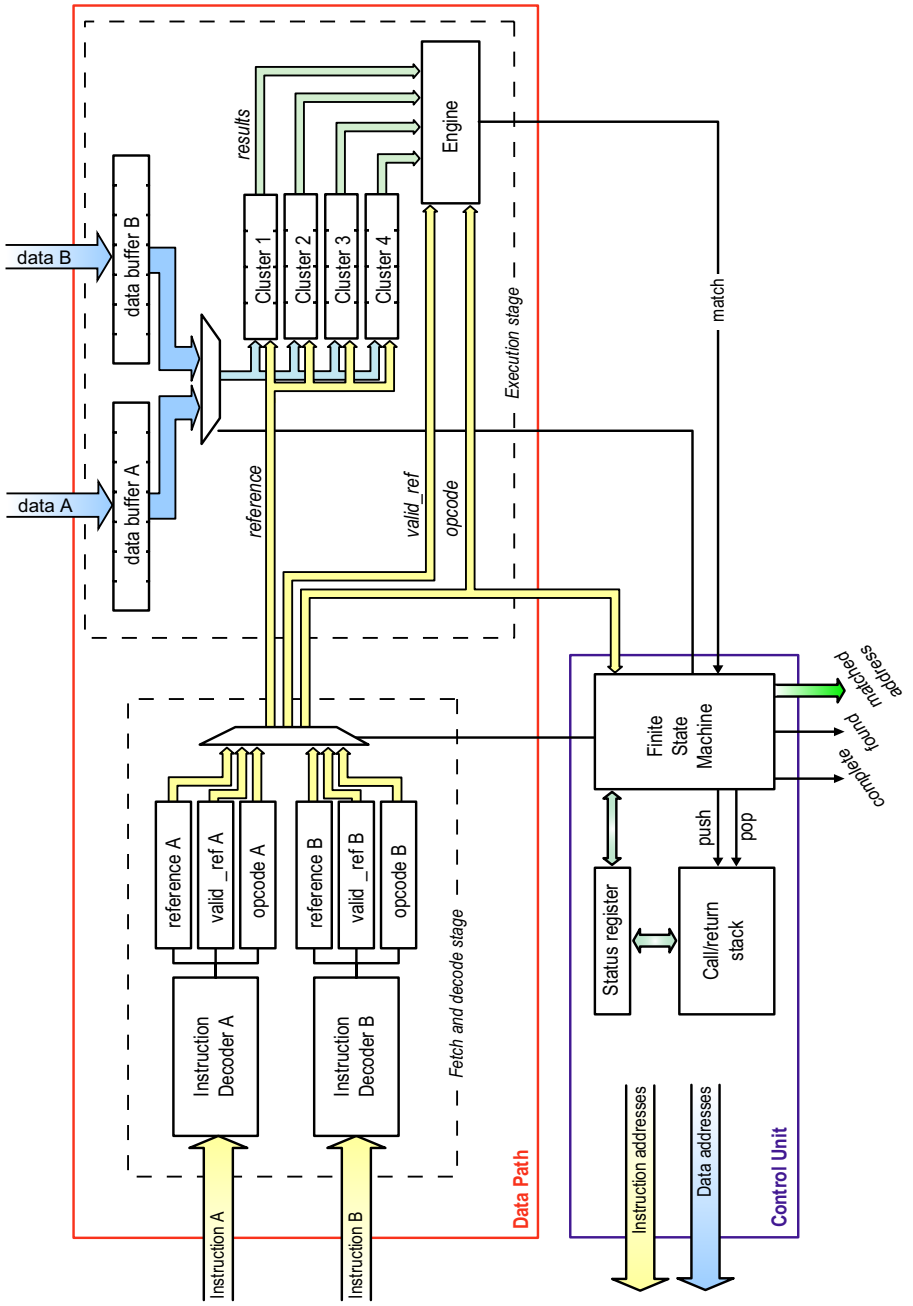

Fig. 3. Block diagram of ReCPU with 4 Clusters, each of those has a ClusterWidth of 4. The main blocks are: Control Path and Data Path (composed by a Pipeline with Fetch/Decode and Execution stages).

3 Architecture Description

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 are dual port to allow parallel accesses to the parallel buffers described in Sect. 3.1. As shown in Fig. 3, the structure of ReCPU is divided into two parts:

- The *Data Path* is in charge of decoding the instructions, executing the comparisons and producing the partial matching result.
- The *Control Unit* selects the next instruction to be executed, collects the partial matching results to check the correctness of the RE and is in charge of executing complex instructions such as loops and parenthesis.

