ALVEARE: a Domain-Specific Framework for Regular Expressions

Filippo Carloni
filippo.carloni@polimi.it
Politecnico di Milano
Milano, Italy

Davide Conficconi
davide.conficconi@polimi.it
Politecnico di Milano
Milano, Italy

Marco D. Santambrogio
marco.santambrogio@polimi.it
Politecnico di Milano
Milano, Italy

ABSTRACT

Regular Expression (RE) matching enables the identification of patterns in datastreams of heterogeneous fields ranging from proteomics to computer security. These scenarios require massive data analysis that, combined with the high data dependency of the REs, leads to long computational times and high energy consumption. Currently, RE engines rely on either (1) flexibility in run-time RE changes and broad operators support impairing performance or (2) fixed high-performing accelerators implementing few simple RE operators. To overcome these limitations, we propose ALVEARE: a hardware-software approach combining a Domain-Specific Language (DSL) with an embedded Domain-Specific Architecture. We exploit REs as a DSL by translating them into flexible executables through our RISC-based Instruction Set Architecture that expresses from simple to advanced primitives. Then, we design a speculation-based microarchitecture to execute real benchmarks efficiently. ALVEARE provides RE-domain flexibility and broad operators' support and achieves up to 34× speedup and 57× energy efficiency improvements against the state-of-the-art RE2 and Bluefield DPU 2 with its RE accelerator.

KEYWORDS

Regular Expressions, Domain-Specific Architecture

ACM Reference Format:
Filippo Carloni, Davide Conficconi, and Marco D. Santambrogio. 2024. ALVEARE: a Domain-Specific Framework for Regular Expressions . In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657378

1 INTRODUCTION

Pattern-matching based on Regular Expressions (REs) finds relevant applications in various scenarios such as text analytics [15], computer security [3, 7, 26], and proteomics [18]. However, these contexts demand massive data analysis, which, combined with the high data dependency of the REs features, can quickly result in long computational times and high energy consumption [15, 16]. While traditional REs represent patterns through alternations and iterations, exceeding simple string matching, modern REs exhibit advanced features demanding more powerful primitives than the simpler ones [21]. For instance, character classes (e.g., `[A-Z]`) require alternations among all the characters while bounded repetitions are transformed into unfolded sequences of concatenations (e.g., `A{3,5}`). Such transformation results in inefficient performance scaling of the matching engines [11], making their management challenging and demanding for advanced primitives.

Currently, the State of the Art split into Software (SW)-based, automata-acceleration-based, and Domain-Specific Architecture (DSA)-based. The first one supports the most advanced operators thanks to the flexibility of CPUs [10, 24]. However, low-latency systems require near-to-data analysis, usually exploiting embedded processors, to avoid costly data transfers to the central system, e.g., Packet Inspection on SmartNICs, excluding mainstream CPUs [8]. Similarly, other approaches save precious CPU cycles through GPU offloading [12], risking missing latency requirements. The second class of approaches embeds automata representations into either in-memory [5, 19] solutions or FPGA-based ones [17]. However, they struggle to handle commonly used advanced RE operators, such as in PCRE standard, or even fail in their support. Lastly, the DSA-based approach combines the benefits of the two previous ones with SW flexibility and Hardware (HW) specialization [2]. Nevertheless, they support a limited number of primitives [4, 14].

Based on these observations, we propose ALVEARE: **a full-stack domain-specific framework** to overcome the state-of-the-art limitations and target constrained and near-data scenarios where mainstream CPUs are either unavailable or saving their cycles is paramount. Using the REs as a Domain-Specific Language (DSL), we design a **novel domain-specialized RISC-based Instruction Set Architecture (ISA)** tailored to compactly express REs most common operators. We then combine the **flexibility of a compilation flow** with the efficiency of HW approaches through a **specialized DSA**. We prototype our DSA on an embedded FPGA and assess its execution time and energy efficiency against state-of-the-art solutions in different scenarios. Overall, ALVEARE achieves up to 34× speedup and 57× energy efficiency improvement against state-of-the-art solutions for constrained and near-data scenarios, while overwhelming GPU-based one with a minimum speedup of 356×. To summarize, we make a three-fold contribution:

- A **RE-tailored** domain-specialized **ISA** that support simple and advanced REs primitives for compact and efficient REs representation (§4);
- A **full-stack HW/SW methodology** relying on a **flexible and optimized compiler** moving the complexity from the hardware to the software, strongly simplifying the hardware design (§5) and on a **specialized microarchitecture** for the efficient execution of REs (§6).
- A **hardware mechanism** to handle speculative counter modes (greedy and lazy) supporting different matching modalities employed in real applications (§6).

