An Energy-Efficient Domain-Specific Architecture for Regular Expressions

Davide Conficconi, *Graduate Student Member, IEEE,* Emanuele Del Sozzo, *Member, IEEE,* Filippo Carloni, *Graduate Student Member, IEEE,* Alessandro Comodi, Alberto Scolari, Marco Domenico Santambrogio, *Senior Member, IEEE*

Abstract—Regular Expressions (REs) are a computational kernel widely used for finding patterns in data in compute-intensive tasks such as genomic markers research, signature-based detection, and database query. Although flexible on the set of searched REs, software-based solutions cannot fulfill latency or throughput requirements to analyze massive data volumes at a given power budget. For this reason, many approaches exploit hardware accelerators as an offloading engine for REs matching. Indeed, various solutions rely on FPGA reconfigurability to embed automata into the reconfigurable fabric. However, this approach leads to time-consuming updates of the REs to search. This work exploits REs as sequences of basic instructions and builds a Domain-Specific Architecture (DSA), called TiReX, for RE matching on FPGAs. Our approach enables the user to change the desired RE at run-time, providing software programmability, flexibility, and specialized hardware mechanisms. Our DSA delivers performance in line with other state-of-the-art hardware approaches, while providing remarkable flexibility and we underline the importance of energy efficiency for these computations. We compared with multiple state-of-the-art software obtaining remarkable performance while achieving noticeable results with a better energy efficiency that ranges from 3× to 490× with our multi-core.

Index Terms—Regular Expressions; Domain-Specific Architecture; FPGA; Multi-core; Reconfigurable ISA

1 Introduction

Regular Expressions (REs) are widely used in several fields, from genome analysis [1] to text analytics [2] and Intrusion Detection Systems (IDSs) [3] for fast and efficient pattern matching tasks [4], [5]. The requirements of these applications and the high amounts of data to analyze make the performance of RE matching solutions crucial, keeping fostering research on this problem. For example, IDSs may require near-real-time analysis of packets at the network rate, thus requiring specialized hardware solutions [3], [6]. At the same time, the growth of fields such as "Personalized Medicine" [7], [8] highly relies on REs to identify genomic patterns in data [9] and requires fast analysis abilities to push forward research. Many researchers address these challenges by efficiently representing the automata that accept the language defined by the RE [10], [11], [12]. In contrast, others exploit specialized hardware (e.g., ASICs or FPGAs [2], [13]) to build effective domain-specific solutions [14]. Indeed, reconfigurable devices demonstrate computing and energy efficiency advantages over general-purpose processors [15], as well as higher adaptability than ASICs. For this reason, several methodologies embed automata into the reconfigurable fabric of an FPGA. In this way, they leverage the reconfigurability to change the RE(s) to find [2], [5], [16], [17], [18].

Generally, the reconfiguration time of an FPGA is in the order of milliseconds to seconds, while generating a new bitstream requires from one to several hours, making real-time adaptation unfeasible [18]. Thus, the reconfigurable fabric embedding requires a database of ready-to-use bitstreams or bitstream regeneration if the pattern is new. On the one hand, sacrificing the run-time adaptability with automaton embedding achieves remarkable performance [11]. On the other hand, having this flexibility could lead to sub-optimal performance [19]. However, easily changing the searching pattern is an essential feature in many application fields [18], where wasting a few microseconds could lead to unsustainable performance degradation [20]. IDS on datacenters [21], malware detection [22], genome markers search [1], and database queries [23] are all application scenarios that require to update their patterns, or REs continuously, to keep pace with the newest advancements, otherwise, they would result in incomplete computations.

This work proposes a software-programmable Domain-Specific Architecture (DSA) for reconfigurable systems tailored to the REs domain. Our DSA, called TiReX [24], overcomes the reconfigurability embedding issue exploiting the idea of using REs as a programming language for our custom Instruction Set Architecture (ISA) [24], [25], [26]. This domain narrowing leads to an optimized architecture for multi-character analysis easily adaptable at run-time to the REs to analyze. We devised a multi-core architecture that operates in a multi-pattern single-stream of data or single-pattern multi-stream of data using software control. Moreover, our DSA is easily deployable on different FPGA-

• *D. Conficconi, E. Del Sozzo, F. Carloni, A. Comodi, A. Scolari, M. D. Santambrogio are with the Novel, Emerging Computing System Technologies Laboratory (NECSTLab), Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, 20133 Milano, Italy. E-mail: {davide.conficconi, emanuele.delsozzo, filippo.carloni, alberto.scolari, marco.santambrogio}@polimi.it, alessandro.comodi@mail.polimi.it*

based platforms depending on the target workload, e.g., embedded or high-performance. The domain specialization of TiReX pushes towards the energy efficiency required to address current computing challenges [14], [27]. In summary, the main contributions are:

- The design of a multi-core DSA for RE matching. We evaluated it on various platforms showing remarkable performance against software solutions with the same adaptability while reaching performance in line with related hardware solutions;
- Two architectural models tailored for latency- and throughput- driven scenarios and their evaluation on state-of-the-art benchmarks, showing the remarkable energy efficiency achieved.

The rest of the article discusses the background and the motivations for REs matching (Section 2) and related work (Section 3). Then we describe our approach (Section 4) and system architecture models (Section 5). We evaluate our approach architectural parameters, multi-core scaling area, and run-time results (Section 6), and finally, we draw the conclusion of this work (Section 7).

2 Background and Motivations

REs are a declarative way of describing sets of character strings used to define Regular Languages [28], with meta-characters for specifying operations. Table 1 presents minimum symbols for RE algebra, some of the most used RE operators in all the typical application scenarios (e.g., text-editors search functionality, software libraries for REs, web search engines). There are advanced operators on top of these symbols, which are nowadays involved in pattern matching but not related to Regular Languages, such as backreference. In this work, we focus on classical REs.

Finite Automaton (FA) is a descriptive way of defining a finite state machine that accepts a particular RE or a class of REs that belong to Regular Languages. FA splits in Deterministic FA (DFA) or Non-deterministic FA (NFA) based on the possibility of being in one or more states simultaneously, i.e., deterministic vs. non-deterministic execution model. Though the NFA's execution model relies on a breadth-first-like approach and provides the best theoretical performance, it is equivalent to a DFA, which usually adopts a depth-first-like approach, on the class of acceptable languages [28].

REs and Automata are general execution engines for a wide variety of applications ranging from simple text searching to random forest execution [1]. Among these application fields, REs find room in computer security scenarios, for example, in *Signature Based Detection* techniques for malware detection, such as Firewall filtering and IDSs; here, tools like *YARA* [22] attempt to identify and classify malware samples via REs. Additionally, these fields require low-latency and high-throughput security countermeasures without sacrificing usability. Previous IDS solutions employ specialized hardware [3] to sustain the increasing networking rate and remove the CPU burden, often renouncing the fundamental ability to run-time change the patterns without synthesizing a new design or using more flexible alternatives like GPUs [6]. A completely different field regards the adoption of REs in Bioinformatics to find common patterns among genomic sequences for various goals, ranging from classification of proteins or generic genome sequences [9] to the study of diseases and their treatment. In these applications, the shared DNA patterns, called genetic markers, are involved in the identification of various types of health issues (like genetic diseases [7] or cancer treatment [8]), which are heavily based on pattern matching techniques, where the analysis of genomic data facilitates the discovery of the best treatment for the patient. Although initial studies showed promising results, further developments of matching solutions for healthcare- and genomic-related scenarios are fundamental to scale to real-world applications. In addition to bioinformatics, REs find room also in the relational databases field, showing their importance on large applications [23] due to the amount of data to process. Indeed, different database-focused research approaches exploit new in-memory architectures combined with heterogeneous architectures [23] showing promising results in terms of throughput, usage of memory bandwidth, and run-time hardware flexibility, meaning cost reduction of the accelerated operations. All these applications can benefit from a pattern-matching engine with a good ratio of performance-per-watt, tailored to the target deployment environment, and that can keep up with the fast evolution of new patterns and an easy run-time adaptation [18].

TABLE 1: Some meta-characters for Regular Expressions.

Meta-Character	*Description*
xy	concatenation of x and y, same as writing $x\&y$
$x\|y$	alternation of x or y
$()$	priority encoding
x^*	repetition of zero or more x
x^+	repetition of one or more x
$x^{\{n,m\}}$	repetition of x from n times to m times
.	any alphanumeric character
$[x-y]$	character ranging from x to y

3 Related Work

Previous researches explore either the algorithm part of RE matching [4] or the efficiency of the execution platform [13]. The RE matching procedure usually executes through an approach based on DFA or NFA and on their state transition table. While DFA suffers from the exponential memory footprint explosion [10], [11], NFA requires high bandwidth to execute in parallel all possible active states [29]. Other solutions mix DFA-NFA characteristics to achieve the best complexity from time and space requirements [30]. An example CPU-based engine is Intel Hyperscan [31], which overcomes main deep-packet inspection limitations with a novel regex decomposition mechanism and a CPU SIMD pattern matching for multi-string divided in a shift-or part and a verification of the false-positive part, at the cost of high preprocessing mechanisms. The state-of-the-art hardware execution engines are traditionally divided into fixed architecture per automaton, or pattern(s) (e.g., [11]), and reconfigurable architectures. We call the former category Streaming-Dataflow Architectures (SDAs), given their adoption of streaming/dataflow patterns for their fixed architecture. The latter category further divides In Memory-based Architectures (IMAs) based on state lookup

(e.g., [13]), or Software-Programmable Architectures (SPAs) (e.g., [25]). Both the IMAs and SPAs share the ability to change the pattern to be found without regenerating the underlying hardware architecture. This Section gives an overview of these approaches and presents some state-of-the-art techniques applied in pattern matching engines, focusing on FPGAs.

