Combining MLIR Dialects with Domain-Specific Architecture for Efficient Regular Expression Matching

Andrea Somaini
andrea.somaini@mail.polimi.it
Politecnico di Milano
Milan, Italy

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

Giovanni Agosta
giovanni.agosta@polimi.it
Politecnico di Milano
Milan, Italy

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

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

Abstract

Pattern matching based on Regular Expressions (REs) is a pervasive and challenging computational kernel used in several applications to identify critical information in a data stream. Due to the sequential data dependency of REs and the increasing data volume growth, hardware acceleration is gaining attention to address the limitation of general-purpose architectures. RE-oriented Domain-Specific Architectures (DSAs) combine the flexibility of translating REs into binary code with the efficiency of a specialized architecture, filling the gap between frozen hardware accelerators and the versatility of CPUs/GPUs. However, existing DSAs focus mainly on the efficiency execution challenge while missing the optimization opportunities that a structured compilation infrastructure can provide. This paper proposes a RE-tailored multi-level intermediate representation strategy embodied by the MLIR framework at the compiler level to exploit different abstraction optimizations via two domain-specific dialects, one targeting the abstract representation of REs and the other targeting the underlying domain-specific ISA. Moreover, this paper proposes a novel architectural organization of an open-source state-of-the-art DSA to maximize the parallelization capabilities. Overall, the proposed approach significantly improves execution time by up to 2.26×, energy efficiency by up to 2.30×, and resource usage.

CCS Concepts: **• Hardware → High-level and register-transfer level synthesis**; **• Software and its engineering → Compilers**; **• Computer systems organization →** *Multicore architectures*; Embedded systems.

Keywords: Domain-Specific Architecture, Domain-Specific Compiler, Parallel Enumeration, Regular Expressions, MLIR

CGO '25, March 01–05, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708916

ACM Reference Format:
Andrea Somaini, Filippo Carloni, Giovanni Agosta, Marco D. Santambrogio, and Davide Conficconi. 2025. Combining MLIR Dialects with Domain-Specific Architecture for Efficient Regular Expression Matching. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 17 pages.
https://doi.org/10.1145/3696443.3708916

1 Introduction

Thanks to their concise and expressive patterns, Regular Expressions (REs) are a powerful tool for many applications comprising natural language processing, genomics, databases, and deep-packet inspection [5, 10, 11, 26, 28, 36, 46, 53, 54, 56, 61, 62]. The straightforward approach to execute REs exploits their equivalent recognizer representation called Finite-State Automata (FSA) that can be divided into deterministic (DFAs) and non-deterministic (NFAs), depending on how many outgoing transitions a state has with the same input character [26]. DFAs are simple to execute since there is only one possible choice based on the input character, but they could quickly lead to exponentially blowing up the number of states [2–5, 9, 14, 30]. Differently, NFAs can have a compact representation but demand higher computational capability to handle the simultaneous parallel paths to activate [3, 16, 26, 32, 49, 50].

Existing solutions rely on different architectural paradigms to overcome the REs execution challenges, spreading among CPU- [22, 41, 42, 56, 63], FPGA- [35, 43, 45, 55, 58, 60, 61], in-memory- [15, 16, 20, 45, 48], GPU- [8, 19, 32, 33, 64], and Domain-Specific Architectures (DSAs)-based [6, 13, 17, 17, 21, 39, 59]. Notably, the DSA approach has demonstrated a better tradeoff between flexibility and execution efficiency by exploiting the paradigm of RE-as-programming language [14, 50], keeping high flexibility thanks to software (SW) abstractions while delivering domain specialization and high efficiency via hardware (HW) acceleration [13, 17, 21, 37, 39, 52, 59]. Specifically, DSAs leverage domain-specific compilers to translate the REs into executable code following the

RE-tailored ISA specifications. With this language abstraction, it is possible to recompile REs quickly and load them on the execution engine, thus providing HW specialization efficiency without paying the cost of significant overhead for HW reprogramming [35]. In this context, Cicero [39] is an open-source DSA for REs based on Thompson's algorithm [50], which showcased remarkable energy efficiency improvements over embedded and mainstream CPUs.

However, although this paradigm has demonstrated interesting benefits, current DSA-oriented solutions for REs [7, 13, 39, 59] lack a structured compilation flow to push the coding efficiency behind simple translators and an advanced degree of parallelism to fulfill the most demanding performance requirements. Indeed, current DSA compilers suffer from *premature lowering* [31, 34], which causes abstraction removal in the first stages of the compilation flow, potentially missing powerful high-level optimizations that enormously improve the generated code performance. Instead, exploiting a tight HW-SW synergy that combines multi-pass optimizations of Multi-Level Intermediate Representation (MLIR) [31] dialects with architectural enhancements pushes for new high-performance RE-tailored solutions.

This work proposes a multi-dialect MLIR-based compiler to leverage different abstraction semantics and allow multi-pass optimizations for the RE scenario, addressing the current limitations of the RE-based DSA compilers. We combine the general high-level *RE-tailored dialect* with the proposed *low-level MLIR dialect* for Cicero ISA to provide seamless integration at different abstraction levels and simplify the lowering process while ensuring RE efficient execution on the final architecture. The proposed compilation flow leverages the high-level Intermediate Representation (IR) for general and algebraic optimizations while removing the lower-level ones of the previous compiler. Instead, the low-level dialect facilitates the implementation of Cicero architecture-specific optimizations that cannot be implemented on high-level IR. This novel multi-level abstraction improves the execution times of the original Cicero up to 1.7×.

Then, we propose an improved architectural organization of the Cicero engine to foster the RE matching parallelization and further take advantage of the compiler optimizations. Our optimized organization increases the enumeration capabilities by adding more cores in the single Cicero engine. In this way, we improve the parallelization efficiency by removing the need to distribute threads across engines to balance the thread count among them, achieving a more streamlined and efficient approach. This denser packing of computing cores better exploits the HW resources and, combined with the multi-level abstractions, addresses the code locality issue, showcasing latency and energy efficiency improvements.

In summary, this work has a three-fold contribution:

- A novel compilation infrastructure based on **RE-centric multi-level custom MLIR dialects**, allowing for **multipass high- and low-level optimizations** and targeting the Cicero ISA as the back-end (§3).
- An **enhanced multicore design** of the Cicero engine architecture by **providing parallelism** with **minimizing thread movements** (§4).
- An **architecture-aware approach** that exploits the benefits of multi-level compiler abstractions to improve instruction layout for a higher code locality and enhanced microarchitectural performance (§5).

Thanks to our approach[1] [47], we improve the execution time up to 2.26× and energy efficiency up to 2.30× (§6).

2 Motivation

Expressing the enumeration capabilities of NFAs has been illustrated by the Thompson construction algorithm [1, 50], and further explored by R. Cox [14] with the Google RE2 principles [22]. Cicero's end-to-end domain-specific framework embodies these principles for the efficient execution of REs [39] as it comprises: a compiler for translating an RE into an executable binary, an ISA that exposes the basic primitives, and a configurable domain-specific architecture. Its foundations are driven by the concept of decomposing REs in simple operations and enumerate all the possible parallel execution *threads* working in lockstep on a sequence of characters [14]. Cicero ISA operation classes are three: *matching*, *control flow*, and *acceptance*. The first one expresses the scenarios of matching any character, a specific one, or not matching a specific one. The control flow operations create two execution threads with the split instruction to express alternative paths to explore or unconditional control flow changes. The last class represents the conclusion of an RE execution in the exact matching scenario (i.e., the whole text is matched) or the more classical one of finding a match at any point in the input stream. We summarize Cicero instructions in Table 1.

Table 1. Cicero Instruction Set Architecture. `PC` is Program Counter (i.e., the address of the next instruction to be executed), `cc` is the pointer to the current character, and `OP` is the instruction operand.

Class	Instruction	Operand	Description
Matching	`MatchAny`	-	`PC+1` and `cc+1`.
	`Match(OP)`	Character	if `OP == *cc`, `PC+1` and `cc+1`.
	`NoMatch(OP)`	Character	if `OP != *cc`, `PC+1`.
Control Flow	`Split(OP)`	Target Addr.	Produces two flows `PC+1` and address `OP`.
	`Jump(OP)`	Target Addr.	Unconditional jump to address `OP`.
Acceptance	`Accept`	-	Accepts if at the end of the string.
	`AcceptPartial`	-	Accepts at any point in the string.

2.1 Premature Lowering Issues

Cicero features a domain-specialized compiler that automatically translates from RE to a binary compatible with its ISA.