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3657378

2 RELATED WORK

Current REs matching solutions aims to combine **efficiency** and **flexibility**. Especially, they split based on the supported *operators*, way to handle *dependencies*, if leveraging *NFA* and *DFA* models, or the employment of *enumerative* and *speculative* approaches. A key feature of these engines is the **execution architecture** and their **software stack** to make them practically usable.

CPU-based solutions generally present the higher degree of flexibility [10] and try to squeeze out the performance from the architectural characteristics [24] or exploit algorithmic improvements for dynamic adaptation among speculation or enumeration [15, 16]. Differently, **GPU-based solutions** exploit the SIMD/SIMT capabilities for multi-FSM and multi-striding [12], but the *embarrassingly sequential* [15] nature of this kernel highly limits their adoption [13].

Other solutions exploit **FPGAs** [17], **ASICs** [9], or **In-Memory** [5, 19, 20] **accelerators** to directly embed their NFA formulation in the reconfigurable fabric of the given architecture. Among the solutions for logic embedding, FPGAs represent the commercially available platform with higher potential [13]. However, these approaches suffer the lack of advanced primitives [11] and the long reconfigurability issue as for other In-Memory architectures [13, 19, 25].

To fill the gap among flexible and highly specialized solutions **software-programmable DSAs** are a promising approach that transforms patterns in sequences of instructions to execute [2, 4, 6, 14, 22]. However, complex REs primitives are poorly considered, and the main target scenario are datacenters. Among them, the NVIDIA Bluefield DPU 2 [2] is a SmartNIC with a RE accelerator to offload the pattern matching task that avoid CPU cycles burning.

Our methodology falls in these RE-centric architectures. However, our DSA combines a full-stack HW/SW approach to bring REs as first-class citizens covering a wide range of operators.

3 ALVEARE RE-CENTRIC FRAMEWORK

Our framework[1] leverages a flexible RISC-based ISA that decomposes REs in a set of primitives (§4) to support the widely adopted standards POSIX ERE and PCRE. The ISA breaks down into primitives performed intra-characters (e.g., concatenation or alternation) and among sub-RE (e.g., counters) accounting for dependencies and parallelization opportunities. The software-centric component of ALVEARE framework is the DSL compilation flow (§5). Its flexibility allows lifting part of the REs complexity towards the compiler to simplify the ISA and microarchitecture designs. Finally, the microarchitecture performs vector RE matching on the stream of data, while a *tailored controller* handles the data dependency and control issue through a depth-first-like speculation approach with a backtracking mechanism for *mispredictions* (§6).

4 RE-TAILORED ISA

The ISA relies on an operators-composition strategy similar to a VLIW approach to build advanced instructions, but with a fixed size. As Figure 1 illustrates, each instruction is 43 bits wide and breaks down into three fields: *opcode*, *enabling character*, and *reference*.

Bits 42 to 36 encode the operation; an operator is enabled when the corresponding bit is set to "1". The default operation among instructions is the AND, meaning that the match process is complete

[1] https://github.com/necst/alveare

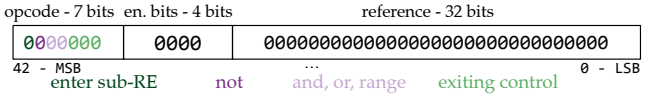

Figure 1: Our 43-bit instruction format breaks into three fields: 7-bit for the opcode (Table 1), 4 bits to validate the following 4 bytes, and 32 bits for the reference. E.g., `([^A-Z])+` translates second and third field of the second instruction as "`1100`" and "`01000001 01011010`" (the other bits are zeros).

Table 1: ALVEARE operation classes. Some of these operations are composable intra-class (e.g., the `NOT`) or inter-class (e.g., `)` or `QUANT` with base), via the *don't care* terms (`-`). E.g., `([^A-Z])+` translates "`1000000`", " `0111010`", and "`0000000`".