DFA-based hardware approaches, though they achieve better time complexity, generally suffer the memory footprint explosion for state transition table representation [18]. For this reason, many approaches focus on compression techniques such as states and transition clustering [11]. BFSM [10] encodes state transitions as rules, and groups similar transitions into a rule, achieving a very short and predictable memory lookup latency on an IBM Power Edge of Network. Differently, Gogte et al. [2] implement a solution in ASIC, based on the Aho-Corasick algorithm, overcoming its high memory requirements by splitting each input character in single bits. Tang et al. [32] propose a flexible real-time update FSM and optimized DFA encoding scheme sacrificing the performance. Moreover, DFA's original execution scheme is limited to single-character analysis. Hence, to tackle this issue, PiDFA [12] parallelizes character processing with a first computation of all the possible input character transitions, which is then merged in a pipelined fashion. Meiners et al. [3] propose a solution based on Ternary Content Addressable Memory (TCAM), which encodes the transitions of the DFA to improve the lookup process. Another body of work leverages hardware parallelism to increase the number of REs concurrently analyzed. Vasiliadis et al. [6] leverage the characteristics of GPUs to match multiple REs in parallel, encoding each RE as a separate DFA. Other solutions embed in the FPGA logic many REs for Network-IDS [33].

NFA-based hardware approaches start with Sidhu et al. [5], that were the first to implement an NFA embedded in the FPGA logic, showing a flexible self-reconfigurable device, paving the way to further solutions directly synthesizing NFA [34] or exploiting dynamic reconfiguration [35]. Trying to overcome the run-time adaptability of approaches based on the fabric embedding, different solutions exploit either a partially reconfigurable solution based on a multi-character NFA [36], or a fully flexible structure with some patterns expression limitations [37]. Other FPGA-centric approaches focus on specific applications, such as database query with fixed parametrizable operation [23] and signature-detection with YARA rules [38], which achieve remarkable performance results while sacrificing the time required for unknown REs at compile time. On a different view, Dulgosh et al. [13] propose an attractive non-Von Neumann reconfigurable architecture, called the Automata Processor, for parallel automata processing for which they released a simulator. With this simulator, many solutions show promising results in applications such as NLP and genomics [1]. Indeed, from the Automata Processor push, Xie et al. propose REAPR [16], a tool that transforms a high-level automaton description to a target RTL representation for FPGAs. Sadredini et al. [17] claim the inadequacy of current automata-based computations to exploit spatial architectures and present an open-source toolkit for automata simulation. Nourian et al. [29] provide a comprehensive analysis of NFA solutions for different platforms (e.g., GPUs, FPGAs, Micron's Automata Processor) with different workloads and a partitioning scheme aimed at large NFA handling, showing that NFA-parallelism is not well suited to GPUs or vector architectures. In contrast, reconfigurable architectures are the most promising ones.

Hybrid approaches mix DFA and NFA to tackle their individual disadvantages. For example, Atasu et al. [19] use NFA to tackle traditional DFA limitations like repeating the matching process for each possible initial character or verifying an unbounded number of possible initial positions. It activates a new DFA for each first matching character, which significantly improves the performance at the cost of not scaling to complex or multiple REs. Others exploit multiple DFA-based engines, called B-FSM [10], for a flexible NFA-like approach in a heterogeneous system.

Different approaches do not rely on the use of FA but focus on achieving an efficient lookup process. For example, Agarwal et al. [39] propose a hash-based encoding scheme for text patterns (not specifically REs) that generates a dictionary matching engine. Instead, Nguyen et al. [40] employ bitmap index structures to encode the strings to match against, achieving multi-character lookups on FPGA.

Other methodologies consider REs as instructions, such as Google's library RE2[1], which is mainly based on Thompson's NFA and detailed by Russ Cox, showing the overwhelming power against a backtracking solution. Furthermore, Cox [26] illustrates the *VM approach*, where REs are seen as instructions to be executed [41]. Similarly, ReCPU [25] explores RE matching by translating a RE into a set of instructions executed on a "dedicated CPU", offering the run-time adaptability of a CPU with the performance efficiency of an ASIC solution. However, its application to real scenarios is limited, and the absence of a communication subsystem prevents its adoption. Employing REs as instructions provides an attractive alternative methodology to the automata embedding for building an efficient hardware accelerator that can execute new "programs" without regenerating the hardware. Moreover, this methodology avoids the employment of explicit automata transition tables by exploiting the REs declarative language.

This research work takes inspiration from these last approaches and sees the REs as a programming language where REs are a sequence of operations repeated over a set of characters. Thanks to this approach, we build a custom run-time adaptable architecture for the REs domain only, called TiReX [24]. We rely on a custom compiler and ISA to represent REs belonging to the Regular Languages and on an optimized and energy-efficient microarchitecture that shows remarkable results. Moreover, we present a software–programmable multi-core architecture to exploit a multi-pattern or multi-chunk-of-data and further tailored for different scenarios. This work extends our previous one [24] providing unpublished details of the architecture and two novel architectural models. Thanks to these models, we enable the adaptation of latency-sensitive or throughput-sensitive workloads and pave the way to deploy TiReX-as-a-service thanks to the software-hardware application implementation on a public AWS F1 instance. Finally, we

1. https://github.com/google/re2

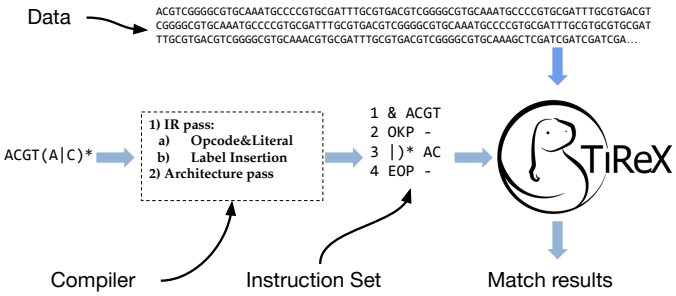

Fig. 1: TiReX flow for a Regular Expression.

present novel experimental results and analyses that showcase TiReX better energy efficiency over literature software and hardware solutions.

4 Design Methodology and Approach

This research work deals with REs as a programming language where REs are a sequence of operations repeated over a set of characters. The RE translates into a custom-defined ISA, on top of which we build our custom DSA called TiReX. The current ISA does not comprise advanced primitives of pattern matching that do not belong to Regular Languages or are derivable from those considered. Our DSA is not a general-purpose engine but is a software-programmable architecture tailored to the REs domain.

Figure 1 represents the workflow for executing RE matching on our DSA. The user provides both a RE (or a set of REs), the "program" TiReX will execute, and the data to analyze. The TiReX compiler translates the input RE(s) down to our low-level representation, i.e., the ISA. Then, the DSA takes as input the compiled code, which is loaded in the instruction memory. Finally, the architecture retrieves the data to analyze and outputs the matching procedure results. The next Sections first describe the compiler, then the ISA (Section 4.2), and finally the single- and multi-core architecture (Section 4.3 and Section 4.5), along with a performance model (Section 4.6).

4.1 Regular Expression Compiler

The compiler is a custom Python-based component that performs RE-to-ISA transformation. The compiler follows a Very-Long-Instruction-Word (VLIW) approach, where it is aware from the beginning of the underlining architecture characteristics, and it produces the binary according to the hardware features, e.g., the reference size determined by Execute parameters (Section 4.3.2). The compiler builds on two main passes: the Intermediate Representation (IR) pass and the architecture-aware pass. On the one hand, the IR level pass divides into two phases. The first one performs opcode and literal recognition and detects the need for a second pass for jump address insertion. The second (and optional) pass deals with labeled locations from the first pass and inserts the addresses for absolute jumps. On the other, the architecture-aware pass accounts for character parallelism abilities and word alignments.

TABLE 2: Opcode encoding of the operations.

opcode	*RE*	*Description*
000000	EOP	End of Pattern
010000	AND/&	And of cluster matches
001000	OR/|	Or of cluster matches
011000	.	Match any character
100000	(	Function call/sub-RE start
000100	)	Function return/sub-RE end
000001	)*	Match any number of sub-RE
000010	)+	Match one or more sub-RE
000011	)|	Match previous sub-RE or next one
000101	OKP	Open Kleene Parenthesis
000111	JIM	Jump If Match

4.2 Instruction Set Architecture

The TiReX ISA approach relies on a VLIW-like machine, where multiple operations are bound together to form a bundle. Table 2 shows the primary operations that compose the TiReX ISA primitives. These primitives are composable operators to form different REs, with their literal part, and are at the basis of all the complex REs, such as repeating n times the string x, i.e., $x^{\{n\}}$. The compiler decomposes advanced REs into TiReX primitives. Complex REs carry the cost of a bigger program size; therefore, the instruction memory has to accommodate a reasonable amount of instructions or to provide an adequate memory hierarchy. For example, the RE $a^{\{3\}}$ translates into a sequence of *aaa*, hence turned into a simple concatenation of three *'a'* characters. Each instruction is divided into two fields: the opcode field, where we encode different pluggable operations, and the remaining part, which is called reference or instruction operands. The reference field usually contains the number of parallel characters a TiReX core can crunch, or, in case of particular operators, it contains helpful information such as the branch target address. In our implementation, the instruction word is *38-bit* long, and it has *6 bits* for representing the opcode field, while the remaining *32 bits* represent the reference field, i.e., at most four parallel standard ASCI characters. TiReX operations divide into three main groups: Character Match, Control Flow, and Support.