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

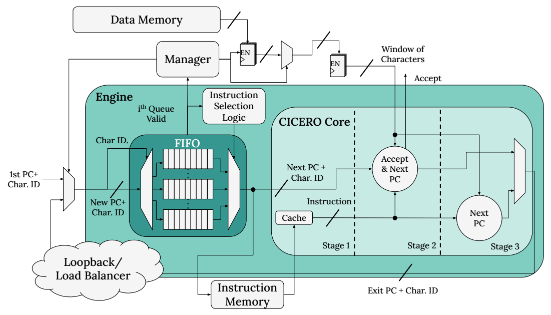


Figure 1. Single-engine Cicero: the core emulates the enumeration over a window of characters through a set of FIFOs and a time-multiplexed three-stage pipelined core.

It is completely built from scratch as its instructions are stateless, and the operation mode differs from a general-purpose processor, making it impossible to rely on traditional CPU-based compiler optimizations. However, the current compilation approach misses important high-level optimizations due to prematurely lowering the REs to the target ISA. In fact, the old compiler features a single level of IR, which specifies the order of instructions and the control flow between basic blocks just after parsing. Given this single layer of IR, the old compiler performs at this point the mapping of basic blocks to instruction memory and generation of control instructions, and optimizations occur on mapped IR. Although the old compiler can, in the first instance, optimally map basic blocks and generate control instructions, it later needs to apply optimizations to improve the code performance, potentially raising sub-optimal implementations due to the required control flow restructuring as shown in §5.

Based on this observation, and given the peculiarities of REs, a dedicated multi-level IR avoids lowering prematurely the IR to an architecture-specific one [31, 34]. To build the proposed RE-centric compiler infrastructure, we take advantage of the MLIR [31] approach since it allows us to provide a high-level intermediate language for REs while at the same time abstracting the proposed hardware architecture (§3). We introduce a high-level dialect for REs tailored for abstract optimizations and then lower to an ISA-oriented representation where a dedicated dialect can now apply an optimal mapping of basic blocks, devising a different compilation infrastructure, as Figure 2 shows.

2.2 Enumeration Limitations

Cicero's main architectural component is the engine, a unit capable of enumerating the possible execution threads of an NFA over a window of character in a lockstep fashion. Figure 1 illustrates the main components of an engine: the `core` and the First-In-First-Out (`FIFO`) buffers. The FIFOs contain the Program Counters (`PCs`) associated with one character of the window (`CC_ID`) to enumerate execution threads. The core is a *time-multiplexed* computational unit that retrieves the instruction, thanks to the `PC`, and processes it through a three-stage pipeline. Each core features its instruction cache, which pays the cost of reading from the central instruction memory in the case of cache misses. Cicero's architecture aims to enumerate all the threads and execute them in a lockstep fashion, flowing over a character at a time.

Table 2. Average energy (Wµs) per RE execution for a subset of the benchmarks [39]. The virtualized enumeration via cross-engine load balancing stops scaling after 9 engines.

Engine #	PROTOMATA	BRILL	PROTOMATA4	BRILL4
1	39.08	72.30	147.74	102.33
4	24.62	72.24	49.52	125.19
9	24.94	68.72	40.27	94.16
16	27.23	73.25	43.58	91.73
32	39.20	105.05	61.66	110.42

Moreover, the multi-engine architecture has a centralized controller to drive the lockstep execution aiming to improve this enumeration capability. Each novel `PC` is evaluated by a distributed load-balancer component that decides at run-time whether or not to offload the thread to the next engine. Their design space exploration shows that this *cross-engine* scale-out strategy pays back up to a certain number of engines while matching four REs in parallel. Indeed, this architectural organization achieves a virtualized enumeration, which leaves the parallelization of RE execution as an open issue. Table 2 illustrates how adding more engines does not imply a performance improvement. First, the single Cicero engine can not overlap the thread execution at a finer grain with a single time-multiplexed core. Second, the multi-engine architecture with the centralized controller adds a non-negligible overhead to distribute the computation across the engines.

Therefore, we design a different architecture organization that can improve this virtualized enumeration capability for more effective parallelization of the RE matching (§4).

3 RE-tailored MLIR Dialects

We devised a novel MLIR-based compilation flow to provide a combined synergic HW-SW approach to the RE matching domain. The proposed compiler pipeline is a linear process that transforms a textual RE into a target binary, eventually loaded into Cicero's architecture. Figure 2 shows the complete system, comparing the old structure with the novel one. The novel infrastructure comprises multiple stages, lowering the RE representation towards the ISA-compliant one. The first stage performs syntax and grammar checking, ensuring that input REs are well-formed and employ only supported operations. It parses the textual RE using the ANTLR4 library [38], which facilitates the creation of lexer rules and a grammar defining the syntax of the RE. This parsing process results in the generation of an Abstract Syntax Tree (AST) representing the structural hierarchy of the RE pattern in an easy-to-analyze way for the next stages.

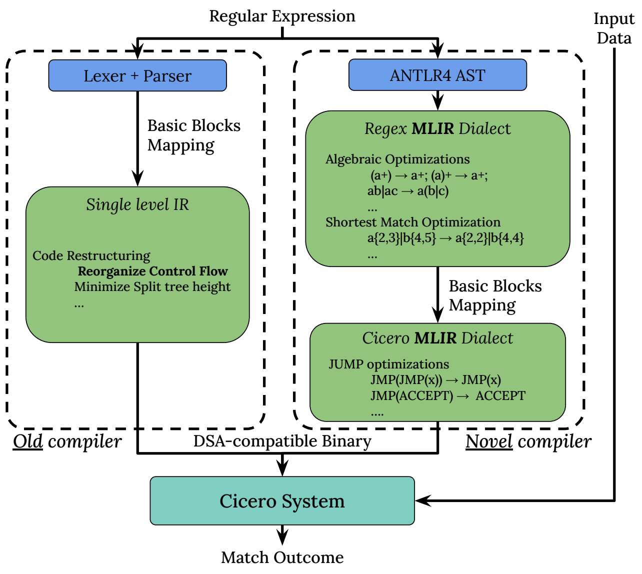


Figure 2. Cicero system compilation flow comparison: the old (left-hand side) and our novel (right-hand side. The compiler translates the RE into an AST, then the novel RE MLIR dialect applies algebraic and shortest-match aware optimizations, and finally, the MLIR Cicero dialect applies back-end specific optimizations by simplifying `JUMP` operations.

Table 3. Regex Dialect - High-Level IR for RE Semantics.

RE Operator	Operation Name	Operation Arguments
`...|...|...`, ^ and $	RootOp	Bool: $hasPrefix, Bool: $hasSuffix
Concatenation	ConcatenationOp	
`{a,b}`, `+`, `*`	QuantifierOp	INT64: $min, INT64: $max
Single character	MatchCharOp	INT8: $targetChar
`.`	MatchAnyCharOp	
`[...]` and `[^...]`	GroupOp	BoolArray: $targetChars
`(...|...|...)`	SubRegexOp	
`$`	DollarOp	

Then, the AST is converted into the proposed high-level MLIR dialect, called **Regex** (§3.1). At this stage, various optimizations can be applied to the IR to improve performance and efficiency without being constrained by the specifics of the target architecture. In the next step, the Regex MLIR dialect is lowered into a lower-level one called **Cicero** (§3.3). The process maps basic blocks to instruction memory and inserts control instructions. This last dialect is specifically tailored to the Cicero architecture, allowing for optimizations that exploit its specific features to maximize performance without restructuring the control flow. The final stage of the compiler pipeline involves code generation, where each operation in the Cicero dialect is transformed into the corresponding instruction in the Cicero ISA. This process completes the compilation of the RE pattern into a binary format that is ready for execution on the Cicero architecture.

3.1 Regex: The High-Level IR Dialect

This dialect introduces a set of operations that enable a flexible and expressive representation of RE patterns and their operators. Table 3 provides an overview of these operations, including their names, arguments, and mapping to RE operators. In particular, `RootOp` is the top-level operation, representing the entire RE pattern. Its boolean arguments specify whether the RE has an implicit $.*$ at the beginning or end. This parameter is paramount in pattern matching, as we are mainly interested in patterns anywhere within the data. Moreover, `RootOp` contains a sequence of `ConcatenationOp` operations, implicitly alternated by the | operator, which in turn contains a sequence of `PieceOp` operations. `PieceOp` acts as a wrapper for `AtomOp` and `QuantifierOp`.