One of the main features of ReCPU is the capability to process more than one character comparison per clock cycle. In this design we applied some well known computer architecture techniques - such as pipelining and prefetching - to provide a higher throughput. We achieved this goal by incrementing the level of data and instruction parallelism and by limiting the number of stall conditions, which are the largest waste of computation time during the execution.

The architecture is adaptable by the designer who can specify the number of parallel units, their internal structure as well as the width of the buffers and the memory ports. To adapt the architecture to the requirements is necessary to trade-off 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 Sect. 4).

This section overviews the internal structure of ReCPU - shown in the block diagram of Fig. 3 - focusing on the microarchitectural implementation. A detailed description of the two main blocks: the *Data Path* and the *Control Unit*, is provided in the sections 3.1 and 3.2.

3.1 Data Path

We applied some techniques from processor architecture field to increase the parallelism of ReCPU: *pipelining*, data and instructions *prefetching*, and use of multiple memory ports. The pipeline is composed by two stages: *Fetch/Decode* and *Execute*. The *Control Unit*, as explained in Sect. 3.2, takes one cycle to fill the pipeline and then it starts taking advantage of the prefetch mechanism without any further loss of cycles. Moreover we introduced duplicated buffers in each stage to avoid stalls. This solution is advantageous because the replicated blocks and the corresponding control logic are not so complex: the increase in terms of area is acceptable, and no overhead in terms of critical path. This way, we have a reduction of the execution latency with a consequent performance improvement.

Due to the regular flow of the instructions a good prediction technique with duplicated instruction fetching structures is able to avoid stalls. In the *Fetch/Decode* stage, 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 the matching process fails (i.e. prefetching is useless) the content of the second buffer - the backup of the first one - can be used without the need of 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 Fig. 3, the multiplexer is controlled by the *Control Unit*. The decoder logic 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 (see Fig. 3) 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 *Fetch/Decode* stage this technique reduces the latency of the access to the memory avoiding the need of a stall if a jump in the data memory is required. 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 restarts the search from the address where the first match occurred.

The core of ReCPU is based on sets of parallel bitwise comparators grouped in units called *Clusters*, which are shown in Fig. 4. Each comparator compares an input text character with a different one coming from the reference of the instruction (see Fig. 2). The number of elements of a *Cluster* is indicated as *ClusterWidth* and represents the number of characters that can be compared every clock cycle whenever a sub-RE is matching. This figure influences the throughput whenever a part of the pattern starts matching the input text. The *Execute* stage is composed by several *Clusters* - the total number is indicated as *NCluster* - used to compare a sub-RE. Each *Cluster* is shifted one character from the previous cluster in order to cover a wider set of data in a single clock cycle. This influences the throughput whenever the pattern is not matching. The results of each comparator *Cluster* are collected and evaluated by the block called *Engine*. It produces a match/not-match signal to the *Control Unit*.

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. Each parameter is propagated through the modules using the VHDL *generics* technique. Moreover, we defined some packages to define some constants, types and other values used in entities and architectures.

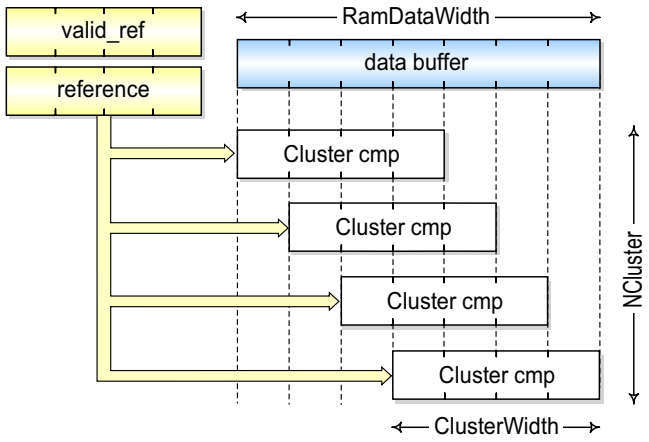

Fig. 4. Detail of comparator clusters.