Character Match Operators represent the basic operations on REs to match characters with boolean logic, like the AND instruction to match a fixed sequence of characters or the OR instruction to accept several alternative characters.

Control Flow Operators represent an advanced class of operators to compose advanced REs against the Character Match operators. In particular, these operators control the flow of the instructions (e.g., jump), perform a "function" call preserving the "context" (e.g., matching process status, current data and instruction address, prefetching hints). These operators are described as follows:

- **EOP operator**: The EOP is a special instruction whose purpose is to signal the end of the program (i.e., the end of the RE matching procedure).
- **(operator**: The open parenthesis operator represents a "function call", or sub-RE, and translates to context preservation.
- **) operator**: The closed parenthesis operator represents a "function return"; therefore, it instructs the processor to restore the previous context.

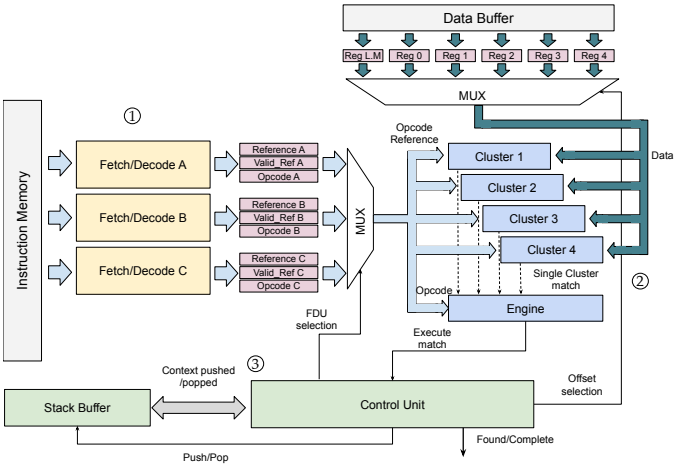

Fig. 2: Details of the core logic, with the pipeline components and the reference to Section 4.3 subsections.

- **$)^*$ and $)^+$ operator**: These symbols represent the Kleene operators at the end of a function call, which, like a for-loop, re-execute the loop body of the matching code until either a mismatch occurs or the string ends.
- **$)|$ operator**: This operator composes chains of OR-ed REs: the sequence of functions matches when any of the REs matches, in which case the hardware can skip the remaining functions in the chain.

Support Operators are instructions to support the Control Flow ones and increase performance. Indeed, they are needed to inform the architecture of the instruction memory location to jump. In particular, the function call operator "(" can ambiguously lead to a function with the Kleene operators or an OR chain: in the former case, the architecture has to re-execute the function, hence jumping to a previous instruction, while, in the latter case, the flow of instructions goes forward and the architecture can skip the following REs in case of a match.

For this reason, we specialize the "(" operator to discriminate the two cases and inform the architecture prefetchers on which instructions and data TiReX has to load in the next cycle for all the possible comparisons results. **OKP operator** hints the architecture about the presence of a for-loop-like sequence, hence showing a possible backward jump in a matching case or a forward jump in a mismatch case. **JIM operator** gives the flow-controller a hint of the presence of OR-chained function calls, therefore preparing for a possible forward jump in the flow in case of a match. In contrast, a mismatch causes the architecture to load the next instruction and re-read previous data (which can be cached).

4.3 Single-Core Architecture

TiReX DSA has a two-stage pipeline divided into Fetch/Decode and Execute stages. The core tracks the RE procedure status, and it can be in two different states (i.e., match or not match state), which determine different execution flows. Figure 2 shows the block design of the microarchitecture and implementation details. The main components of the core are the Instruction Memory (IM), which stores the program instructions, the Fetch/Decode Units (FDUs) for the homonymous stage, the Data Buffer (DB), which stores a portion of the input data necessary for the matching process, the Clusters and the Engine, which form the Execute Unit (EU), and the Control Unit (CU), devoted to the control of the whole matching process.

A general flow of the matching process starts with the loading of the compiled RE into the IM, then the FDU loads one instruction at a time, decodes it, and propagates the control signals (e.g., the *valid_ref* signal, which indicates how many valid characters are in the reference in one-hot encoding scheme) and the instruction operands to the EU and the CU. The EU loads the input characters from the DB and searches for patterns according to what received from the FDU, emitting a match to the CU when the current input characters match the pattern encoded in the current instruction. Finally, the CU exposes signals to the external logic to indicate the completion of the matching process and the presence of a match. The basic RE matching is an intrinsically sequential control dominated flow that analyzes a single character per clock cycle [42]. We now describe how TiReX deals with control hazards within a RE instruction flow (Section 4.3.1) and how we increase the number of characters analyzed in a single clock cycle (Section 4.3.2) showing remarkable performance (Section 6.3.1).

4.3.1 Fetch/Decode stage

The Fetch/Decode stage consists of three copies of an FDU, exploited to prefetch every possible instruction flow. An FDU takes the bundled word from the IM and unpacks it in the three pieces of information needed for the RE matching procedure: the opcode, the reference, and how many references are really in that field, i.e., the signal *valid_ref*, as shown in Figure 2. Thanks to the Control Flow Operators, which we tailored to change the instruction flow, we exploit different instruction-pre-fetching mechanisms. By instantiating three different and specialized FDUs (marked A to C in Figure 2), the core can avoid cycle losses in the case of *not match* or case of special instructions such as the JIM or the OKP. The identified flows are mainly three.

Sequential Execution: The RE matching process can run in the simple sequential execution flow. Indeed, the FDU-B continuously prefetches the next instruction, essential when a match is found. Thus, we cover the basic sequential execution flow of a "program".

Instruction Rollback: The "program" can find a false initial match up to a certain point. Indeed, discovering a false partial submatch requires rollback the execution to the first "program" instruction. Since restoring everything in case of a false match leads to large cycle loss, FDU-A keeps a copy of the very first instruction and its control signals.

Special Jump: Another possible performance degradation source comes from jump instructions. Indeed, as in general-purpose processors, control flow modifications that depend on run-time computations, such as a comparison result, require special hardware components (e.g., branch resolution anticipation, dynamic branch predictors) or clock cycle stalls for control resolutions. Jumping back and forward in the "program" leads to control hazards. Considering a backward jump, as for the Kleene operations, or a forward jump, as for OR-chain, we need to know the target

instruction. To handle this, the second pass of the compiler hints at the core by inserting this target jumping address. The FDU-C leverages compiler hints to prefetch instruction located after the sequence of OR-ed REs (in case of JIM) or the initial instruction of a sub-RE (in case of an OKP).

4.3.2 Execute stage

The other datapath portion is the EU, and it consists of two parts: the Clusters and the Engine. The Clusters are the components that compare input data against input reference; hence, the more the Clusters, the more the characters the core can analyze. The most basic primitive in the RE matching process is a comparator for single character-to-character comparison. More "advanced" primitives are *AND*, or concatenation of characters, and *OR*, or alternation that in our ISA corresponds to the Character Match operations. These primitives correspond to TiReX Clusters, and each of them takes as input the reference characters and the data characters. The EU has two orthogonal design parameters that determine the parallelism degree in the character analyzable per clock cycle. The first one is the width of the Clusters, called *ClusterWidth* parameter. Figure 3 shows a ClusterWidth equal to four. Hence, that Cluster performs at most four character comparisons in a clock cycle. Depending on the opcode, we feed the Cluster with different data characters. In the case of an *AND*, the four comparators will receive different characters, as in Figure 3. In the other case, i.e., the *OR*, each comparator will receive the first data character of the substring (e.g., the first C in the CCGT substring), and compare against all the reference characters. In this way, the ClusterWidth determines the maximum amount of characters the core can analyze with the *AND* opcode.

The other design parameter regards how many Clusters the EU has, called *NCluster*. Considering Figure 3, NCluster is equal to four. Following the previous data feeding scheme, each Cluster will receive a substring shifted by one of the data characters, i.e., Cluster one will receive CCGT, Cluster two CGTA, etc. Hence, considering an *OR*, we feed each Cluster with a single data character, while, with *AND*, we feed them with four characters. The Clusters then compare such characters against the reference ones. In this way, the NCluster parameter determines the maximum number of characters the core analyzes with an *OR* opcode.

The Engine collects all the intermediate results from the Clusters, knows the current Character Match operation, and combines them to produce an aggregate result for the Control Unit. The Engine works in two different ways, depending on being in matching or not matching state, and it controls which Cluster is active or not, stalling the execution if necessary. All these components are almost combinational; hence, balancing the EU parallelism is crucial to avoid unused or small resources.

Overall, the architecture might have different inputs feeding schemes depending on the instruction result (*match/not match*) and the number of matching characters (from 1 to ClusterWidth). For this reason, we adopt different data-shifting mechanisms for the DB, where we use registers to store possible inputs for the next instruction.