Class	Operator	Opcode	Description
Control	EoR	0000000	End of RE
Base	AND	0010---	Char-based And
	OR	0001---	Char-based Or
	RANGE	0011---	Char-based Range
	NOT	01-----	Match Inversion
Complex	(	1000000	New Sub-RE
	)	0---100	End of Sub-RE
	QUANT L	0---001	) + Lazy Quantifier
	QUANT	0---010	) + Greedy Quantifier
	)|	0---011	) + OR of Sub-RE

when all the instructions of the RE have a *sub-match*. Notably, to modify this behavior, we use advanced operators such as complex OR or quantified repetitions to force the execution flow to skip or repeat instructions. The bits from 35 to 32 are reference-enabling bits, *base operators* use them to identify whether a character in the reference is valid by setting them to "1". Specifically, the reference field encodes sequentially the 8-bit characters pattern for base operators, or it encodes execution information such as the number of repetitions or the instructions relative addresses to jump for the complex operators. Given their sequential encoding, the enabling bits are "`0`" ended (e.g., `A|B` is represented as an OR operation with "`1100`" as enabling bits and "`AB`" as reference). Such reference-enabling bits are essential in binary-based pattern-matching applications [3], where we also need not human readable ASCII values (e.g., `\x00`).

Table 1 shows the three operator classes in our ISA: *control*, *base*, and *complex*, which handle the execution end, the intra-characters operations, and intra-REs operations, respectively. Notabily, the ISA allows operators from different classes to be active in the same instruction if and only if only one of them uses the reference part.

ISA Simple REs: Control and Base Operators. The *control operator* instruction class comprises only the End of RE (EoR) operator, indicating the end of the RE matching process.

The *base operator* class represents all the basic intra-characters operations with their execution primitives. These operators have a one-to-one correspondence with REs DSL and can be used only between strings of characters. While the AND and OR are relatively intuitive, the RANGE represents the first optimization step. In principle, a RANGE operation is a sequence of OR alternations among

a set of characters, reporting a match if one of them has a correspondence in the analyzed data. For instance, all the lowercase alphabetical characters `[a-z]` are also representable as `a OR b ... OR z`. However, such a big chain of alternations implies many instructions. Thus, we design an ISA `RANGE` operation primitive to heavily reduce the instruction count and the execution cycles (see §7.1). The `NOT` is another advanced primitive that exploits the ISA's composable structure and thus negates alternation operations (i.e., `OR` and `RANGE`) for PCRE and POSIX standards (Table 1).

The base operator class encodes the characters to match into the 32-bits of the reference field for at most four 8-bit vectorized characters. While the `AND` operator requires a correspondence among instruction and data characters, the `OR` requires a single character match. Similarly, the `RANGE` matches if the processed data contains a character within the start and the end of at least one of the two `RANGE`s. The ISA enables the packing of `[a-z0-9]` in a single `RANGE`.

ISA Advanced REs: Complex Operators. The *entering sub-RE operator*, indicated as `(`, represents the entry point for a sub-pattern in the RE flow. Figure 2 shows the structure of its reference to set flags and information, hinting at the sub-RE execution type. The first five bits enable a valid reference subfield containing other helpful execution information. The two leftmost bits enable the counting repetition limits, i.e., a possible minimum and maximum, to express bounded quantifiers or Kleene's operators. We optimize the ISA design with a single counter primitive to be bounded or not in the sub-RE repetitions. We encode an infinite upper bound with all the bits set to "1". Thus, as the minimum and maximum counters encode with 6 bits, the supported range of the bounded counter is between 0 and 62 (2^6 - 2), and 63 is reserved for the unbounded.

The following 2 bits in the complex operator field (the 3^{rd} and 4^{th}) indicate whether the backward and forward addresses are valid (Figure 2). Specifically, the execution flow of an RE is extremely data-dependent; therefore, we might shift from the sequential execution flow and jump forward or backward in our RE based on the matching outcome. Hence, these backward and forward addresses specify the relative jumps to the address of the *entering sub-RE operator*, limiting the code size and improving the performance. The last of the five enabling bits sets and anticipates if the sub-RE execution is *greedy* or *lazy* (by setting the bit to "1").