`QuantifierOp` specifies how many times the corresponding `AtomOp` can be repeated by indicating a minimum and maximum amount as integer arguments. `AtomOp` is not a concrete operation but can be any of the remaining ones. `MatchCharOp` and `MatchAnyCharOp` operations are used to match specific characters or any character in the input string, respectively. `GroupOp` matches any of the characters specified by a bitmap argument, e.g., "*[ac]*" becomes [false, ..., true, false, true, false, ...].

`SubRegexOp` matches a sub-RE within the larger RE. Finally, `DollarOp` matches the end of the input string. Listing 1 shows an example IR generated from the RE $(ab)|c\{3,6\}d+$. `hasPrefix=true` indicates an implicit ".*" prefix at the beginning of the RE, while `hasSuffix=true` indicates an implicit ".*" suffix at the end. RE users often want to match a pattern anywhere within the data, so both implicit prefix and suffix parameters are enabled. However, if users need to match from the very beginning or until the very end of the data, they can disable the implicit prefix/suffix by using the caret (^) and dollar ($) standard operators in the RE.

Listing 1. *Regex Dialect* Example for $(ab)|c\{3,6\}d+$

```
regex.root {hasPrefix=true, hasSuffix=true} {
  regex.concatenation { // (ab)
    regex.piece {
      regex.sub_regex {
        regex.concatenation {
          regex.piece { // a
            regex.match_char a
          }
          regex.piece { // b
            regex.match_char b
          }
        }
      }
    }
  } // Implicit "|" between concatenations
  regex.concatenation { // c{3,6}d+
    regex.piece { // c{3,6}
      regex.match_char c
      regex.quantifier from 3 to 6
    }
    regex.piece { // d+
      regex.match_char d
      regex.quantifier from 1 to -1
    }
  }
}
```

As this dialect provides a flexible and abstract representation of the RE structure, it operates independently of the underlying hardware ISA runtime environment. This characteristic ensures its relevance beyond the scope of the Cicero architecture, making it applicable to any other RE-matching engine. Specifically, this approach simplifies the implementation and application of transformations and optimizations agnostic to specific architectures or runtimes, enhancing their utility across various contexts.

3.2 High-Level Transformations for Regex Dialect

We implement three sets of transformations in the high-level Regex dialect to simplify regex expressions in a more concise and straightforward representation. Each transformation is optional and can be enabled or disabled individually by toggling different compiler options. In the first set, we simplify sub-expressions *(sub-Regex)* into a more concise representation, applying canonicalization[2] whenever possible to remove the unnecessary parenthesis. Notable examples are: `(abc)` ⟶ `abc` while `(abc)+` remain the same to respect the operator precedence; `(a+)` and `(a)+` become `a+`; `(a{2,3}){4,7}` remains the same (note that *a{8,21}* would be incorrect, as it would accept 9 *a* which is not accepted by the original RE.)

The second set of transformations focuses on **factorizing alternations** that contain the same prefix, applying the distribution property of the *concatenation* with respect to the *alternation.* These optimizations are implemented for the *sub-Regex* and for the *root* regex. For example: `this|that|those` ⟶ `th(is|at|ose)`; `a(bc|bd)` ⟶ `a(b(c|d))`

The third transformation applies to regex engines aimed at producing *any* match rather than finding the longest match, which is a common scenario in applications requiring at least a match to trigger a specific behavior. We can apply this transformation to quantified operations at the pattern boundaries. Otherwise, it would derive a new semantic. Indeed, pattern matching has an implicit $.*$ at the beginning and the end of the pattern; thus, applying reduction to the quantifiers is permitted only at the boundaries of the RE, e.g., "ab+.*" becomes "ab.*". Notably, this transformation is not executed if the $.*$ prefix or suffix are explicitly disabled via the RE \$ or ˆ operators. We can exemplify this transformation with: `a{2,3}|b{4,5}` ⟶ `a{2}|b{4}`; `abcd*|efgh+` ⟶ `abc|efgh`; while `ab*$` remains the same. All transformations, excluding this last one, preserve the original semantics of the RE with an equivalent behavior.

3.3 Cicero: Low-Level IR Dialect

Although the *Regex dialect* can effectively cover high-level RE representations for general-purpose targets, a lower-level IR is essential for an architecture-specific representation to encode the RE into the target ISA and to implement architecture-oriented optimizations. Consequently, we propose the *Cicero dialect*, designed to represent RE computations in a one-to-one mapping to the Cicero ISA (Table 1). This dialect is crafted to ensure a straightforward lowering process from the *Regex Dialect* while simultaneously being tailored to implement optimizations specific to the Cicero architecture. Table 4 provides an overview of the *Cicero dialect operations (Op)*, including their names, the mapping to Cicero ISA, and arguments (Table 1). Operations are divided into three categories: 1) **Acceptance**: terminate the execution by yielding a positive RE matching result; 2) **Control Flow**: jump to a fixed address or split the execution flow into two threads, handling the alternative paths *enumeration*; 3) **Matching**: match character from the input, otherwise kill the corresponding thread. *Acceptance* operations are divided into two modalities called `AcceptOp` and `AcceptPartialOp`. The former halts execution, signaling the RE matching only if the execution has traversed the entire input, while the latter potentially accepts a partial segment.

Table 4. Cicero Dialect - Low-level IR for Cicero

Cicero ISA	Operation Name	Operation Arguments
`Accept`	AcceptOp	
`Accept Partial`	AcceptPartialOp	
`Split`	SplitOp	Symbol:\$splitReturn
`Jump`	JumpOp	Symbol:\$target
`MatchAny`	MatchAnyOp	
`Match`	MatchCharOp	INT8:\$targetChar
`NotMatch`	NotMatchCharOp	INT8:\$targetChar

Control flow operations encompass `JumpOp` and `SplitOp`. The former executes an unconditional jump, while the latter spawns a thread continuing to the subsequent operation and another one jumping to a specified operation.

Matching operations consist of `MatchAnyOp`, `MatchCharOp`, and `NotMatchCharOp`. The former matches any character in the input string, while the other two are more specific. `MatchCharOp` targets a particular character, while the remaining one matches any character except the specified one. `NotMatchCharOp` does not advance through the input string. This design choice aims to streamline the process of lowering negated groups. For example, `[ˆab]` can just be lowered into `(NotMatchCharOp(a); NotMatchCharOp(b); MatchAnyOp)`. In any case, if the matching or non-matching operation fails, the thread is terminated.

The one-to-one mapping reduces the complexity of the code generation step, making it fast and straightforward to implement Cicero-specific optimizations, e.g., simplifying jumps, thus efficiently enhancing runtime performance.

4 Parallel Enumeration Architecture

As anticipated by §2, the existing microarchitecture relies on virtualizing the alternatives enumeration within an NFA.

[2]https://mlir.llvm.org/docs/Canonicalization/

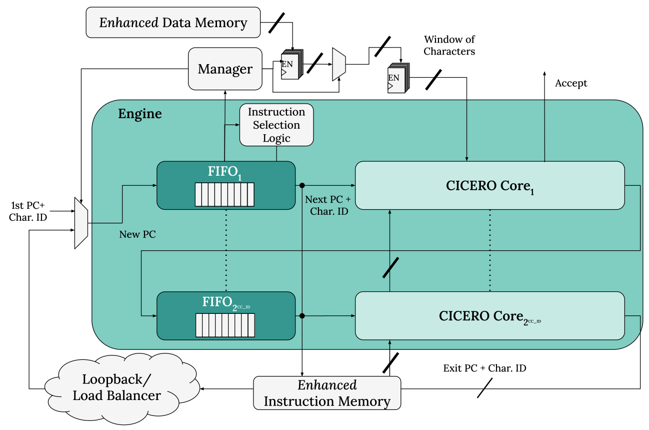
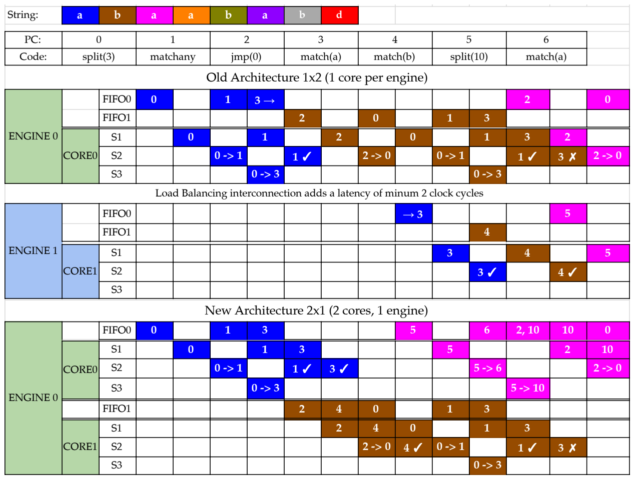


Figure 3. Our proposed new microarchitecture: we apply an effective enumeration of threads by packing multiple cores in a single engine that are all actively processing .