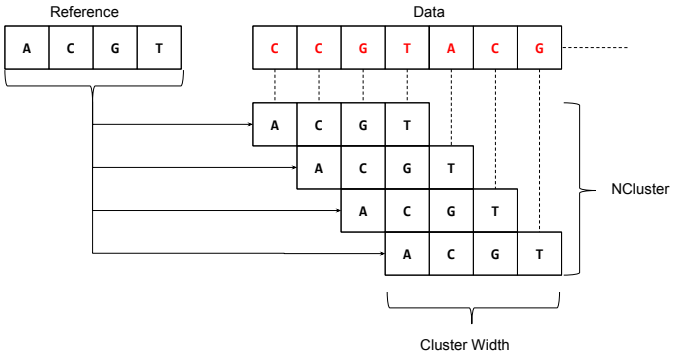

Fig. 3: Detailed view of the EU during the first AND instruction execution.

TABLE 3: Execution example of a Regular Expression on a single core (the effective data analyzed are underlined).

Instruction	Cycles							EU Data
	#1	#2	#3	#4	#5	#6	#7	
& ACGT	FD	EX						CCGTACG
& ACGT		FD	EX					ACGTATT
OKP			FD	EX				ATTGCAC
|)* AC				FD	EX			ATTGCAC
|)* AC					FD	EX		TTGCACT
EOP						FD	EX	TTGCACT
								Match found

Instruction: *ACGT(A|C)**
Data: *CCGTACGTATTGCACTA*

4.3.3 Control stage

While the EU handles the Character Match operations, the CU handles all the other complex operations, e.g., Kleene ones. Indeed, this unit is a centralized controller that synchronizes the different prefetching and prediction mechanisms. It is aware of the current RE matching procedure status, e.g., match or not match, enabling dynamic scheduling of the proper instruction, keeps track of the instruction and data pointers, and generates the proper addresses for the memories. Moreover, whenever the Control Unit finds a function call, e.g., the "(" operation, it pushes the current RE matching context to the Stack Buffer, our context memory, and synchronizes the datapath to work in this new context.

4.4 Regular Expression analysis example

Table 3 shows an example of the matching process on the TiReX core. First, the FDU-A retrieves the very first instruction producing the *AND* opcode, ACGT as reference, and *valid_ref* signal equal to 1111, since every character in the reference is valid. Then, the EU performs the comparison, resulting in a mismatch. Indeed, as shown in Figure 3, the reference is matched against the input stream in NCluster exact starting positions, given the initial not-matching state. Thanks to the presence of multiple FDUs, the CU does not flush the pipeline as the first instruction is still present in FDU-A. The data *pointer* jumps four characters ahead (due to the presence of four Clusters), and the next batch of characters is checked against the first instruction. This process is executed until a valid intermediate match is found (this happens at cycle #3 in the example), which moves the

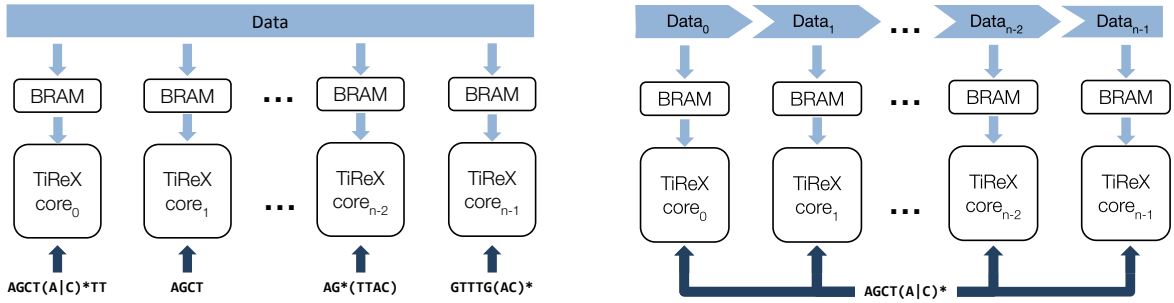

Fig. 4: Multi-core modalities with multi REs single data stream, or single RE multiple data streams, respectively.

CU to a matching state and increments the program counter and data pointer by one and four, respectively. This offset is computed from the Engine and depends on which is the matching Cluster. The core continues until it reaches the last instruction (EOP) and reports the final result, in this case, a match found.

4.5 Multi-Core Architecture

The proposed single-core DSA adopts improved parallelism at the character level. Although we could consider it similar to a SIMD architecture, the domain requirements (i.e., real-time analysis and flexibility) are not yet satisfied. Hence, we propose a multi-core architecture based on the replica of the same tile described previously. Each tile has its private memories, separate for data and instruction, and it is software programmable to work in two different ways depending on the needed parallelism level, as shown in Figure 4. According to the application, we tailor the system to the RE recognition process, which can operate in two different modalities, i.e., MISD- and SIMD- like.

Multiple REs Single Data Stream. Considering a scenario where there is a need to analyze a wide range of patterns simultaneously, e.g., Signature-Based Detection [22], we designed a dedicated multi-core architecture based on the TiReX tile described previously. Each core is equipped with its private instruction memory, i.e., different REs that it has to deal with, while the data stream is the same for every core. In this way, we can increase the number of patterns processed per single execution while keeping a similar data crunching ability. However, this execution mode requires further investigation in the topology of the architecture, in the interconnection logic, and can lead to data divergence and issues related to memory coherency when considering terabytes of data, such as in the genomic case. We will not address this issue here because we think it is out of the scope of this work, and we think that exploiting memory coherent protocols (e.g., OpenCAPI, CCIX, or CXL) is a sufficient solution. **Single RE Multiple Data Stream.** In contrast to the previous version, this operational way exploits the parallel core to increase data crunching rates. Indeed, a fast scan rate is essential when considering a vast amount of data to analyze, such as the Human Genome, or big database. To improve our single tile abilities, we fill each tile IM with the same RE while providing different data stream portions. These chunks of data are not independent a priori, and potential matches in the crossing regions may happen. Therefore, we do not consider a sharpened cut, but we adopt a simple heuristic to provide an overlapping region to avoid false mismatch at a negligible overhead cost. The data splitting task is currently handled at the software level and relies on a domain-expert specified threshold that indicates the maximum length of a match, translating into the overlapping region. We compute the *batch size*, or B_{size}, for each of the *i-th* core of the N ones, through $B_{size} = \frac{S_{data}}{N_{tc}}$, which divides the *size of the data* (S_{data}) by the *number of cores* instantiated in the system (N_{tc}). With the batch size, we compute the *End of Data*, or EoD_i, $\forall i \in N \; EoD_i = min(B_{size} \cdot (i+1) + Tr, S_{data})$ per each *i-th* core by simply adding a user-defined *threshold* (*Tr*) to ensure coverage of a possible matching sequence of *Tr* maximum length. Finally, we compute the new *Start of Data*, or SoD_i, except for the first core, which computes from the very beginning, $\forall i \in N - \{0\} \quad SoD_i = EoD_{i-1} - Tr$, as the difference from the previous core ending point minus the domain-specific threshold, ensuring coverage of a possible matching sequence of *Tr* maximum length.

4.6 Performance Analysis

Here follows a theoretical performance model to analyze the expected throughput in both the worst and the best cases. Although the matching process is highly data-dependent, the time required to process a single character is the main critical parameter measured in $\frac{ns}{char}$. This *time-per-char* metric (the lower, the better) is mainly affected by the process status of the RE, i.e., match or not match case. Being in a not matching case, the time per char T_{nm} depends on the system frequency, F, and the number of characters processed per clock cycle. In the worst case, it is *NCluster*; in the best case, it implies the full utilization of the Execute unit. Consequently, $T_{nm} = \frac{1/F}{NCluster}$ and $T_{nm} = \frac{1/F}{NCluster + ClusterWidth - 1}$. On the other hand, considering a matching state, we span from a single character per clock cycle in case of union, or OR, $T_{mu} = \frac{1}{F}$, to a number of *ClusterWidth* character processed in the concatenation case, or AND, $T_{mc} = \frac{1/F}{ClusterWidth}$. Substituting the effective design parameters, we can compute the worst and the best case of the time per char required to analyze a character at the single-core architecture level. Additionally, we can model the expected throughput of the architecture, both single- and multi-core, adopting the presented equations and estimate the *bitrate* of the system in *Gb/s* as $B_x = \frac{1}{T_x} \cdot 8 \cdot N_{tc}$, where T_x is the time per char obtained with the previous equations, N_{tc} is the number of TiReX cores.

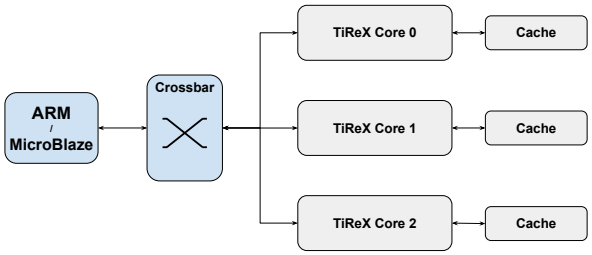

Fig. 5: Block design of the multi-core TiReX architecture.

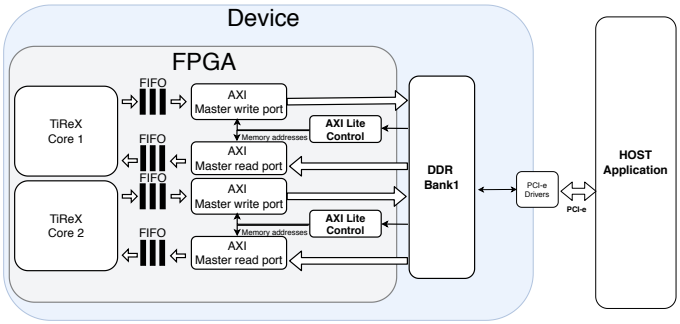

Fig. 6: Architectural model of the external host implementation of the TiReX system.