The remaining operators of the complex class conclude the sub-RE execution in three different ways. The *quantification* repeats the sub-RE instructions zero or more times based on the specification at the entrance of the sub-RE. The execution flow jumps backward, thus repeating the sub-RE, or forward, hence continuing with the next operation. Notably, until the minimum number of repetitions is not satisfied, the execution can only jump backward. Instead, if the minimum limit is reached, the greedy or lazy mode impacts the execution. Specifically, the greedy mode repeats the sub-RE, thus jumping backward as many times as possible until a mismatch occurs. Differently, the lazy tries to move to the operation that follows the quantified sub-RE, repeating the sub-RE match only in case of a mismatch for the following operator. The *alternation of sub-REs* (i.e., `)|`) jumps to the end of the alternation chain (forward jump) if the analyzed sub-pattern has a match. In contrast, if the sub-pattern has a mismatch, it jumps to the following sub-RE in the alternation chain (backward jump). Finally, the *simple termination* of a sub-RE notifies a sub-pattern conclusion. Since all the *closing complex* operators do not use the reference, they are combined with the base operators to reduce the instructions count.

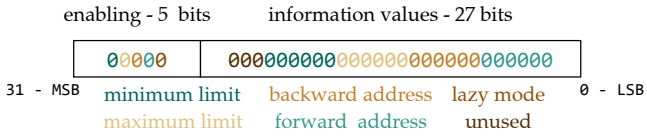

Figure 2: Reference detail for the enter sub-RE operator. Five bits are enabler bits for the following 27-bit validity of the respective subfields or to enable lazy or greed matching. The 27 bits encode the counter limits and the backward and forward relative jumps, while the 3 MSBs are currently unused. E.g., `([^A-Z])+` translates the first instruction as "11110" "00000000111111100000000010".

5 COMPILATION FLOW

Our compiler comprises the three traditional stages that transform a RE downto the DSA executable: front-end, mid-end, and back-end.

Front-End: Lexical and Syntax Analysis. The front-end takes the input REs and checks their lexical and syntactic compliance. The lexer employs *FLEX* tool and provides specific rules for tokenizing the supported RE operators. Then, we build a grammar of supported REs via BISON and let the compiler check the RE correctness after the tokenization, generating the corresponding *Abstract-Syntax-Tree (AST)*, an optimizable high-level syntactic structure. Currently supported POSIX and PCRE operators are: characters alternation and concatenation; character classes (`[abc]`), ranges (`[a-z]`), their negation (`[^abc]`), and their short end (e.g., `\w`); the any character except `\n` (`.`); the bounded (`?,{n},{n,m}`) and unbounded (`*,+,{n,}`) quantifiers with their lazy options (e.g., `{n,}?`); the escaping of characters with `\`.

Middle-End: Lowering and Optimizing the REs. The middle-end transforms the corresponding AST to an ISA-oriented representation. Then, it removes over-parenthesized sub-RE, assuming the ISA default `AND` operation among two consecutive instructions.

Although the ISA primitives contain at most four characters in the reference (limiting long string matching operations) the implicit `AND` among consecutive instruction lets us overcome this limitation. Specifically, it implies that two or more derived groups of `AND` operations behave as a bigger `AND` comprising all the characters. Similarly, we group `OR` and `RANGE` expressions by four characters as for the ISA limit. Differently from the `AND` case and supposing the RE exceeds this limits, the compiler chain them into a *complex* `OR` chain with subgroups of *base* `OR` with a maximum of four operands per node. This approach supports an indefinite number of characters in a single base expression, overcoming the ISA-limited reference. As the ISA offer a single quantifier primitive, the compiler transform Kleene operators (i.e., `*` and `+`) into the equivalent unbounded representation (i.e., `{1,}` and `{0,}`). Likewise, to reduce the primitives required, the compiler translate other ISA-unsupported primitives (e.g., `\d` or `.`) into a sequence of supported ones. For instance, the `\w` is translated in its equivalent representation `[a-zA-Z0-9_]`, or the `.` translates into `[^\n]`.