String:	a	b	a	a	b	a	b	d

PC:	0	1	2	3	4	5	6
Code:	split(3)	matchany	jmp(0)	match(a)	match(b)	split(10)	match(a)

Old Architecture 1x2 (1 core per engine)

ENGINE 0		FIFO0	0		1	3 →							2		0
		FIFO1					2		0		1	3			
	CORE0	S1		0		1		2		0		1	3	2	
		S2			0 -> 1		1 ✓		2 -> 0		0 -> 1		1 ✓	3 ✗	2 -> 0
		S3				0 -> 3						0 -> 3			

Load Balancing interconnection adds a latency of minum 2 clock cycles

ENGINE 1		FIFO0								→ 3				5	
		FIFO1										4			
	CORE1	S1									3		4		5
		S2										3 ✓		4 ✓	
		S3													

New Architecture 2x1 (2 cores, 1 engine)

ENGINE 0		FIFO0	0		1	3				5		6	2, 10	10	0
	CORE0	S1		0		1	3				5			2	10
		S2			0 -> 1		1 ✓	3 ✓				5 -> 6			2 -> 0
		S3				0 -> 3							5 -> 10		
		FIFO1					2	4	0		1	3			
	CORE1	S1						2	4	0		1	3		
		S2							2 -> 0	4 ✓	0 -> 1		1 ✓	3 ✗	
		S3										0 -> 3			

Figure 4. High-level view comparison of **old** multi-engine (top) and **new** multi-core (bottom) architectural organizations executions. Each cell contains the PC and is colored with the equivalent character thread. '$a \longrightarrow b$' is a jump towards b; 'a✓' successful match; 'a×' unsuccessful match, thread killed; '$a \longrightarrow$' and '$\longrightarrow a$' represent the thread movement from/to the engine. Our organization reduces threads movements and optimizes parallelization efficiency.

Whenever multiple active paths are present (e.g., the presence of an OR operator allows for multiple matches), the original Cicero multi-engine can leverage this parallelism through what we call *cross-engine* load balancing. The thread count is distributed among an interconnected system of *engines*, each housing a single core and 2^{CC_ID} FIFOs, one for each character in the input window, as shown in Figure 1. The input window is set to exactly 2^{CC_ID} characters wide because it is addressed by a *CC_ID*-bit wide pointer. This approach maximizes the pointer's addressing capability. Considering this network of engines, congestion is solved by moving threads across them, which, however, is an expensive setup in terms of resource, power, and design complexity.

To overcome these limitations, we introduce a new architectural design where the engines are packed with 2^{CC_ID} Cicero cores instead of one, each paired to its corresponding FIFO, as illustrated in Figure 3. On top of this, each core is responsible for processing the threads associated with a single character in the input window. This approach leverages that each core can manipulate threads in three ways: 1) *Update* only `PC` (Control flow operation); 2) *Match* a character and *increment* by 1 both `PC` and `CC_ID`; 3) *Kill* the thread or *terminate* overall execution by yielding the matching result. As such, a thread coming from FIFO N and processed by the corresponding core can only end up in FIFO N or $N + 1$, as its `CC_ID` can remain the same or incremented by 1. The spread of threads to different FIFOs (and consequently cores) naturally facilitates load balancing without the extensive replication of FIFOs in many engines or the movement of threads among engines. We call this technique ***in-engine*** load balancing, as threads only move between FIFOs in the same engine. By eliminating the need for elaborate load-balancing mechanisms and redundant FIFOs in multiple engines, we create a more efficient and resource-conscious system that improves the performance scalability.

Our new engine design is backward compatible with the previous version, maintaining an identical ISA and microarchitectural interface. Indeed, the new engine can adopt the same multi-engine approach but with a significant distinction. The former engine featured a single core where its output threads were promptly routed to the load-balancing infrastructure, and incoming threads were just multiplexed to the corresponding FIFO. Instead, our new engine comprises multiple cores, but only one core can connect to the existing load-balancing infrastructure. In our design, the last core outputs threads to the load balancer, while the first FIFO receives the incoming threads. Consequently, should thread congestion occur in the second core (such as a chain of split instructions), *cross-engine* load balancing can only commence once threads progress to the final core of that engine. This observation indicates that having multiple engines with our design is less efficient than the original Cicero, given the limited movement of threads between engines and, therefore, minimal *cross-engine* load balancing.

To summarize, we provide an exemplification of how the two architectural organizations execute the same RE on an input string in Figure 4. Our novel organization can significantly enhance the architecture's scalability, reducing resource and power consumption with a higher locality while keeping compatibility at the ISA and microarchitectural level.

5 Architecture-Aware Optimizations

Until now, we have considered the multi-level compiler optimizations and the enhanced parallel architecture separately. However, synergically combining these two approaches opens up further improvements, surpassing single-component benefits with higher performance and efficiency [25].

Each Cicero core fetches instruction from an instruction cache, which, in the case of cache misses, pays the cost of reading the central instruction memory (see Cicero Core in fig. 1). For this reason, the architecture is very susceptible to instruction cache misses, demonstrating better performance when the code it executes exhibits a better locality (see Figures 10 and 11 in §6 for a quantitative analysis). Therefore, we aim to improve code locality through compiler transformations by evaluating the code locality statically at compile time through the following proxy metric. We define the *total jump offset* D_{offset} as:

$$D_{\text{offset}} \triangleq \sum_{i \in I} d_{\text{offset}}(i) \tag{1}$$

where $d_{\text{offset}}(i)$ is 0 for all instructions except for `JumpOp` and `SplitOp`, for which it is the offset of the jump. These offsets represent the distances between basic blocks. A higher value indicates a lower code locality, as basic blocks are farther apart. We use this metric to compare the code locality of the old and new compilers and to evaluate our optimizations.

We start by analyzing the original Cicero compiler, focusing on the *Code Restructuring* [39] optimization. This optimization reorganizes the sequences of Split instructions into a tree with minimal depth, with the goal of minimizing the longest instruction path to execute any of the leaves in the tree. An example is shown in Figure 5, where we can see that the number of jump instructions is reduced and the tree's depth is decreased. While this appears promising in theory, it could quickly degrade the performance in practice. To better understand the reasons behind this point, we must consider two factors. First, split instructions produce two threads that could end up in two different FIFOs, but in case of congestion, they end up in a single FIFO, making the execution serial. Second, this optimization restructures the control flow, resulting in basic blocks that are farther apart. To demonstrate the latter, we measure D_{offset} before/after applying *Code Restructuring* on a simple example. Specifically, Figure 6 illustrates the process on the RE $ab|cd$, preceded by an implicit matching of ".∗". In this case, the transformation is not effective, and furthermore, Listing 2 shows how the code locality is negatively affected in the assembly.

Instead, our approach wants to optimize code locality without restructuring the control flow. Thus, we introduce a new *Jump Simplification* optimization in the low-level IR, which performs a simplification of the jump structures and avoids jumping to `AcceptPartialOp` operations by duplicating them, as shown in Figure 7 for the same RE $ab|cd$.

More in detail, *Jump Simplification* is applied to each `JumpOp`: 1) it is removed if it targets the next operation; 2) it is replaced by an acceptance operation (`AcceptOp` or `AcceptPartialOp`) if it targets one; 3) if it targets a second `JumpOp`, the first `JumpOp`'s target is updated to the second's one: this process is repeated recursively.

Indeed, reducing jumps to the next instruction or chain of jumps requires fewer instructions and reduces control flow changes. Finally, we relax the condition of a single acceptance state by changing the sequence of jump-acceptance directly into an acceptance operation. In this way, the NFA traversal can stop as soon as possible without paying the cost of additional jump operations.

Listing 2 shows the generated Cicero assembly code for the running example RE $ab|cd$ without the optimization, using the *Code Restructuring* one, or using the *Jump simplification* one. Listing 2 also reports D_{offset}, showing that the proposed optimization leads to an improved locality compared to the baseline non-optimized code and especially to the one optimized by *Code Restructuring*.

6 Evaluation Results

This section evaluates the benefits of the introduced contributions. Firstly, it considers the single-IR and MLIR compiler impacts on the generated code size, compilation time overhead, code locality, and final REs execution times (§6.1). Then, it evaluates the proposed enhanced parallel architecture in terms of FPGA resource usage, power consumption, and REs execution times (§6.2). To measure execution times, we first count the cycles required to complete the execution of a complete benchmark and then divide by the number of REs executed to obtain the average cycles needed for a single RE. Then, we divide it by the clock frequency to obtain the execution time and multiply it by the total on-chip power estimated by Vivado's power analysis tool for energy usage.