3.2 Control Unit

In Sect. 2, we defined an RE as a sequence of instructions that represent a set of conditions to be satisfied. If all the instructions of an RE are matched, then the RE itself is matched. Essentially, the ReCPU *Data Path* fetches an 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, because it does not know the next address to be loaded.

To manage the execution of the RE we designed a *Control Unit* block based on some specific hardware components. The core of the *Control Unit* is the *Finite State Machine* (FSM) shown in Fig. 5. The execution of an RE requires two input addresses: the RE and the text start addresses. The FSM is designed in such a way that after the preload of the pipeline (`FD` state), two different 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.

While the text is not matching, the same instruction address is fetched and the data address advances exploiting the comparisons with the clusters of the *Data Path*. If no match is detected the data memory address is incremented by the number of clusters. This way, multiple characters are compared every single clock cycle leading to a throughput clearly greater than one character per clock cycle. Further details are presented in Sect. 4.

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 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 of the RE fails the whole process has to be restarted from the point where the RE started to match.

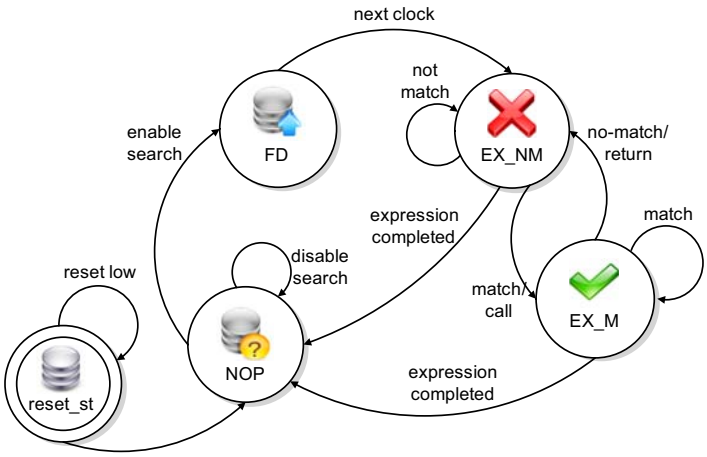

Fig. 5. Finite state machine of the *Control Path*.

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 overall result is given as output. The ReCPU returns a signal indicating the matching of the RE and the memory location of the first character matching the string. We conceived an operation mode able to find more than one pattern in a text: each time a RE is matched, ReCPU stops until an input signal requests another RE matching. In such case the RE is reloaded and the sequence of operations is restarted.

A particular case is represented by loops (i.e. + or * operators). We exploit these operators with a `call` and `return` paradigm. When an open parenthesis is detected a call is performed: the *Control Unit* saves the content of the status register (i.e. the actual matching value, the current program counter and the current internal operator) in the stack module drawn in Fig. 3. The RE is then executed normally until a return instruction 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 size of the stack and computing the maximum level of nested parenthesis it is able to determine whether the architecture can execute the RE or not.

4 Experimental Results

4.1 Analysis of Synthesis and Simulation Results

ReCPU has been synthesized using Synopsys Design Compiler[6] on the STMicroelectronics HCMOS8 ASIC technology library featuring 0.18μm silicon process. The proposed architecture has been synthesized setting *NCluster* and *ClusterWidth* equal to 4. The synthesis results are presented in Table 3:

Table 3. Synthesis results for ReCPU architecture with NCluster and ClusterWidth set to 4.

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

The papers describing the hardware solutions covered in Sect. 1 show a maximum clock frequency between 100MHz and 300MHz. The results 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 [8] and [9]).

Let us 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

$$T_{cnm} = \frac{T_{cp}}{NCluster + ClusterWidth - 1} \tag{1}$$

where the T_{cnm}, expressed in $ns/char$, depends on the number of clusters, the width of the cluster 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:

$$T_{onm} = \frac{T_{cp}}{NCluster} \ . \tag{2}$$

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)

$$T_m = \frac{T_{cp}}{ClusterWidth} \ . \tag{3}$$

For each different scenarios, using the time per character computed with the formulas (1), (2) and (3) it possible to compute the corresponding bit-rate to evaluate the maximum performance. The bit-rate B_x represents the number of bits[7] processed in one second and can be computed as follows:

[6] www.synopsys.com

[7] It is computed considering that 1 character = 8 bits.

$$B_x = \frac{1}{T_x} \cdot 8 \cdot 10^9 \tag{4}$$

where T_x is any of the quantities (1), (2) and (3).