These equations provide an estimate of bitrate values for possible implementations of TiReX with a variable number of cores and frequency, allowing us to foresee how performance scales up as the number of cores and frequency increases. This makes it possible to estimate the goodness of the various TiReX implementations before going through empirical measurements.

4.7 Architecture analysis summary

To summarize, the proposed architecture relies on a custom ISA for RE matching flexibly. In contrast with many NFA/DFA approaches, the proposed methodology handles different run-time tunable REs without modifying the underlying architecture. The current execution model of the architecture is based on a depth-first-like approach, where the execution of the instructions is intrinsically sequential [42]. Though it suffers from classical backtracking issues, this model could lead to possible future solutions based on a breadth-first-like model, similar to a theoretical NFA, achievable through mainly NFA logic embedding. Additionally, the architecture has two levels of parallelism to increase execution efficiency. The first one resides in the number of characters analyzed per clock cycle, which is a design parameter, and all the prefetching mechanisms to handle multi-flow executions. The second level of parallelism relies on a private memory multi-core architecture that can employ two different theoretical execution models, i.e., MISD- and SIMD- like. The first model increases the number of parallel REs analyzed, while the second one increases the parallelization of vast amounts of data.

5 Scenario-Specific Architectural Models

To demonstrate the adaptability of this work, we implement the system following two different architectural models. The first model targets all those platforms in embedded-like scenarios, e.g., a System on a Chip (SoC) comprising a CPU and an FPGA. This model targets latency as the primary metric and tries to increase energy efficiency as much as possible. This specific architecture is called *Embedded Host*, given that the host application executes on a CPU tightly coupled to the FPGA. Instead, the second model involves server-like systems targeting a throughput-oriented scenario, where tackling highly intensive data workloads is paramount. In this model, called *External Host*, the host application executes on the server processor connected through a PCIe bus with the accelerating device.

5.1 Embedded Host

This model targets a more constrained execution environment where the FPGA is tightly coupled with a CPU. Examples are the PYNQ platform provided with ZYNQ technology, which embeds an ARM Cortex A-9, or a soft-core, such as the MicroBlaze, directly instantiated on the programmable logic. The embedded processor and TiReX communicate through lightweight AXI-Lite ports for fast, small transactions of both control or data of 32 bits per transaction. We instantiate a multi-core architecture interconnected through a crossbar, as in Figure 5, by a simple tile replica, each of which has its additional private cache memory, implemented through BRAMs, to exploit the available reconfigurable fabric fully.

The overall system starts with the processor loading the various caches and instruction memories of all the TiReX cores instantiated in the system following a SIMD or MISD fashion depending on the user's needs. Once filled, the processor enables the beginning of TiReX cores computation. Once one of the cores produces a result, the search completes, reporting the outcome to the host. Section 6.3.1 shows the results this architectural model can achieve in latency-sensitive scenarios.

5.2 External Host

Thanks to the presence of cloud FPGAs, everyone can access a high-end cloud FPGA that communicates through a peripheral bus to an external host processor. We devised the external host model to target such a scenario. Consequently, we can implement TiReX on systems like the AWS F1 instances, which contain high-end FPGAs equipped with four physical DDR ports. Each physical port can transmit up to *512 bits* of data per clock cycle, for a total of *2048 bits*. Moreover, we can time-multiplex each physical port for logic port interleaving and instantiate a wider number of cores. Therefore, we decide to exploit this chance by deploying the kernel via the SDAccel framework, as shown in Figure 6. The tool automates some system design steps and provides a communication infrastructure, which requires specific design interfaces and APIs for the transmission phase among host and device through the high-bandwidth

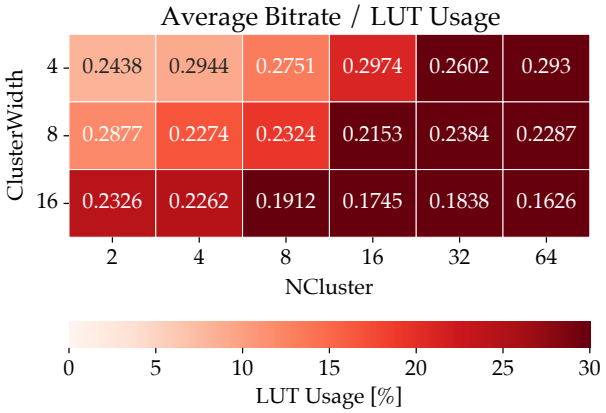

Fig. 7: Heatmap representing ClusterWidth and NCluster space exploration considering the ratio of average bitrate and LUTs usage on a PYNQ-Z1. The darker the color the closer (or beyond) the resource usage to the given budget for a TiReX single-core. The (4,4) configuration reports a good trade-off while respecting the budget.

PCI-e link. Each tile has an AXI-Lite interface for control exchange, while most of the data exchange goes through an AXI-Master port attached to the DDR bank. The interface requirements need additional *glue logic* and FSMs to handle instruction and data transmission and some performance counters. The Master port passes the data through a FIFO to provide a back-pressure mechanism for DDR-tile exchange, and the same happens in the other direction. We replicate the tile and the additional interfaces to provide a multi-core architecture, where each core is independent.

The final system flow starts with the RE(s) compilation. We feed the data according to the operational way and apply the splitting heuristic if needed. Then, the host writes the data to the DDR through the PCI-e and starts the FPGA kernel. Each core, which has a private portion of the assigned DDR bank, reads and loads the instructions and then retrieves the data to analyze. Once done, the overall kernel writes back the computation results, while the host waits for all the cores to end and reads the match outcome along with performance counters, such as clock cycle count and matching position. Section 6.3.2 details the performance results the External host architectural model achieves in throughput-oriented scenarios.

6 Experimental Setup and Results

We designed TiReX architecture in VHDL, with additional glue logic in System Verilog. We targeted three different Xilinx platforms: the VC707 and the PYNQ-Z1 boards with an embedded host model, while the external host model makes use of the AWS F1 instance powered by a Virtex UltraScale+ 9 (VU9P) FPGA attached through a PCIe connection, paving the way to deploy TiReX-as-a-Service. While the VU9P system is a more throughput-oriented solution, thus suggested for a server-like system use case, the PYNQ-Z1 system represents a more constrained use case, such as embedded systems for an autonomous vehicle or IoT scenarios. We use Xilinx Vivado and SDK 2016.4 to generate the bitstream and manage the bare-metal host for the embedded host model. For the external host model, we use Xilinx SDAccel 2016.4 to implement the system on the VU9P, and we exploit the OpenCL library for the host. We employ four different state-of-the-art software to compare against TiReX. The considered baseline is FLEX, which produces a DFA for each RE in a C file that needs to be compiled. Then, we employ Grep, which builds the matching NFA at run-time, Google's RE2, an optimized multi-threading C++ library that builds an NFA as well, and Hyperscan [31] by Intel, which applies an intensive preprocessing mechanism to decompose the RE(s) into a SIMD NFA. We compile FLEX code, RE2 library, and Hyperscan with *-O3* optimization level and adapt the software tools to stop as soon as a match is found. In particular, for RE2 we use the `RE2::PartialMatch` function, while for Hyperscan we employ the `hs_compile` with the `HS_FLAG_SINGLEMATCH` flag. For reference CPU, we have the ARM Cortex A9 of the PYNQ-Z1 running at 650 MHz, an Intel i7-8750H with six cores and a peak frequency of 2.2 GHz, and a dual-socket Xeon E5-2680 v2 with a total of twenty cores and a peak frequency of 2.8 GHz.

We focused on bioinformatics and networking to test TiReX in real-life fields with state-of-the-art benchmarks. We divided the experiments into two sets: a *small* one, which we indicated as *S*, aimed at a latency-oriented scenario, and a *large* one, told as *L*, targeting a throughput-oriented scenario. For the S scenario, we used the first *16 KB* of the first human chromosome[2]. We matched this input against the three REs, whose complexity increases to measure the internal core execution latency correctly and stress the performance. For the L scenario, we first selected the *E-coli* bacteria *proteome*, about *8.5 MB* of data retrieved from UniProtKB database[3] and used *PROSITE* [43] as the REs dataset. PROSITE consists of a biologically significant database and patterns formulated to reliably identify which known family of proteins the new sequence belongs to, including the REs. Then, we tested the throughput with the PowerEN benchmark from ANMLZoo [1] with 1MB of data, which has been employed in the validation of the IBM PowerEN for networking function embedding at the edge. We analyze both the execution time and energy efficiency of the proposed architecture for these testbeds. We collect the power consumption of the whole board, hence including the host, through a Voltcraft 4000 energy logger for the VC707 and the PYNQ-Z1, while we take the Thermal Design Power (TDP) for the VU9P as from a literature work [44], i.e., 42W, excluding the host. Besides, we use the TDP for the Intel CPUs while we measure the ARM A9 power consumption using the energy logger. We compute the energy efficiency as $throughput/power\ consumption$, where throughput is the inverse execution times.