We evaluate Cicero architectures and compilers on an Ultra96 v2 board powered by an AMD Zynq Ultrascale+ MPSoC XCZU3EG A484. In the remainder of the text, we refer to the previous Cicero architecture and compiler as **old** and the proposed ones as **new**. When comparing architectures, we consider engines in ring topology since the previous work [39] proved it is the most efficient.

Similar to the [39] evaluation, we use Protomata and Brill benchmarks from AutomataZoo [54] in the simple and *alternate* strategy. The simple one takes the first 200 REs from each benchmark. The *alternate* one reflects the scenarios in which, among a set of REs, it is essential to have at least one of them matching to trigger an acceptance behavior. For this purpose, we randomly sample 800 REs from each benchmark and alternate 4 at a time in a single RE using the | operator, resulting in 200 REs, called Protomata4 and Brill4, respectively. For our evaluation, we split the input data into chunks of 500 bytes each: the first 10000 chunks for 10MB Protomata

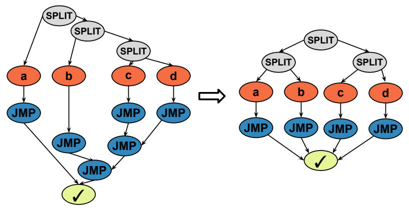


Figure 5. Previous *Code Restructuring* transformation applied to RE $(a|(b|(c|d)))$. The tree of splits is balanced, its depth minimized, and the number of JMPs is reduced.

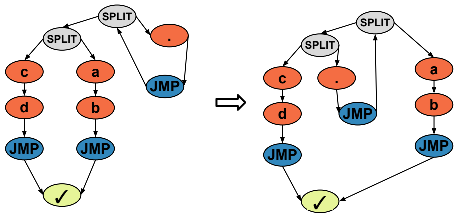


Figure 6. Previous *Code Restructuring* transformations applied to the RE $ab|cd$ with an implicit $.*$ at the beginning. The transformation has a negative impact on the implicit term as it now executes two SPLIT instead of one, and the amount of JMP remains the same.

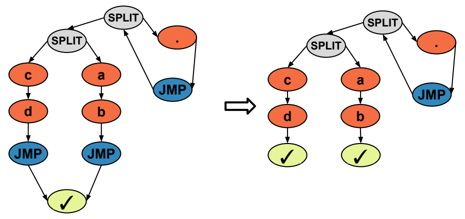


Figure 7. Proposed *Jump simplification* transformations applied to the RE $ab|cd$, comprising an implicit $.*$ at the beginning. The amount of JMP is reduced, while the traversal of split remains optimal.

Listing 2. Assembly comparison and D_{offset} (the lower, the better) for $ab|cd$, applying different optimizations.

```
No optimization:
D_offset = 3+2+5+1+3 = 13

000: SPLIT        {1,3}
001: MATCH_ANY
002: JMP to       0
003: SPLIT        {4,8}
004: MATCH        char a
005: MATCH        char b
006: JMP to       7
007: ACCEPT_PARTIAL
008: MATCH        char c
009: MATCH        char d
010: JMP to       7
```

```
Code Restructuring opt:
D_offset = 4+4+4+9 = 21

000: SPLIT        {1,4}
001: MATCH        char a
002: MATCH        char b
003: ACCEPT_PARTIAL
004: SPLIT        {5,8}
005: MATCH        char c
006: MATCH        char d
007: JMP to       3
008: MATCH_ANY
009: JMP to       0
```

```
Jump Simplification opt:
D_offset = 3+2+4 = 9

000: SPLIT        {1,3}
001: MATCH_ANY
002: JMP to       0
003: SPLIT        {4,7}
004: MATCH        char a
005: MATCH        char b
006: ACCEPT_PARTIAL
007: MATCH        char c
008: MATCH        char d
009: ACCEPT_PARTIAL
```

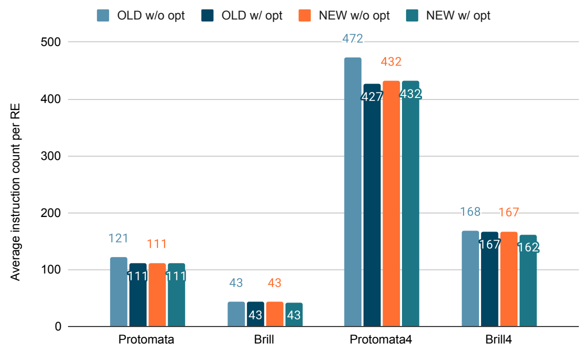


Figure 8. Code Size comparison of the **new** and **old** compilers w/ and w/o optimizations.

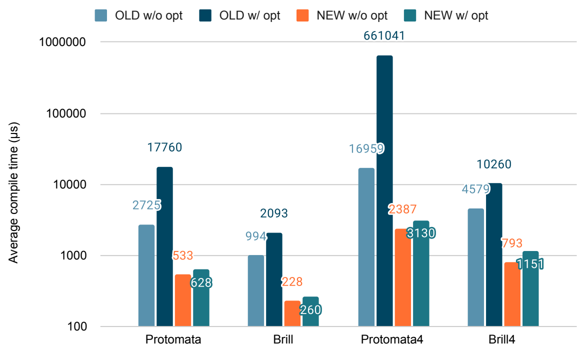


Figure 9. Compile Time comparison (log scale) of the **new** and **old** compilers w/ and w/o optimizations.

and 4018 chunks for 2MB Brill. The *alternate* versions of the benchmarks use the same inputs as their simple counterpart since they are merely compositions of REs.

6.1 Compilers Comparison

We start comparing the **old** and **new** compilers measuring static indicators: code size, compilation time, and code locality (Equation (1)). We always evaluate both of them with and without enabling the optimizations. To conclude, we demonstrate the impact that the two compiler toolchains have on

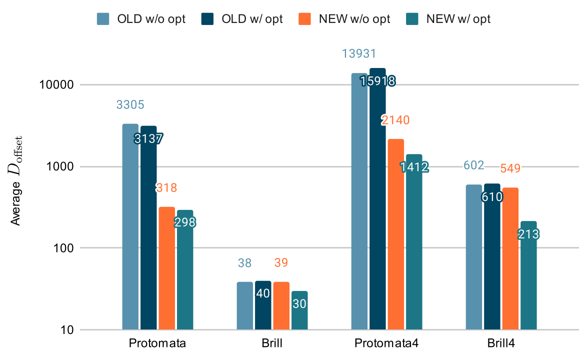


Figure 10. Code locality (Equation (1)) of **new** and **old** compilers w/ and w/o optimizations (log scale). The lower, the better.

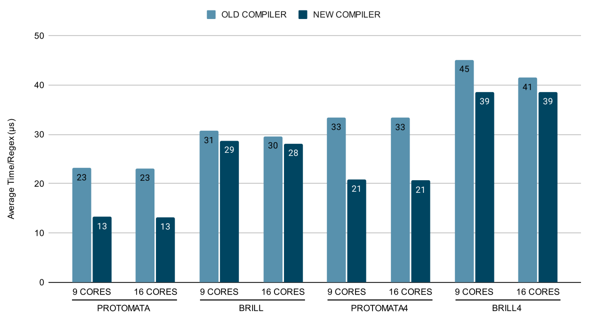


Figure 11. Compilers impact on the **old** architecture with 9 and 16 cores considering the average execution times per RE for each benchmark. The lower, the better.

the execution time when considering the top (unmodified) architectures of the original Cicero paper[39].

Code Size and Compilation Time Overhead. Figures 8 and 9 showcase the averaged results of these metrics. Notably, the code sizes remain similar for both compilers when optimizations are enabled, highlighting that the **new** compiler optimizations do not require larger instruction memories.

When optimizations are turned off, our compiler exhibits remarkable improvement in compilation times, which are 5.11×, 4.36×, 7.10× and 5.77× faster for Protomata, Brill, Protomata4, and Brill4, respectively. The improvement can be derived from the choice of the more performing state-of-the-art MLIR framework during compiler construction.

We can focus on the overhead impact that the optimizations introduce for the different compiler infrastructures. The old one slows down by factors of 6.52×, 2.10×, 38.98×, and 2.24× for Protomata, Brill, Protomata4, and Brill4. Despite having multi-level passes, our novel compilation infrastructure's optimizations introduce an overhead of 1.18×, 1.14×, 1.31×, and 1.45×, respectively. The lower overhead comes from the tailoring and efficiency of each pass, which is performed at the proper level without adopting a single complex pass, which slows down the process.