The numerical results for the implementation we have synthesized are shown in Table 4.

Table 4. Time requested 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 4 represent the maximum achievable throughput with different scenarios. Whenever there is a function call (i.e. nested parenthesis) one additional clock cycle of latency is required. The throughput of the proposed architecture depends on the RE as well as on the input text so it is not possible to compute a fixed throughput but just to provide the maximum performance achievable in different cases.

In our experiments we compared ReCPU with the popular software *grep*[8] 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 at 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 5.

Table 5. 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

We noticed that if loop operators are not present our solution performs equal either with more than one instruction and OR operators 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 solutions. In case of loop operators it is possible to notice a slow-down in the performance but still achieving a *speedup* of more than 60 times.

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

To prove the performance improvements of our approach respect to the other published solutions, we compare the bit-rates described in the Table 6. It was not possible to compare the bit-rate for [6], [9] because this quantity was not published in the papers.

In Table 6 and in Fig. 6, the bit-rate range for different solutions is shown. We compared it with the one of ReCPU computing a speedup factor that underlines the goodness 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.

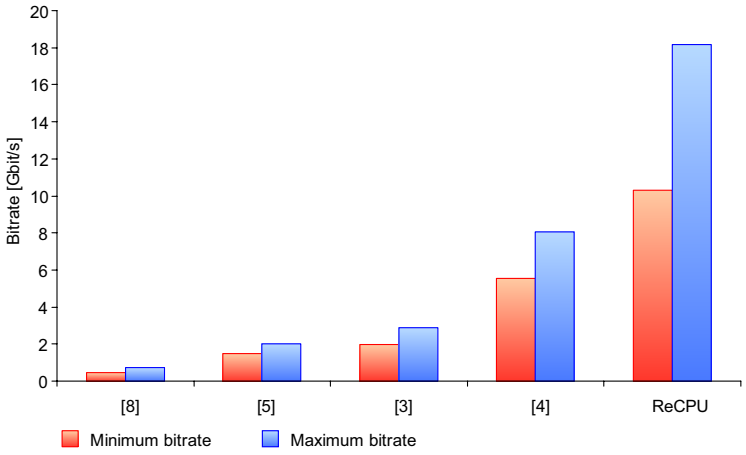

Fig. 6. Comparison of the bitrates of ReCPU and the state of the art solutions.

Table 6. Bit-Rate comparison between literature solutions and ReCPU.

Solution published in	bit-rate Gbit/s	ReCPU Gbit/s	Speedup factor (x)
[3]	(2.0, 2.9)	(10.19, 18.18)	(5.09, 6.26)
[5]	(1.53, 2.0)	(10.19, 18.18)	(6.66, 9.09)
[4]	(5.47, 8.06)	(10.19, 18.18)	(1.82, 2.25)
[8]	(0.45, 0.71)	(10.19, 18.18)	(22, 25)

4.2 Design Space Exploration

In this section we present a Design Space Exploration used to find the optimal ReCPU architecture configuration synthesized on a Xilinx Virtex-II FPGA[9]. Taking advantage of the fully configurable VHDL description, we modified the structure altering the number of parallel comparator clusters - i.e. `NCluster` (NC) - and the number of bitwise comparator units - i.e. `ClusterWidth` (CW). We analyzed how area and performance scale.

We performed the Design Space Exploration with `NCluster` in the range $\{2, 4, 8, 16, 32, 64\}$ and `ClusterWidth` in $\{4, 8\}$. Also the width of the memory ports (i.e. RWD: `RamWidthData`, RWI: `RamWidthInstruction`) must be adapted according to the following rules:

$$RWI = CW$$

$$RWD = CW + NC$$

Increasing the number of `NCluster`, more characters are compared simultaneously, and so ReCPU results to be faster whenever the pattern is not matching the input text. Nevertheless, due to the higher hardware complexity, the *Critical Path* raises up and thereby the maximum possible clock frequency decreases. On the other side, a larger `ClusterWidth` corresponds to much better performance whenever the input string starts matching, since a wider sub-RE is processed in a single clock cycle. The results of the synthesis are shown in Table 7.

Table 7. Results of the synthesis of ReCPU with different parameters on Xilinx Virtex-II FPGA.