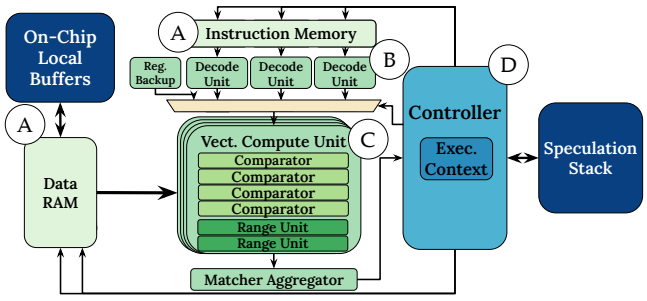

Figure 3: Single-core μ-architecture: (A) the *memories* for instructions and data, (B) the *decode units* prefetch the instructions for any change in the control flow, (C) the *vectorized compute unit* and a *matcher aggregator* for high-performance *simple operators*, (D) the *controller* and the *speculation stack* deal with *complex operators*, handle the matching procedure, and feature speculation approach to alternative paths.

These transformations output an *Intermediate Representation (IR)* which defines a compact ISA-like version of the REs. After the latest optimized syntax tree step, the compiler decorates the IR with relative jumps calculated on the exact instructions number. The IR traversal strategy follows the depth-first strategy, assigning the address to the leaves of a node before moving to the adjacent one.

Back-End: Operation Fusion and Binary Code. Finally, the back-end translates the resulting IR into the executable binary and, whenever possible, applies further architectural-aware optimizations allowed by the ISA, such as merging a closing sub-RE operator (e.g., `)|`) with a base one (e.g., `AND`), creating a single instruction. This fusion is possible because base operators leverage the reference to encode the characters to scan, while the complex closing operators do not use this field §4. However, when two consequent closing operators are used in the RE, only the one nearest to the base operator is merged with it. In contrast, the outermost closing one requires additional instructions. The final output is a binary loadable into ALVEARE instruction memory and then executed.

6 ARCHITECTURAL ORGANIZATION

The RE-tailored DSA is optimized on top of the proposed ISA (§4). It executes a compiled RE (§5) and analyzes a data stream, producing the matching results. Figure 3 illustrates the architectural organization and its fundamental components.

Feeding the RE Matching Process via Memories (A). The architecture embeds two distinct memories for instructions and data to analyze (Figure 3 (A)). Specifically, the instruction memory stores the compiled binary code. This memory receives three addresses simultaneously and handles all the possible instruction flows in a RE: *sequential*, *backward*, and *forward* jumps. Thus, the DSA prefetches these instructions for the next cycle and decides which one to execute. In this way, each instruction completes at each cycle, following the RISC-based ISA principle.

Conversely, the data memory stores the data to analyze in a 2-level memory hierarchy to handle the access pattern. The on-chip local buffer stores the entire data chunk to analyze and continuously feeds the data to a smaller RAM until the data stream to explore is consumed. Specifically, the smaller RAM interacts directly with the execution core and handles two addresses. The first address targets the latest sub-match address. We exploit it whenever a mismatch occurs, and the execution has to restart from the last match of the small RAM. Instead, the second address refers to the current pointer in the data stream. For this address, the data memory outputs the necessary number of chars to execute a base operator in the next cycle, compatibly with the number of vectorial compute units.

Preparing the Instructions: Decode Unit (B). Given that RE matching is highly data dependent and can choose an execution flow based on the sub-matching outcome, ALVEARE has three decode units that process the three instructions coming from the instruction memory at each cycle (Figure 3 (B)). Given the three prefetched instructions these units transform the 43 bits instruction into three different outputs: the reference, a signal to indicate which of the data in the reference are valid (if a base operation), and the one-hot decoded opcode. Then, the controller selects which prefetched instruction will pass to the following stage according to the matching outcome and on the operation. Additionally, we save the first instruction decoded in a backup register that is essential in case of a complete sub-matching failure to restart the RE procedure.

High-Performance Matching: Execution (C). Once the decoded instruction and the data stream are ready for analysis, the execution stage begins. This stage, building on top of a *vectorial unit* and an *aggregator unit* (Figure 3 (C)), processes the base ISA primitives (§4). On the one hand, the vectorial unit embeds four compute units (CUs), each of which includes four comparators that parallelly process the `AND`, the `OR`, and two `RANGE` units. Each additional CU implies an additional character analyzed in an overlapping fashion such that we find all the possible matches in the data stream. For instance, having four CUs means a maximum of analyzed data of seven characters (i.e., 4 chars for each CU with an overlapping of 3 chars with the adjacent: $\#_{Comparators} + 1 \cdot (\#_{Compute\_Units} - 1)$).