Overall, reducing compilation time is particularly beneficial for scenarios requiring frequent reprogramming of the RE being executed, such as the database scenarios [28].

Code Locality. We measure the previously introduced proxy metric for code locality, D_{offset} (Equation (1)), computed through static analysis of the compilers' generated code. This metric sums the jump offsets of control flow instructions, which represent the distances between basic blocks. A higher value indicates a lower code locality, as basic blocks are farther apart. Figure 10 illustrates the average D_{offset} over all the considered benchmarks, showing that the **new** compiler excels in consolidating code paths and highlighting the effectiveness of our optimizations. Specifically, the proposed compiler achieves a code locality improvement of 10.53×, 11.27×, and 2.88× over the **old** compiler for Protomata, Protomata4, and Brill4, respectively, while Brill remains steady. Our multi-abstraction approach avoids the premature lowering issue and improves code locality compared to the single ineffective IR.

Compilation Impact on RE Execution Time. We now aim to evaluate the benefit of the new compiler on the old Cicero architecture, isolating the sole compilation flow benefit. Figure 11 shows that the **new** compiler results in 1.7× and 1.2× faster execution for Protomata(4) and Brill(4) benchmarks than using the **old** one. Protomata4 has a more significant improvement in execution time than Brill4, which reflects the higher enhancement in code locality (Figure 10) despite a slightly higher instruction count than before (Figure 8). As anticipated by §5, this evidence highlights the importance of prioritizing code locality over minor theoretical optimizations, which is crucial for improving execution time performance.

6.2 Architecture Configurations Comparison

Since our compiler already yields optimal gains for the previous architecture, we now consider only the proposed compiler to demonstrate that the synergy of HW and SW leads to even further improvements. As we evaluate several Cicero configurations, we use the concise `NxM CORES` wording to represent a configuration that packs `N` cores in `M` engines. Notably, configurations with $N > 1$ are enabled by our contributions and were not previously attainable in the **old** one. The original Cicero manuscript [39] extensively evaluates previous architecture configurations, electing the multi-engine architecture with topology ring and $CC\_ID = 3$ as the optimal ones.

For the **old** architecture (i.e., $N = 1$), we limit the number of engines to $M \leq 32$ due to the scalability limitations detailed in Table 2 in §2. The **new** architectures taken into account are organized into groups based on CC_ID. Thus each group has $N = 2^{CC\_ID}$. We select $CC\_ID \geq 3$ to align

Table 5. Average energy per RE execution (Wμs) for each micro-benchmark and architecture configuration, and overall average. For each micro-benchmark, the most efficient configuration within each group is highlighted in **bold**, while the overall most efficient one is underlined.

Configuration	PROTOMATA	BRILL	PROTOMATA4	BRILL4	AVG overall
OLD 1x1 CORES	39.08	72.30	147.74	102.33	90.36
OLD 1x4 CORES	**24.62**	72.24	49.52	125.19	67.89
OLD 1x9 CORES	24.94	**68.72**	**40.27**	94.16	**57.02**
OLD 1x16 CORES	27.23	73.25	43.58	**91.73**	58.95
OLD 1x32 CORES	39.20	105.05	61.66	110.42	79.08
NEW 8x1 CORES	**22.65**	**61.03**	**35.35**	**76.86**	**48.97**
NEW 8x4 CORES	26.03	69.70	39.23	85.04	55.00
NEW 8x9 CORES	30.84	82.60	45.52	100.75	64.93
NEW 8x16 CORES	38.14	102.24	55.22	124.47	80.02
NEW 16x1 CORES	**24.54**	**64.40**	**28.54**	**73.94**	**47.86**
NEW 16x4 CORES	32.96	86.34	37.39	97.52	63.55
NEW 16x9 CORES†	54.47	142.68	60.32	160.65	104.53
NEW 32x1 CORES	**31.90**	**80.40**	**34.54**	**86.56**	**58.35**
NEW 32x4 CORES†	57.98	146.07	61.83	156.81	105.67

All at 150 MHz, except † at 100 MHz as > 70% LUTs and > 90% BRAMs are used.

with the previous work [39] findings and constrain $M \leq 32$ and $CC\_ID \leq 5$ ($N \leq 32$), which allows a better energy-efficiency and resource utilization, showcasing a better scaling compared to the **old** architecture organization. Notably, we exclude **new** architecture configurations with a number of cores not equally sized to 2^{CC_ID} (e.g., 9x1 CORES) as they would be underutilized by construction, and those that do not fit the FPGA resource budget (e.g., 32x9 CORES).

Micro Benchmarking Pre-Filtering. Given the large combination set of possible architecture configurations and benchmarks, we aim to filter out less energy-efficient architectures with micro benchmarking by sampling a limited number of REs and inputs from each benchmark to gain immediate insights into discerning viable configurations. Specifically, we select the first 100 REs and inputs from each benchmark and execute them on the **old** multi-engine architectures (i.e., 1xM configurations) and the **new** architectural configurations (i.e., NxM ones). Table 5 reports the average energy required to execute a single RE, computed as the average RE execution time multiplied by the corresponding power consumption (see Figure 12 for the power). Our **new** multi-engine configuration (i.e., NxM) is always less efficient than the **new** single-engine counterpart (i.e., Nx1), as we expect from our design choice of using the *in-engine* balancing (§4). Specifically, this result confirms that packing multiple cores in a single engine is beneficial due to the removal of complex thread distribution among the cores. Therefore, we excluded such configurations from the following extensive evaluation, keeping in consideration NEW 8x1, NEW 16x1, and NEW 32x1 for our **new** architecture and OLD 1x9 and OLD 1x16 for the **old** configuration, being the best compromises.

Power Consumption and Resource Utilization. For completeness with the micro-benchmarking, Figure 12 reports the power consumption of different configurations scaling the cores and the number of engines. Instead, Figure 13 illustrates the resource utilization for the restricted selected configurations on the Look-Up Tables (LUTs), Flip Flops (REGs), and Block RAMs (BRAMs) (DSPs are unused). We observe that although the OLD 1x16 and NEW 16x1 contain the same amount of Cicero cores, we have a lower power consumption and resource utilization on the FPGA. Indeed, our strategy of packing multiple cores is more efficient, as the *in-engine* balancing avoids the usage of costly external additional FIFOs for additional engines and load distribution stations while reducing the thread movement energy cost. Moreover, we can scale to a higher amount of computing units (i.e., greater than 16), which, however, has a high impact in terms of power and resource consumption. Notably, the NEW 8x1 is the most resource-efficient configuration among the ones evaluated across all the resources.

Execution Time and Energy Consumption. In this case, we extensively evaluate the benchmarks by averaging the RE execution time. Figure 14 illustrates the speedup of each configuration normalized to OLD 1x9 CORES one. Notably, despite using fewer resources than the **old** configurations (Figure 13), NEW 8x1 CORES achieves comparable execution time against them.

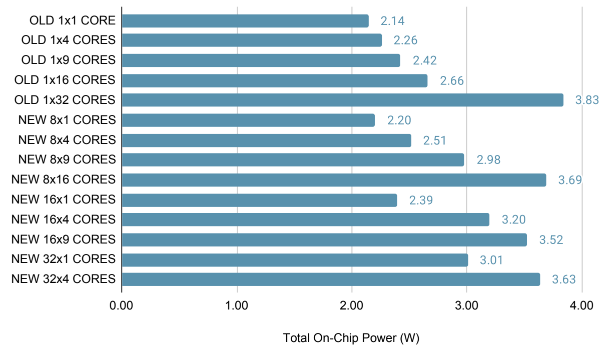


Figure 12. Power consumption (static + dynamic) comparison for all the architecture configurations of **old** and **new**.

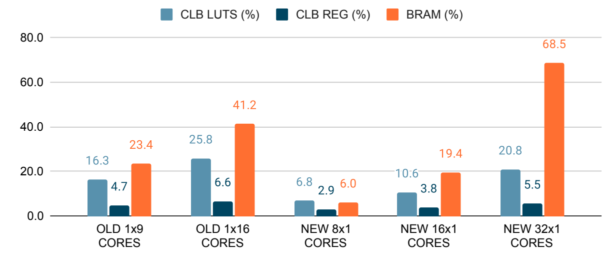


Figure 13. Resource usage (%) of **old** and **new** architectures on the XCZU3EG. NEW 8x1 is the most resource-efficient.