6.1 Exploration of Design Parameters

Here, we explore ClusterWidth and NCluster design parameters employing an open-source automation tool [45] and targeting a PYNQ-Z1 device. The exploration aims at finding a trade-off of these parameters that shows noteworthy

2. ftp://ftp.ncbi.nlm.nih.gov/genomes/H_sapiens/CHR_01/hs_alt_CHM1\1.1_chr1.fa.gz
3. https://www.uniprot.org/uniprot/?query=ecoli&sort=score

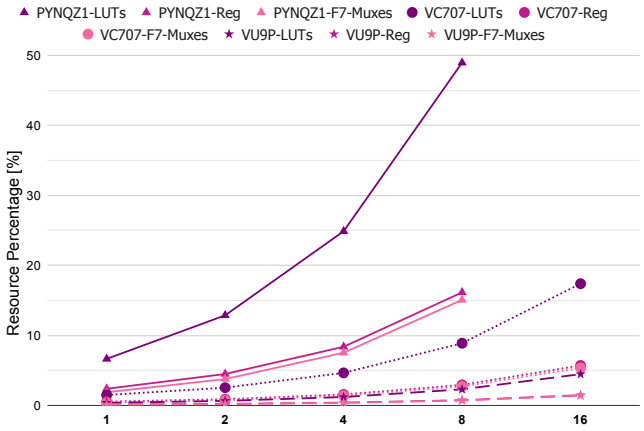

Fig. 8: Resources scaling (darker to lighter: LUT, Reg, Mux) against core number on the PYNQ-Z1 (triangles), VC707 (dots), and VU9P (stars).

performance while keeping a low critical resource footprint (i.e., LUTs). Figure 7 reports a heatmap with the ratio of average bitrate, and LUTs usage. We set a maximum budget of 30% of LUTs for a single core since we aim at scaling in the core number; hence, Figure 7 shows darker configurations that are close (or beyond) this upper bound. Among the lighter configurations, (4,4) and (4,16) report remarkable performance, but the (4,4) configuration shows less resource usage. In this context, the (4,4) represents the optimal trade-off of delivered bitrate per employed LUTs that eases the core scaling even with resource-constrained devices.

6.2 Multi-core scaling synthesis results analysis

Here, we aim to exploit the reconfigurable fabric as much as we can to scale the core number. Figure 8 shows the resource scaling with power-of-two cores (though every number of cores is suitable) architectures; in this way, we explore a reduced portion of the design space and simplify the memory traffic model. Given the limited number of physical memory ports, those channels have to work in interleaved mode. However, increasing the number of nodes attached in an unbalanced way may not result in deterministic performance prediction and load management requirements. To devise a multi-core architecture, we first investigated the area utilization of a single core implementation, which can provide insights into the number of cores that can fit in a multi-core design. Figure 8 shows a small resource footprint of a single TiReX core on the three target platforms in all cases. We report the most relevan resources utilization, given the nenglibile amount of the others. Therefore, they do not represent an issue at all. Although the numbers in Figure 8 suggest that it is possible to deploy a high number of cores on each platform (like more than a hundred for the VC707), this is not possible due to routing-related issues. Indeed, a high number of cores leads to congestion on the interconnection paths towards the processor and the communication infrastructure, with very high fan-outs hindering the routing phase. These issues limit the deployable cores to 16 on the VC707, given that designs beyond 32 cores fail during the routing phase. However, on the PYNQ platform, the number of deployable cores scales down to 8 since designs with more cores (from 16 on) or targeting higher frequencies fail during the routing phase, having fewer resources available than the VC707. We envision designing a dedicated interconnection network to scale to more cores while keeping acceptable frequencies. However, this is beyond the scope of this work.

Conversely, on the AWS F1 instance, the limit of 16 cores is not due solely to routing problems, even if the area utilization also comprises the more significant AXI logic. In this scenario, the limitation is due to the number of logic AXI-Master ports that can be instantiated. The toolchain constraints to up to 16 different AXI-Master logic ports. Indeed, with internal design unchanged, each core retrieves its data portion from the DDR banks after a setup phase from an external host through PCIe. To instantiate a higher number of cores, it is necessary to insert back-pressure mechanisms in the interconnection logic and the internal logic itself for handling variable unavailability of data. Indeed, the data transfer rate is limited to *2048 bits* per clock cycle, since there are at most four-DDR ports available for each DDR memory bank, each one capable of transmitting *512 bits* per cycle.

6.3 Performance analysis against software references

Alongside the synthesis results, we compare our system against state-of-the-art software approaches in terms of performance, accounting for the latency (*S* scenario) and the throughput (*L* scenario), and considering the energy efficiency. The following experiments evaluate the multi-core architecture presented in this paper in SIMD modality and the single-core architecture on mainly the VU9P device. We adopted the heuristic presented in Section 4.5 and a threshold *Tr* equal to 100, which we manually computed to be suitable for both the *S* and *L* scenarios. We analyze the data transfer overhead from the host processor to the target FPGA on the external host model, focusing on the L scenario. The overhead is generated both by hardware and software components. On the one hand, the data transfer occurs via a PCIe-Gen3x16 connection, which has a transfer rate of up to 15.6 GB/s, hence providing a latency based on the input data size. For example, the E-coli bacteria protein dataset size is *8.5 MB* and the resulting transmission time is around *544* μs. On the other hand, the software introduces non-negligible overheads: the time for the call to the OpenCL APIs, the CPU context switch time, the time required for the interrupt to be served, and the additional asynchronous driver time. The sum of the previous software-based overhead times and the system setup time for the execution of the first instruction is around *70* μs. This value has been computed by injecting empty instructions and empty data into the core, eliminating the data transfer overhead. The transfer rate, along with the software-based overheads, provides the base execution times of the TiReX system that has to be taken into account.

6.3.1 The S scenario

This scenario evaluates the latency of the considered approaches on the first *16 KB* of the first human chromosome. We employ three different tests to stress the methodology

TABLE 4: Performance results (in bold the best) of the S scenario tests, chosen to stress incrementally our architecture (FLEX on the i7 is the baseline, while FLEX on the A9 is omitted being far slower than Grep on A9).

Method	Architecture	Exec. Time [μs]			Speedup			Energy Efficiency [$1/(ms \cdot W)$]		
		Test 1†	Test 2‡	Test 3*	Test 1†	Test 2‡	Test 3*	Test 1†	Test 2‡	Test 3*
Grep	ARM A9 650 MHz	11963	12185	12374	0.02×	0.01×	0.02×	0.02	0.02	0.02
RE2	ARM A9 650 MHz	589	353	391	0.46×	0.34×	0.67×	0.42	0.70	0.64
FLEX	Intel i7 2.2 GHz	271	121	263	1×	1×	1×	0.08	0.18	0.08
Grep	Intel i7 2.2 GHz	492	805	221	0.55×	0.15×	1.18×	0.04	0.03	0.10
RE2	Intel i7 2.2 GHz	50.69	29.30	41.98	5.35×	4.13×	6.27×	0.44	0.76	0.53
Hyperscan	Intel i7 2.2 GHz	78.92	53.58	35.08	3.43×	2.25×	7.50×	0.28	0.41	0.63
FLEX	Xeon E5 2.8 GHz	598	136	404	0.45×	0.88×	0.65×	0.01	0.06	0.02
Grep	Xeon E5 2.8 GHz	205	108	336	1.32×	1.11×	0.78×	0.04	0.08	0.02
RE2	Xeon E5 2.8 GHz	34.48	23.02	28.28	7.86×	5.25×	9.30×	0.25	0.38	0.31
Hyperscan	Xeon E5 2.8 GHz	59.49	52.21	27.95	4.56×	2.32×	9.41×	0.11	0.17	0.33
TiReX	VU9P 1 core 299 MHz	37.66	18.32	29.63	7.19×	6.60×	8.87×	0.63	1.30	0.80
TiReX	PYNQ-Z1 8-core 70.5 MHz	7.20	8.21	30.30	37.63×	14.73×	8.67×	**41.75**	**36.61**	9.92
TiReX	VC707 16-core 130.1MHz	2.07	4.54	3.36	130.9×	26.65×	78.27×	22.74	10.37	**14.01**
TiReX	VU9P 16-core 202.7 MHz	1.03	0.75	2.96	**263.11**×	**161.33**×	**88.85**×	23.11	31.74	8.04