On the other hand, the aggregator unit has a local view of the execution and coordinates the CUs. Specifically, it reorders the single operator results for each CU and raises a match in the following cases: if at least one of the comparators has a match for the `OR`; if one of the two range units has a match for the `RANGE`; if all the comparators of the CU have a match for the `AND`. Considering that the `NOT` operator is composable with the `OR` and `RANGE`, this unit inverts the overall results if needed.

ALVEARE Core: Controller and Speculation (D). The controller is at the core of the microarchitecture (Figure 3 (D)). It is a centralized control unit responsible for executing the complex operators of the ISA and managing the overall RE execution, which is highly data-dependent and embarrassingly sequential. To guarantee a correct execution flow, the controller must select the data to analyze and the instruction to execute among the prefetched ones. The expected flow for REs retrieves the data and executes the instructions sequentially. However, the complex operators, such as quantifiers or sub-REs alternation, change the data and instructions flows depending on sub-match occurrences. Moreover, to fully support the quantifiers modalities employed by the widespread PCRE standard, we designed the controller with a speculative approach with two modalities (i.e., greedy and lazy). The speculation allows the flow alteration, forcing the repetitions of the sub-pattern or

Table 2: ALVEARE ISA advanced primitives improvements.

Microbench. RE	Minimal Primitives	Advanced Primitives	Code/Clock Cycles Reduction
`[a-zA-Z]`	26	1	26×
`[DBEZX]{7}`	28	6	4.66×
`.{3,6}`	1160	2	580×
`[^ ]*`	66	2	33×

moving on to the next one depending on the modality and repetition requirements. Indeed, once entered into a sub-RE, the controller repeats the pattern as much as possible in the greedy mode (it speculates on the re-match of the sub-RE) or the least in the lazy one (it speculates on the match after the sub-RE), always satisfying the minimum number of repetitions.

To perform speculation, the controller exploits an additional stack memory structure to keep track of the alternative paths in the execution flow if a mismatch occurs during the speculation. Whenever a complex opening operator is encountered, the controller pushes the execution status information to the stack. In this way, we save a snapshot of the essential information to recover from the speculation in a rollback situation. The snapshot includes the quantification bounds (minimum and maximum), the current matches number of a quantifier, the possible sub-matching state of the architecture, the latest matched character position, and the current address in the data stream (i.e., before entering in the sub-pattern).

Scaling Out to a Multi-Core. We design a scale-out version replicating independent ALVEARE cores with private local memories. Such multi-core architecture operates on the same RE and searches for that pattern on different data stream portions. In this way, we can parallelize at the stream of data level and search for the pattern more effectively through a divide-and-conquer approach. The constraint for such scale-out is the area devoted to the multi-core (§7.2). Indeed, the data and instruction (i.e., the REs) memories are decoupled into two distinct private memories.

7 EVALUATION

We evaluate the benefits of adopting the advanced ISA primitives (§7.1) and ALVEARE against state-of-the-art solutions in embedded, near-data, and offloading scenarios (§7.2).

7.1 ALVEARE ISA Reduces Code and Cycles

We select representative RE microbenchmarks from [23] focusing on operators beyond the minimal set of regular language and widely employed by the standards. We compare these REs compiled with their minimal representation (i.e., performing compiler-based unfolding) and with our advanced ISA primitives. Table 2 shows the reduction of code size (excluding the `EoR`) and, being RISC-based, the cycles required to execute the whole set of instructions.

Although in the first RE our ISA shrinks `a|...|z|A|...|Z` alternations into 26 primitives thanks to the vectorization, it also offers the `RANGE` to express them into a single operation compactly. The second RE requires the concatenation of seven explicit `(D|B|E|Z|X)` repetitions, while our ISA compresses the quantification via the homonymous primitive with a 4.66× reduction. The third RE raises

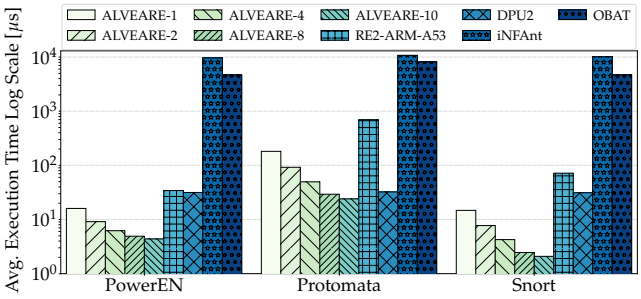

Figure 4: Execution time, the lower, the better (log scale). Top speedup of 7× (A53), 15× (DPU), 356× (GPU).

the complexity since the `.` advanced operation translates in all the ASCII (128 chars in this case) but the "\n" repeated between 3 and 6 times. Nonetheless, we can reduce its cost by expressing it as `[^\n]` along the quantification primitive, with a 580× gain. Finally, the last RE expresses as before all the characters (but the space one) repeated zero or more times, which we represent with the `GREEDY QUANTIFICATION` of a `NOT RANGE`, for a 33× reduction.