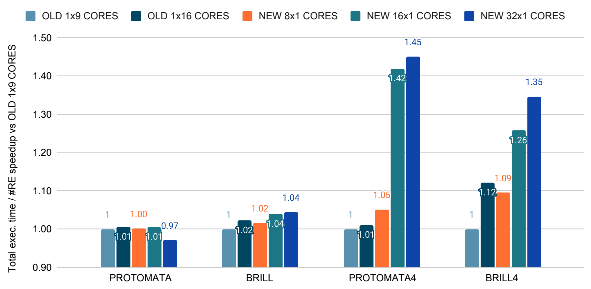


Figure 14. RE execution speedup normalized against the `OLD 1x9 CORES`. Our synergic HW-SW NEW16x1 configuration always yields improvements over the best **old** configurations.

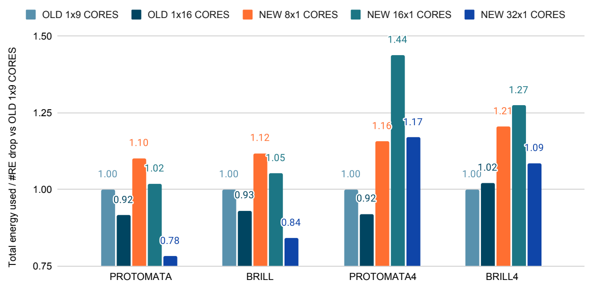


Figure 15. Energy efficiency improvements normalized against the `OLD 1x9 CORES`. The `NEW 8x1 CORES` resource-efficient configuration is the best in the single-RE scenario, while the `NEW 16x1 CORES` shows the best improvements over [39] for the alternated benchmarks.

We then measure the energy by multiplying the average RE execution time with the corresponding power consumption, similar to the micro-benchmarking. Figure 15 illustrates the energy efficiency improvement normalized with respect to the `OLD 1x9 CORES` configuration, which aligns with the preliminary analysis for the micro-benchmarks (Table 5). Despite the `NEW 8x1 CORES` not being the fastest configuration for the simpler Protomata and Brill benchmarks, its resource efficiency and reduced power consumption make it the most energy-efficient option for these scenarios. Instead, when considering the alternated, thus more parallel, Protomata4 and Brill4 benchmarks, the `NEW 16x1 CORES` stands out as the most energy-efficient solution with 1.44× and 1.27× of improvements against the older architectural organization.

Result Summary. Table 6 highlights the results when considering the initial **old** compiler infrastructure and architectural organization of `1xM` engines against the novel MLIR compiler and microarchitecture. We obtain a 1.48× speedup and 1.56× energy efficiency improvement by averaging over all the benchmarks and a top 2.27× speedup and 2.30× energy

Table 6. Average RE performance (execution time and energy consumed) for *alternate* benchmarks (in bold the best), and overall average for *single* and *alternate* benchmarks. We compare the best configurations of the **old** and **new** approaches, showcasing our whole improvements.

Configuration	PROTOMATA4		BRILL4		AVG overall	
	[µs]	[Wµs]	[µs]	[Wµs]	[µs]	[Wµs]
Old Compiler, OLD 1x9 CORES	33.41	**80.95**	45.16	**109.42**	33.11	**80.23**
Old Compiler, OLD 1x16 CORES	**33.36**	88.70	**41.50**	110.34	**31.90**	84.82
New Compiler, NEW 9x1 CORES	19.86	43.70	35.25	77.59	24.17	53.20
New Compiler, NEW 16x1 CORES	**14.71**	**35.16**	**30.70**	**73.41**	**21.57**	**51.58**
Best (old) / Best (new)	2.27	2.30	1.35	1.49	1.48	1.56

efficiency improvement over Protomata4. Therefore, the synergic combination of the novel compiler and the enhanced architectural organization exhibits better scalability in terms of resource utilization and energy efficiency, thanks to an improved abstraction through a multi-dialect MLIR-based infrastructure and a more resource-efficient design.

7 Related Work

The *embarrassingly sequential* nature of the RE computational kernel pushes for different solutions to solve its intrinsic data dependency, with the goal of providing low-latency, high-throughput, and energy-efficient solutions.

Overall, the literature proposes three different approaches to overcome RE challenges, which can be categorized into: CPU-based [22, 41, 42, 56, 63], automata-based accelerators [8, 15, 16, 19, 20, 32, 33, 35, 43, 45, 45, 48, 55, 58, 60, 61, 64], and DSAs [6, 13, 17, 21, 37, 39, 52, 59]. **CPU-based** solutions exploit general-purpose architectures to execute REs, allowing a large set of supported operators and run-time flexibility. Google RE2 [22] provides fast RE matching without relying on specific HW extensions, while Intel Hyperscan [56] exploits SIMD extensions of x86 CPU and efficient sub-string analysis at the cost of high compilation overhead and CPU extension dependency. However, HW offloading is paramount to execute RE matching in scenarios such as SmartNIC for DPI, where saving precious CPU cores for central tasks and avoiding cumbersome data movement to the host is paramount [18]. **FPGA-based** accelerators exploit FPGA features to map automata directly on the fabric and provide high performance [35, 43, 45]. However, changing the REs can require a high update time to produce the new bitstream to deploy [35, 58, 61]. **In-memory-based** solutions offer a valid alternative to reduce data movement costs and increase the efficiency in either external DRAM-based configuration [16, 45] or within the CPU memory hierarchy [48]. Other solutions [8, 19, 32, 33, 64] aim to exploit GPU data parallel characteristics to simultaneously execute multiple active transitions in the NFAs. Despite that, various works [5, 35] have demonstrated that GPU-based solutions are slower than CPU ones due to RE intrinsic serialization.

The introduction of **MLIR** has simplified the development of compilation infrastructures, notably improving the ease of creating new compilers, with a particular focus on domain-specific languages (DSLs) [24, 29, 57]. Simultaneously, the adoption of multiple levels of IRs has gained prominence even in compilers for general-purpose languages, such as Swift [12], which introduces the Swift Intermediate Language (SIL), and Rust [44], which features three layers of IR before lowering to LLVM IR and finally genereting code. Our employed high- and low-level optimizations are used in CPU compilation flows, but to the best of our knowledge, this is the first work to use them clearly in the REs domain, exploiting a multi-dialect organization and exploring their benefits and impacts. Existing RE compilers either perform translations into their equivalent automata representation (NFA/DFA) with automata-based optimizations [9, 16, 22, 43, 45, 56] or simply transform the RE(s) to bytecode [6, 13, 39]. Thus, they miss the possible benefit of the proposed multi-dialect RE-aware approach.

Code locality is paramount to boost general-purpose processor performance through various techniques [23, 27, 40, 51]. Researchers improve the code layout to reduce the pressure on the instruction memory hierarchy by profiling the application code [27, 40] or even the operating system [51]. Although the issue is similar to the ones analyzed in this work, and we borrow some concepts for readability, the proposed locality work focuses on microarchitectural and code details specific to the RE domain and Cicero. Indeed, our Jump Simplification technique addresses various scenarios, such as simplifying jumps that target other jumps. This is a specific case of Jump Threading[3], simplified by the fact that our jumps are unconditional and can, therefore, always be threaded (i.e., simplified).

8 Conclusions and Future Work

This paper proposed a multi-dialect compiler and an enhanced architecture organization combined to synergically improve one of the most promising DSAs for REs execution, Cicero. Abstracting the RE representations and optimization through an MLIR-based approach leads to a synergic improvement of the parallelization with top improvement of 2.27× and 2.30× in execution time and energy efficiency.

Future Work. Future directions for this work can extend the current ISA for acceptance instructions to support RE identification in multi-matching scenarios. In this way, the execution engine could return the RE identifiers when a match occurs, increasing the analysis information. Moreover, other existing DSAs could be added as the targeted back-end, pushing for a standard MLIR-based multi-dialect compilation flow for REs execution engines.

Acknowledgments

This work has financial support from ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing, funded by European Union – NextGenerationEU. The authors are grateful to the anonymous reviewers feedback, the AMD University Program support for the board used in this work, and V. Sona for working on the original Cicero architecture simulator.

[3]https://llvm.org/doxygen/classllvm_1_1JumpThreadingPass.html

A Artifact Appendix

A.1 Abstract

This document provides all the information about the artifact to reproduce the main findings of the paper and potentially modify the proposed work for further experiments. The SW/HW setup comprehends two different environments to reproduce all the main results. The x86 machine is used for experiments on the compilation flow (i.e., average compilation time, code size, code locality) and bitstream generation (i.e., FPGA resource usage and total on-chip power). A Docker container is provided with all the dependencies installed to reproduce the result on the compilation, simplifying the procedure. Then, the Ultra96-V2 board is essential to execute the REs benchmarks on the different architecture implementations and reproduce the relative results (i.e., average runtime to match a RE and average energy used to match a RE).