$^\dagger ACCGTGGA \quad ^\ddagger (TTTT)^+CT \quad ^\star (CAGT)|(GGGG)|(TTGG)TGCA(C|G)^+$

effectiveness, from string matching to more complex RE matching tests. Table 4 summarizes the performance results of the S tests run on the various platforms and compared against FLEX, Grep, RE2, and Hyperscan without considering *data transfers* or *I/O parts* for all the considered methods. On the other hand, we account for the preprocessing mechanism of Hyperscan since this phase highly influences the matching methodology and performance. Moreover, we did not evaluate Hyperscan on the ARM A9 since it does not officially support ARM-based architecture. Considering TiReX multi-core implementations, we achieve speedups ranging from a 0.92× up to 33.47× against the best software implementations of Hyperscan and RE2. The slowdown comes from the single-core on the VU9P and the PYNQ-Z1 with eight cores against RE2 and Hyperscan on Test 1 and 3. We must consider the differences in the execution model, our DFA-like versus software NFA-like, and the considered platforms. Indeed, both the software tools run on a server-class CPU, while the PYNQ-Z1 is an embedded device, and the running frequencies are incredibly different, i.e., 2.8 GHz versus 70 MHz. Hence, the achievement of performance in line with state-of-the-art tools on a server-class processor demonstrates the remarkable benefits of our approach even when employing an embedded device. Moreover, scaling cores on the VU9P showcases a considerable improvement (i.e., a top of ∼69×) against the state-of-the-art software solutions, validating the proposed multi-core approach. Considering the energy efficiency presented in Table 4, or ratio of throughput over power consumption, TiReX DSA provides a higher degree of efficiency than software solutions. In particular, TiReX multi-core delivers a remarkable energy efficiency spreading from about 12× up to 95× against the most efficient CPU implementations. These results demonstrate TiReX specialization advantages from single-core to multi-core in execution times and energy efficiency. Although the 8-core PYNQ-Z1 implementation achieves the same speedup as the VU9P single-core one, we should consider the working frequency's role. Indeed, the two running frequencies differ by 220MHz, which gives a non-negligible lead. Additionally, we must consider that TiReX execution times come from the hardware performance counters and account for the shortest matching time. These results compare fairly with those on the CPU since we ran the whole matching process inside a loop where the first ten iterations were used to warm up the L1 data and instructions caches (which are, for all CPUs, 32KB in size, hence they can host the whole input and program code), and we averaged the run-time of 30 iterations. Instead, we averaged 30 iterations and subtracted a fixed amount of time to let the system open the file to compare fairly with Grep.

6.3.2 The L scenario

In this scenario, we focus more on the throughput-oriented solutions, restricting our implementation to the external host model and the AWS F1 instance, as it is the most representative for a server-like environment, accounting for the overhead presented at the beginning. Similarly, we consider the Intel Xeon CPU only. We consider different datasets from the S scenario, as described at the beginning of the Section, for both REs and data to analyze.

Figure 9 shows the geometric means of the execution times (the lower, the better) achieved on the two benchmarks by the considered approaches, with the speedup of the 16-core version of TiReX on the VU9P reported on top of the bar. It is noteworthy that PROSITE exhibits more matching situations, while PowerEN reports few matches. TiReX single-core already offers a speedup compared to tools like Grep and FLEX, showing the benefits of TiReX domain specialization achievable at the single-core level. Scaling TiReX to a multi-core architecture delivers a speedup of 1.233× and 1.585× over state-of-the-art software such as RE2. Moreover, our DSA can reach higher performance by further increasing the number of cores, but, as stated in Section 6.2, the tools and technology limit the number of AXI-Master ports we can instantiate on the VU9P.

Figure 10 shows the energy efficiencies (the higher, the better) of the considered literature software approaches and TiReX single and multi-core architectures. On top of the bars is reported the improvements of TiReX VU9P 16-core compared to the other approaches. We consider for all the

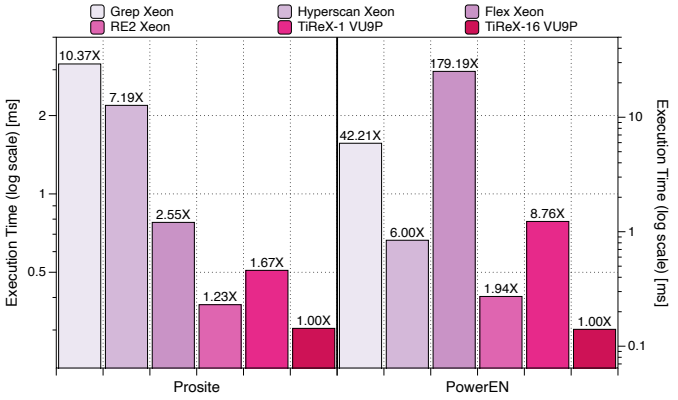

Fig. 9: Execution time geometric means (the lower, the better) and speedup of TiReX-16 VU9P with respect to other solutions on L scenario.

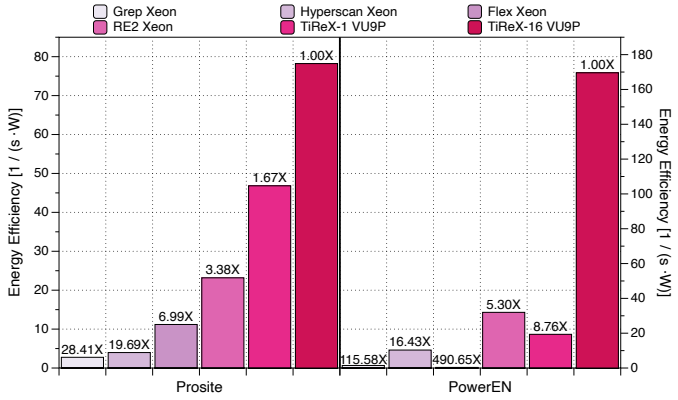

Fig. 10: Energy efficiencies (the higher, the better) and improvement of TiReX-16 VU9P with respect to other solutions on L scenario.

TABLE 5: Comparisons in terms of platform, bitrate, flexibility and benchmarks with the Related Work (in bold the best).

	Solution	Target Platform	Frequency $[MHz]$	Bitrate $[Gb/s]$	Energy Eff. $[Gb/(s \cdot W)]$	Run-time adaptability	Benchmarks
SPA	**VC707 16-core**	FPGA (28nm)	130	16.65 - 116.48	0.78 - 5.48	✓	Synthetic, PROSITE [43] PowerEN [1]
	PYNQ 8-core	FPGA (28nm)	80	4.51 - 35.84	1.35 - **10.77**	✓	Synthetic, PROSITE [43] PowerEN [1]
	VU9P 16-core	FPGA (16nm)	202	25.94 - 180.99	0.62 - 4.31‡	✓	Synthetic, PROSITE [43] PowerEN [1]
	ReCPU [25]	ASIC (180nm)	318	10.19 - 18.18	N/A	✓	Synthetic
	Brodie et al. [33]	ASIC (N/A)	133	16	N/A	✓	Synthetic, Snort, Forensic
	BFSM [10]	PowerEN™ (45nm)	2000	20 - 40	N/A	✓	Open source REs
SDA/IMA	Meiners et al. [3]	TCAM (N/A)	N/A	10 - 19	N/A	✓	Bro, Snort, L7-filter, Networking
	REAPR [16]	FPGA (20nm)	222 - 686	0.20 - 0.60	0.06 - 0.20	-	ANMLZoo [1]
	FlexAmata [17]	FPGA (16nm)	252	8.75†	0.21‡	-	ANMLZoo [1]
	HARE [2]	FPGA/ASIC (28/45nm)	100/1000	3.20 / 256	N/A / 2.13	-	Synthetic, Snort
	jDFA [11]	FPGA (28nm)	150	230 - **430**	N/A	-	Bro, Snort and L7-filter

†Rescaled from 3.7 Gb/s at 213MHz and then doubled for difference in alphabet size (16-bit vs. 8-bit) ‡Using 42W TDP [44]

approaches the TDP of the respective devices, i.e., Xeon (115W) and VU9P (42W [44]). The 16-core design achieves energy efficiencies that range from ∼3× to peak of ∼490×.

6.4 Comparison to related work

Finally, we compare our work against the top-performing hardware-based solutions available in the literature, as reported in Table 5. Among the considered metrics, we focus on the throughput in bitrate, the energy efficiency when available, and the run-time flexibility for replacing the REs. Apart from FlexAmata, the numbers come directly from results reported by the respective authors since other implementations are not open-source, hence not replicable by us, or simulated only, while ours are derived from Section 4.6 best and worst case. The energy efficiency is the ratio of $throughput/power\ consumption$, with bitrate in Gb/s as throughput. Although few approaches (i.e., REAPR and HARE-ASIC) give importance to the energy efficiency for the computation, we claim it is a relevant comparison metric also in this field. Indeed, being FSMs among the most relevant computational kernels in computing fields [42], and although they are intrinsically sequential, providing efficient computations is paramount [14]. Flexibility is a qualitative measurement that refers to the mutability of the matching engine based on the RE(s). We adopted a binary classification based on our understanding of the cited research work. This means that this is not an absolute and quantitative measurement but a criterion that we claim to be relevant. A non-flexible approach means that, given a different (set of) REs, the architecture needs to be re-synthesized or changed accordingly. Indeed, the time to generate a new bitstream can range from one to several hours, depending on the design complexity. Hence, it is often unacceptable for unseen REs and without a database of ready bitstreams.

Considering SDA/IMA approaches, jDFA [11] and ASIC-based HARE [2] achieve the top throughput. Despite this achievement, our approach delivers better energy efficiencies, and the time to adapt to new patterns is in the order of milliseconds against hours of bitstream regeneration. The scaled-down version of HARE, which is effectively deployed on an FPGA, achieves a throughput lower than our PYNQ-Z1 version, not devised for high-throughput scenarios. FlexAmata [17] is an exciting tool based on REAPR [16] for automata computations on FPGA

tested extensively on a broad set of benchmarks. The main focus of FlexAmata is 16-bit symbols, while we target 8-bit ones. Thus, to compare these approaches, we rescaled the bitrate according to their frequency and then doubled it to account for symbols striding, though they do not account for PCIe overhead [17]. Besides, although REAPR reports the power consumption, we exploited the TDP of the VU9P [44] to compute the energy efficiency of FlexAmata, just like we did for our solution. The resulting numbers showcase that, even compared to an SDA approach, our solution reports remarkable throughput and energy efficiency, also providing software programmability. Finally, the work by Meiners et al. [3] is the primary IMA solution that reports noteworthy throughput but no data on power and compilation.