7.2 Setup and Literature Comparison

We implement our DSA in VHDL on an *embedded* Ultra96v2 board (AMD Zynq XCZU3EGA484), generate the bitstream with Vivado 2019.2, run it at 300 MHz, and exploit PYNQ 2.7 framework [1] for Host-DSA communications. We scale out from one to ten cores, as it is the maximum number fitting our FPGA resources due to the linear BRAM scaling (6.71% to 67.13%) and the sublinear LUT scaling (11.39% to 84.65%), and measure their impact.

State of the Art solutions. We focus the analysis on scenarios where CPUs have limited computing and power capabilities, e.g., embedded systems, or processing near to data sources is paramount to reduce the data transfer overhead, e.g., network-related, requiring data offloading is mandatory to alleviate data-intensive burdens. Within this context, we select Google *RE2* [10], which is a cross-platform memory-safe approach to REs, and run it on the Ultra96 ARM A53 with *-O3* optimizations, as embedded CPU solution. Then, we select the NVIDIA Bluefield 2 *DPU* [2], a commercially available system comprising an accelerator for REs as near-data solution. Finally, to represent offloading scenarios, we run on an NVIDIA V100 the first GPU-based algorithm, *iNFAnt*, and the State of the Art on GPUs, *OBAT* with *hotstart* optimization [12].

Benchmarks. We evaluate three representative benchmarks of ANMLZoo [23]. PowerEn, a synthetic benchmark from IBM for the evaluation of the Power Edge of Network SoC [22]. Protomata for research of protein patterns and one of the most complex benchmark in the suite [18]. Finally, Snort [3] a production-ready Deep Packet Inspection from CISCO. We chose the 1MB dataset and 200 REs randomly selected after excluding bad-formed REs. For fairness, we consider the DPU memory limits of 16KB input chunks.

Key Performance Indicators (KPI). We consider for each benchmark the average execution time and energy efficiency. For each RE executed, we assess the matching time after memories loading and repeat the experiments ten times to improve the stability. Considering the power, we use the V100 thermal design

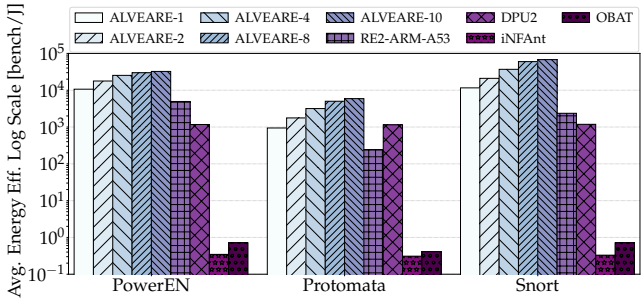

Figure 5: Energy efficiency, the higher, the better (log scale). Top gains: 29× (A53), 57× (DPU), orders of magnitude (GPU).

power, given lack of physical access, and for the A53, the DPU, and ALVEARE, we instrument via a Voltcraft 4000 measuring chip, on-board DRAM, and other parts of the system. The whole Ultra96 board with a 10-core ALVEARE consumes 7.05W, against the 27W of the DPU board and the 5.9W for the A53. We calculate the energy efficiency per benchmark as $Energy\_Eff_{avg} = \frac{1}{Exe\_Time_{avg} \cdot Power_{avg}}$.

Execution Time. Figure 4 shows that the 10-core ALVEARE achieves the best improvements against the single-core with a speedup of 3× for the synthetic PowerEN, while on real-life benchmarks (i.e., Protomata and Snort), around 7×. Nevertheless, the single-core is more than enough to perform better than RE2 in all the benchmarks with speedups from 2.13× to 4.86×, while the 10-core from 7.83× to 34.68×. Instead, the DPU features a divide-and-conquer approach via multi-threaded hardware to improve the performance. However, ALVEARE 10-core outperforms it in all the benchmarks with a peak speedup of 15.11× for real-world Snort. Conversely, GPU-based algorithms exhibit high processing times, at least two orders of magnitude higher than the others, hindering their usage in real scenarios. Indeed, our 10-core achieves a minimum speedup of 356× over OBAT on Protomata.