NC	CW	RWD	RWI	Critical Path ns	Max Freq. MHz	Norm. Area	T_{cnm} ns	T_{onm} ns	T_m ns	Cost	Norm. Cost
2	4	6	4	8.938	111.88	0.1334	1.8	4.4	2.2	2.68	1.00
4	4	8	4	9.722	102.86	0.1619	1.4	2.4	2.4	2.17	0.97
8	4	12	4	10.078	99.23	0.2086	0.9	1.3	2.5	1.8	0.81
16	4	20	4	11.157	89.63	0.3128	0.6	0.7	2.8	1.72	0.77
32	4	36	4	11.583	86.33	0.4944	0.3	0.4	2.9	1.62	0.73
64	4	68	4	12.974	77.08	1	0.2	0.2	3.2	1.72	0.77
2	8	10	8	8.938	111.88	0.1334	1.0	4.5	1.1	1.92	0.86
4	8	12	8	9.722	102.87	0.1619	0.9	2.4	1.2	1.44	0.65
8	8	16	8	10.078	99.23	0.2086	0.7	1.3	1.3	1.11	0.50
16	8	24	8	11.157	89.63	0.3128	0.5	0.7	1.4	0.99	0.45
32	8	40	8	11.583	86.33	0.4944	0.3	0.4	1.4	0.89	0.40
64	8	72	8	12.974	77.08	1	0.2	0.2	1.6	0.91	0.41

In the Design Space Exploration to evaluate the different configurations that have been synthesized and listed in Table 7, we defined a *cost function* that

[9] A Virtex-II pro, technology xc2vp30-fg676-7.

takes into account the previously described scenarios considering the area and the performance. It is defined as follows:

$$costf = p_1 \cdot T_{cnm} + p_2 \cdot T_{onm} + p_3 \cdot T_m \ . \tag{5}$$

The function $costf(\cdot)$ evaluates the different performance indexes (see Sect. 4.1) with a corresponding probability. To better analyze the overall implementation it is necessary to distinguish among different cases. We have not performed a statistical analysis on the utilization of the operators as well as on the probabilities of having or not a matching. This would be necessary to compute an average performance index, but it is strictly dependent on the input search text.

Let us consider an input text and an RE such that the probability of having an *and* operator in the current instruction and the probability of having an *or* operator in the current instruction are the same (i.e. $p_{and} = p_{or} = 0.5$). Among these cases there is respectively the probability $p_m = 0.25$ of matching the pattern and 0.25 of not matching. We actually consider all the cases equiprobable. The $costf(\cdot)$ is the resulting average time per character based on the previous probabilities. Let us define

- p_1 as the probability of having an *and* operator with a not matching pattern ($p_1 = 0.25$);
- p_2 as the probability of having an *or* operator with a not matching pattern ($p_2 = 0.25$);
- p_3 as the probability of having a matching with any operator (0.5).

The Design Space Exploration optimizes the $costf(\cdot)$ given these probabilities for the inputs (i.e. search text and RE). We choose to optimize the cost function with respect to the area of the design. The normalized values computed in Table 7 are plotted in Fig. 7, where the *Pareto* optimal points have been highlighted. Each point in the graph is a synthesis of ReCPU with a different set of values for the generics VHDL parameters. It is easily possible to identify which points belong to the *Pareto* front, and which others are the dominated ones. The best configuration out-coming from this analysis is listed in Table 8.

Table 8. The best configuration out-coming from the analysis of the *Pareto* points generated by the Design Space Exploration.

NC	CW	RWD	RWI	Critical Path ns	Max Freq. MHz	Norm. Area	T_{cnm} ns	T_{onm} ns	T_m ns	Cost	Norm. Cost
8	8	16	8	10.078	99.23	0.2086	0.7	1.3	1.3	1.11	0.50