We now move on to what we named SPA solutions, which provide run-time flexibility as their crucial feature. While Brodie et al. [33] and ReCPU [25] ensure the run-time REs change with very low throughput, the Power Edge of Network (or PowerEN) by IBM represents the most interesting related work. It exposes the software programmability on the pattern to search and delivers good throughput, though not reporting the power consumption.

In conclusion, the proposed approach represents a step forward of the SPA approach with throughput close to the top SDA but with the flexibility of changing to unseen patterns at a faster rate than SDA.

7 Conclusions

This work presents a DSA, called TiReX, based on the REs as a program approach. We design an optimized microarchitecture for latency reduction and throughput increment against software references such as FLEX, Grep, RE2, and Hyperscan. The single-core delivers performance in line with the top software with promising energy efficiencies. We provide a multi-core architecture to increase the parallelism level at both the single RE and the multi-RE level while providing an architectural model for different workloads. We show how we reach comparable results with many state-of-the-art hardware-based solutions, providing a high degree of run-time adaptability of the RE. Our testing scenarios show we can achieve a top speedup of 263× in latency and in throughput of ∼179×. Thanks to our domain specialization, we deliver outstanding energy efficiency results that span from 3× to 490× than state-of-the-art software.

References

[1] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan *et al.*, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016.

[2] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[3] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *Proceedings of the 19th USENIX conference on Security*. USENIX Association, 2010.

[4] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, 1968.

[5] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 2001.

[6] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2009, pp. 265–283.

[7] V. S. Chambers, G. Marsico, J. M. Boutell, M. Di Antonio, G. P. Smith, and S. Balasubramanian, "High-throughput sequencing of dna g-quadruplex structures in the human genome," *Nature biotechnology*, vol. 33, no. 8, p. 877, 2015.

[8] H. Ledford, "Personalized cancer vaccines show glimmers of success," Nature international weekly journal of science, 2017.

[9] R. Overbeek, N. Larsen, G. D. Pusch, M. D'Souza, E. S. Jr, N. Kyrpides, M. Fonstein, N. Maltsev, and E. Selkov, "Wit: integrated system for high-throughput genome sequence analysis and metabolic reconstruction," *Nucleic acids research*, 2000.

[10] J. van Lunteren and A. Guanella, "Hardware-accelerated regular expression matching at multiple tens of gb/s," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1737–1745.

[11] L. Jiang, Q. Dai, Q. Tang, J. Tan, and B. Fang, "A fast regular expression matching engine for nids applying prediction scheme," in *Computers and Communication (ISCC), 2014 IEEE Symposium on*. IEEE, 2014.

[12] J. Yang, L. Jiang, Q. Tang, Q. Dai, and J. Tan, "Pidfa: A practical multi-stride regular expression matching engine based on fpga," in *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–7.

[13] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[14] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *Intl. Symp. on Comptr Architctre (ISCA'18)*, 2018.

[15] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[16] T. Xie *et al.*, "Reapr: Reconfigurable engine for automata processing," in *Proc. of FPL*, 2017, pp. 1–8.

[17] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Flexamata: A universal and efficient adaption of applications to spatial automata processing accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 219–234.

[18] C. Xu, S. Chen, J. Su, S.-M. Yiu, and L. C. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 2991–3029, 2016.

[19] K. Atasu, R. Polig, C. Hagleitner, and F. R. Reiss, "Hardware-accelerated regular expression matching for high-throughput text analytics," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–7.

[20] J. Bakker, B. Ng, W. K. Seah, and A. Pekar, "Traffic classification with machine learning in a live network," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 488–493.

[21] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 1083–1100.

[22] VirusTotal, "Yara's documentation," https://yara.readthedocs.io/en/v3.7.0/, 2016.

[23] Z. István, D. Sidler, and G. Alonso, "Runtime parameterizable regular expression operators for databases," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 204–211.

[24] A. Comodi, D. Conficconi, A. Scolari, and M. D. Santambrogio, "Tirex: Tiled regular expression matching architecture," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 131–137.

[25] I. Bonesana, M. Paolieri, and M. D. Santambrogio, "An adaptable fpga-based system for regular expression matching," in *2008 Design, Automation and Test in Europe*. IEEE, 2008, pp. 1262–1267.

[26] R. Cox, "Regular expression matching: the virtual machine approach," *URL: https://swtch.com/ rsc/regexp/regexp2.html*, 2009.

[27] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014.

[28] J. E. Hopcroft, *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[29] M. Nourian, X. Wang, X. Yu, W. Feng, and M. Becchi, "Demystifying automata processing: Gpus, fpgas or micron's ap?" in *Proceedings of the International Conference on Supercomputing*. ACM, 2017.

[30] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the 2007 ACM CoNEXT conference*. ACM, 2007, p. 1.

[31] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: a fast multi-pattern regex matcher for modern cpus," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 631–648.

[32] Q. Tang, L. Jiang, X.-x. Liu, and Q. Dai, "A real-time updatable fpga-based architecture for fast regular expression matching," *Procedia Computer Science*, vol. 31, pp. 852–859, 2014.

[33] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *ACM SIGARCH computer architecture news*, 2006.

[34] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character nfa," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 131–136.

[35] K. Eguro, "Automated dynamic reconfiguration for high-performance regular expression searching," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009.

[36] J. Teubner, L. Woods, and C. Nie, "Skeleton automata for fpgas: reconfiguring without reconstructing," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 229–240.

[37] Y. Kaneta, S. Yoshizawa, S.-i. Minato, H. Arimura, and Y. Miyanaga, "Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching," in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010.

[38] S. G. Singapura, Y.-H. E. Yang, A. Panangadan, T. Nemeth, and V. K. Prasanna, "Fpga based accelerator for pattern matching in yara framework," 2015.

[39] K. Agarwal and R. Polig, "A high-speed and large-scale dictionary matching engine for information extraction systems," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE, 2013.

[40] X.-T. Nguyen, H.-T. Nguyen, K. Inoue, O. Shimojo, and C.-K. Pham, "Highly parallel bitmap-based regular expression matching for text analytics," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–4.

[41] D. Parravicini, D. Conficconi, E. D. Sozzo, C. Pilato, and M. D. Santambrogio, "Cicero: A domain-specific architecture for efficient regular expression matching," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–24, 2021.

[42] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," 2006.

[43] C. J. Sigrist, E. De Castro, P. S. Langendijk-Genevaux, V. Le Saux, A. Bairoch, and N. Hulo, "Prorule: a new database containing functional and structural information on prosite profiles," *Bioinformatics*, vol. 21, no. 21, pp. 4060–4066, 2005.

[44] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 367–381.

[45] D. Paletti, D. Conficconi, and M. D. Santambrogio, "Dovado: An open-source design space exploration framework," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 128–135.

Davide Conficconi got his Ph.D. Information Technology from Politecnico di Milano in 2022. He received his B.Sc. and M.Sc. in Computer Engineering from Politecnico di Milano in 2015 and 2018 respectively. His research interests revolves around reconfigurable architectures, especially FPGAs, design methodologies, computer architectures, design automation techniques, and abstraction layers . He is currently a PostDoc at Politecnico di Milano.

Emanuele Del Sozzo got his Ph.D. in Information Technology from Politecnico di Milano in 2019. He received his B.Sc. and M.Sc. in Computer Engineering from Politecnico di Milano in 2012 and 2015 respectively. He also receives in 2015 M.Sc. degree in Computer Science from the University of Illinois at Chicago (UIC), and Alta Scuola Politecnica Diploma. His research focuses on reconfigurable architectures, code generation and optimization. He is currently a PostDoc at Politecnico di Milano.

Filippo Carloni is a Ph.D. student in Information Technology at Politecnico di Milano. He received his B.Sc. and M.Sc. in Computer Engineering from Politecnico di Milano in 2018 and 2021 respectively. His main research interests are FPGAs, domain specific architectures, hardware design methodologies, computer security, and distributed systems.

Alessandro Comodi is a Software Engineer at Antmicro. He received his B.Sc and M.Sc in Computer Science and Engineering at Politecnico di Milano. His main research interests include FPGAs, Open Source EDA tools, hardware design methodologies.

Alberto Scolari received his Ph.D. from Politecnico di Milano in 2019, where he also received an M.Sc. and a B.Sc. degree. During his Ph.D. research, he focused on optimizing data-intensive applications by making best use of the features of modern CPUs. He currently works as Industrial PostDoc at Huawei's Zurich Research Center, in the Spatial Computing group.

Marco Domenico Santambrogio (SM'05) received the Laurea (M.Sc. equivalent) degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2004, the M.Sc. degree in computer science from The University of Illinois at Chicago, Chicago, IL, USA, in 2005, and the Ph.D. degree in computer engineering from the Politecnico di Milano, in 2008. He was a Post-Doctoral Fellow with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA. He has been with the NECST Laboratory, Politecnico di Milano, where he founded the Dynamic Reconfigurability in Embedded System Design project in 2004 and the CHANGE (self-adaptive computing system) project in 2010. He is an Assistant Professor with the Politecnico di Milano. His current research interests include reconfigurable computing, self-aware and autonomic systems, hardware/software co-design, embedded systems, and high-performance processors and systems.