Energy Efficiency. Figure 5 illustrates the resulting efficiency of the whole systems. ALVEARE single core can, at worst, be in line with the State of the Art or improve the computation efficiency of a 9.78× on Snort. The top-performing ALVEARE 10-core architecture always delivers energy efficiency improvements, with peaks of 57.88× and 29× against the DPU and the A53. Finally, the single-core overwhelms GPU solutions by four order of magnitude.

Overall, ALVEARE demonstrates remarkable performance and energy efficiency improvements that make the proposed solution highly attractive for constrained and near-data scenarios, where offloading tasks such as RE matching is paramount.

8 CONCLUSION

This paper presents a comprehensive HW/SW framework called ALVEARE. We devise a novel ISA to support advanced RE primitives widely adopted by POSIX and PCRE standards while combining a compilation flow along a domain-specialized microarchitecture to simplify their design and balancing the efficiency-flexibility trade-off. Our evaluation demonstrates top speedup of 34× and *efficiency boost* of 57× against RE2-A53 and DPU (§7.2), representing a viable option in embedded or near-data scenarios.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, L. Cicolini, E. D'Arnese, E. Del Sozzo, G. Sorrentino, and G. Antichi for precious feedback, and AMD University program, NVIDIA Academic HW Grant Program, Oracle Cloud Infrastructure and Oracle for Research for support.

REFERENCES

[1] AMD. 2021. PYNQ v2.7. https://github.com/Xilinx/PYNQ/tree/image_v2.7.
[2] Idan Burstein. 2021. Nvidia Data Center Processing Unit (DPU) Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–20.
[3] Cisco. 2023. Snort Intrusion Prevention System (IPS). https://www.snort.org/.
[4] Davide Conficconi, Emanuele Del Sozzo, Filippo Carloni, Alessandro Comodi, Alberto Scolari, and Marco Domenico Santambrogio. 2023. An Energy-Efficient Domain-Specific Architecture for Regular Expressions. *IEEE Transactions on Emerging Topics in Computing* (2023).
[5] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* (2014).
[6] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*.
[7] Domenico Ficara, Andrea Di Pietro, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, and Gianni Antichi. 2010. Differential encoding of DFAs for fast regular expression matching. *IEEE/ACM Transactions On Networking* (2010).
[8] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, et al. 2018. Azure Accelerated Networking:SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.
[9] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[10] Google. 2020. Google re2. https://github.com/google/re2.
[11] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-hardware codesign for efficient in-memory regular pattern matching. In *ACM PLDI*.
[12] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why gpus are slow at executing NFAs and how to make them faster. In *ACM ASPLOS*.
[13] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP?. In *Proceedings of the International Conference on Supercomputing*. ACM.
[14] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, et al. 2021. CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching. *ACM Transactions on Embedded Computing Systems (TECS)* (2021).
[15] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. 2020. Challenging sequential bitstream processing via principled bitwise speculation. In *ACM ASPLOS*.
[16] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. 2021. Scalable FSM parallelization via path fusion and higher-order speculation. In *ACM ASPLOS*.
[17] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 2020. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines*.
[18] Indranil Roy and Srinivas Aluru. 2014. Finding motifs in biological sequences using the micron automata processor. In *IEEE IPDPS*.
[19] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *International symposium on HPCA*.
[20] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.
[21] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020).
[22] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a programmable wire-speed regular-expression matching accelerator. In *International Symposium Microarchitecture*.
[23] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, et al. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *IEEE International Symposium on Workload Characterization*.
[24] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: a fast multi-pattern regex matcher for modern cpus. In *USENIX Symposium on NSDI*.
[25] Jackson Woodruff and Michael FP O'Boyle. 2021. New Regular Expressions on Old Accelerators. In *58th ACM/IEEE Design Automation Conference (DAC)*.
[26] Chengcheng Xu, Shuhui Chen, Jinshu Su, et al. 2016. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials* (2016).