5 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

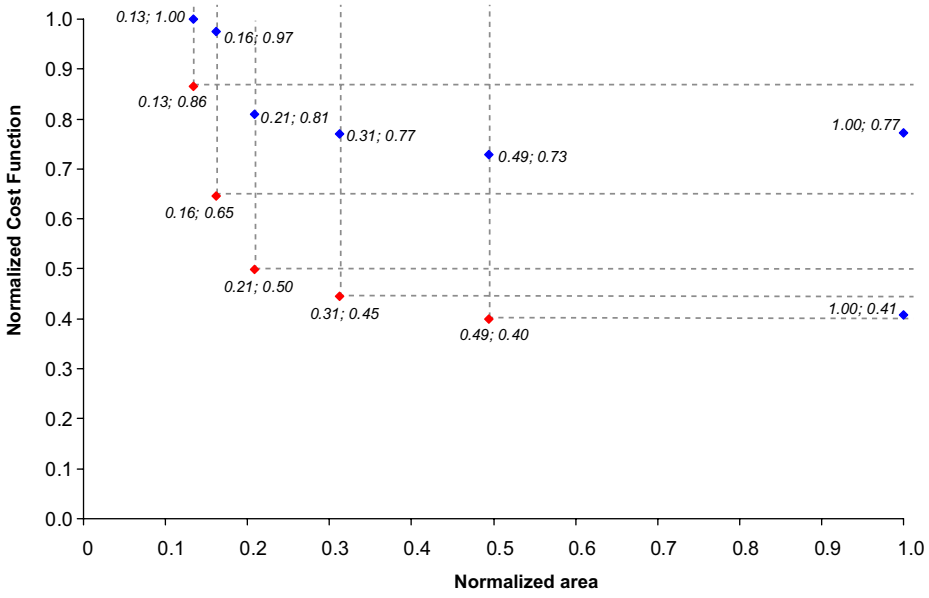

Fig. 7. This plot shows the normalized values of area and cost function of Table 7. The optimal points belonging to the *Pareto front* are colored in red while the non-optimal are in blue.

DNA sequence matching is one of the main applications. To achieve higher performance it is necessary to use hardware solutions for pattern matching tasks. In this paper we presented a novel architecture for hardware regular expression matching.

Our contribution involves a completely different approach of dealing with the regular expressions. REs are considered the programming language of 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 from the VLIW design style.

The current state of the art solutions guarantee a fixed performance of one character per clock cycle. Our goal was to figure out a way of extract additional 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.

We presented the results of the synthesis on ASIC technology to compare the performance with the other state of the art solutions. Moreover, we exploited the configurable VHDL design of ReCPU to perform a Design Space Exploration, proposing a possible cost function based on the probabilities of matching or not a pattern with different RE operators. We provided the results of the exploration

by optimizing the cost function respect to the area requested to synthesize our design on an FPGA.

Future works are focused on the definition of a reconfigurable version of the proposed architecture based on FPGA-devices. This way, we could 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 optimizations in the compiler.

Another possible development is to use ReCPU in a parallel multi-core environment. This way it could be possible to create a cluster of pattern matching processors working together and increasing the throughput. The advantages include the possibility of having a ReCPU cluster in a single System-On-Chip, achieving considerable high performance at a contained cost due to the density of the actual silicon technology that offer large areas with contained costs.

References

1. Yadav, M., Venkatachaliah, A., Franzon, P.: Hardware architecture of a parallel pattern matching engine. In: Proc. ISCAS. (2007)
2. Liu, R.T., Huang, N.F., Kao, C.N., Chen, C.H., Chou, C.C.: A fast pattern-match engine for network processor-based network intrusion detection system. In: Proc. ITCC. (2004)
3. Bispo, J., Sourdis, I., Cardoso, J., Vassiliadis, S.: Regular expression matching for reconfigurable packet inspection. In: IEEE International Conference on Field Programmable Technology (FPT). (2006) 119–126
4. Sourdis, I., Pnevmatikatos, D.: Fast, large-scale string match for a 10Gbps FPGA-based network intrusion. In: International Conference on Field Programmable Logic and Applications, Lisbon, Portugal (2003)
5. Cho, Y., Mangione-Smith, W.: 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) 234–239
6. Brown, B.O., Yin, M.L., Cheng, Y.: DNA sequence matching processor using FPGA and JAVA interface. In: Annual International Conference of the IEEE EMBS. (2004)
7. Chen, L., Lu, S., Ram, J.: Compressed pattern matching in DNA sequences. In: Proc. CSB. (2004)
8. Sidhu, R., Prasanna, V.: Fast regular expression matching using FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01). (2001)
9. Lin, C.H., Huang, C.T., Jiang, C.P., Chang, S.C.: 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) 12–17
10. Fisher, J.A., Faraboschi, P., Young, C.: Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann (2004)
11. Friedl, J.: Mastering Regular Expressions. 3 edn. OReilly Media (2006)
12. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages and Computation. 2 edn. Pearson Education (2001)
13. GNU USA: Grep Manual. (2002)