A.2 Artifact check-list (meta-information)

- **Compilation:** C++, CMAKE, LLVM/MLIR 16, Antlr4, bash
- **Data set:**Automatazoo [54]
- **Run-time environment:** Python3, PYNQ 3.0.1
- **Hardware:** Ultra96-v2 FPGA board; x86 Machine
- **Metrics:** average compilation time (µs), code locality, average code size, FPGA resource usage (%), total on-chip power (W), average runtime to match a RE (µs), average energy used to match a RE (Wµs)
- **Output:** textual tables and PNG figure
- **Experiments:** Replicate Figures 8, 9, 10, 11, 12, 13, 14, 15; Table 5
- **How much disk space required (approximately)?:** 5GB
- **How much time is needed to prepare workflow (approximately)?:** 30 min
- **How much time is needed to complete experiments (approximately)?:** 48 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Data licenses (if publicly available)?:** Refer to single benchmarks licenses
- **Archived (provide DOI)?:** 10.5281/zenodo.13345346

A.3 Description

A.3.1 How delivered. Available on Github at https://github.com/necst/cicero; Zenodo at https://doi.org/10.5281/zenodo.13345346

A.3.2 Hardware dependencies. Two hardware devices are needed:

1. x86 Machine to evaluate code size, compilation time, and code locality metrics of **OLD** and **NEW** compilers.
2. Avnet Ultra96-V2 FPGA board to execute RE benchmarks on the evaluated **OLD** and **NEW** Cicero architectures.

A.3.3 Software dependencies.

1. **x86 Machine**: (Required for comparison of static compilers metrics) Ubuntu 22.04, Docker. (Optional, only to build bitstreams) Vivado 2019.2, python3, zip, and unzip. For simplicity, the comparison of static compiler metrics is automated within a provided Docker container. Otherwise, dependencies can be installed manually in a Ubuntu 22.04 host by running the commands within `cicero_compiler_cpp/Docker/Dockerfile`. Dependencies include g++, CMAKE, LLVM/MLIR 16, Antlr4; and some Python libraries: rich, tqdm, matplotlib, ply.
2. **Avnet Ultra96-V2 FPGA board**: PYNQ image version 3.0.1; g++, CMAKE, LLVM/MLIR 16, Antlr4; Python dependencies: rich, tqdm, matplotlib.

A.3.4 Data sets. Automatazoo [54]. Data sets are already included with the rest of the provided artifact. They both enclose the REs to execute and the data to analyze.

A.4 Installation

A.4.1 x86 Machine. This setup is based on a Docker container to simplify the evaluation process. Follow https://docs.docker.com/engine/install/ for instructions regarding Docker installations on the evaluation machine. Then, use the following command to clone the code repository and its submodules:

```
git clone --recursive https://github.com/necst/cicero
```

A.4.2 Ultra96 v2 FPGA. Pre-build PYNQ images are available from the official website https://www.pynq.io/boards.html. Instructions on how to load the image on the SD card are available in the official documentation. For further information on the board usage, refer to the official guide. Once the board is powered on, clone the code repository and install the required Python dependencies:

```
git clone --recursive https://github.com/necst/cicero
pip install rich==11.2.0 tqdm==4.64.0 matplotlib==3.5.1

# Install C++ dependencies
mkdir -p /etc/apt/sources.list.d
apt-get update && apt-get install -y ca-certificates gnupg
apt-key adv --recv-key --keyserver keyserver.ubuntu.com 15
        CF4D18AF4F7421
echo 'deb http://apt.llvm.org/jammy/ llvm-toolchain-jammy-16 main'
         > /etc/apt/sources.list.d/llvm.list
apt-get update && apt-get install -y libmlir-16-dev mlir-16-tools
        llvm-16-dev antlr4 libantlr4-runtime-dev cmake clang-16
```

A.5 Experiment workflow

A.5.1 x86 Machine. To replicate Figure 8, Figure 9, and Figure 10, an automatic script that executes benchmarks in a Docker container is provided and can be executed as follows:

```
sudo bash scripts/artifact_evaluation/
        compare_compilers_static_stats/run_benchmarks.sh
```

Listing 3. Reproduce results comparing compiler static metrics (ETA: 30 minutes - around 2GB download size)

Results are both printed to console as raw textual tables and plotted to PNG files in `scripts/artifact_evaluation/compare_compilers_static_stats/script_within_docker/fig*.png`.

Pre-built `bitstreams` are provided in the `bitstreams` folder together with the FPGA resource usage and total on-chip power to reproduce Figure 12 and Figure 13 raw data (i.e., `bitstreams/cicero_confs_usage_power.CSV`), simplifying the reproduction process. Optionally, `bitstreams` can be generated from sources. An automatic script generates the bitstreams, extracts the static metrics (i.e., board usage and total on-chip power), and, finally, puts the generated files in the bitstreams folder. The script can be executed as follows:

```
bash scripts/artifact_evaluation/
      generate_bitstreams_and_extract_metrics.bash
```

Listing 4. Re-build bitstreams from architecture source code configurations and extract static metrics (ETA: 24 hours)

Note that before running the synthesis script, Python variables in scripts/synth/synth.py should be updated. In particular, VIVADO_SOURCE must be changed with the current Vivado installation path, and MAX_WORKERS can be changed if you want to run multiple bitstream generations in parallel.

A.5.2 Avnet Ultra96-V2 FPGA board.

The first step is building the compiler:

```
cd cicero_compiler_cpp
mkdir build
cd build
cmake .. -DBUILD_TESTING=OFF -DCMAKE_CXX_COMPILER=$(which clang
      ++-16) -DMLIR_DIR=/usr/lib/llvm-16/lib/cmake/mlir
cmake --build .
```

Listing 5. Build **NEW** compiler (ETA: 5 minutes)

To replicate Figure 11, Figure 14, Figure 15, and Table 5, you first need to run all the required benchmarks. An automated script is provided, and it schedules all the benchmarks in batches and saves the results in CSV files. The script must be run as root:

```
sudo bash scripts/artifact_evaluation/benchmarks/
      run_all_benchmarks.sh
```

Listing 6. Run all RE-matching benchmarks on the FPGA board (ETA: 30 hours)

Once all the benchmarks are completed, a script is provided to automatically aggregate the intermediate results:

```
python3 scripts/artifact_evaluation/benchmarks/
      aggregate_benchmarks_results.py
```

Listing 7. Aggregate and combine all RE-matching benchmarks on FPGA board (ETA: 5 minutes)

Results are plotted as PNG images in scripts/artifact_evaluation/benchmarks/fig*.png (Figure 11, Figure 14 and Figure 15) and to console as raw textual tables (Table 5).

A.6 Evaluation and expected result

All the results are printed as raw textual tables, and some are plotted as graphs in PNG images for better visualization of the trends. Compilation times can differ based on the CPU used, but general trends can be reproduced. RE-matching on the FPGA should have near-identical processing times thanks to the stability of the hardware environment. If bitstreams are re-generated from source, board usages and total on-chip power may vary slightly due to inherent randomness in the synthesis process.

A.7 Experiment customization

Benchmarks started using scripts/artifact_evaluation/benchmarks/run_all_benchmarks.sh can be configured by changing the files in scripts/artifact_evaluation/benchmarks/configs. Individual benchmarks can be configured by changing scripts/measurements/benchmark/bench_config.py, and then their execution can be scheduled running as root scripts/measurements/benchmark/test_top.py. The generation of the bitstreams can be customized by updating CONFIGURATIONS in scripts/synth/synth.py.

References

[1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers Principles, Techniques & Tools.* pearson Education.

[2] Michela Becchi and Patrick Crowley. 2007. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems.* 145–154. https://doi.org/10.1145/1323548.1323573.

[3] Michela Becchi and Patrick Crowley. 2008. Efficient regular expression evaluation: Theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems.* 50–59. https://doi.org/10.1145/1477942.1477950.

[4] Philip Bille, Inge Li Gørtz, and Max Rishøj Pedersen. 2023. Faster Compression of Deterministic Finite Automata. *arXiv preprint arXiv:2306.12771* (2023). https://arxiv.org/pdf/2306.12771.

[5] Filippo Carloni, Davide Conficconi, Ilaria Moschetto, and Marco D Santambrogio. 2023. Yarb: a methodology to characterize regular expression matching on heterogeneous systems. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS).* IEEE, 1–5. https://doi.org/10.1109/ISCAS46773.2023.10181547.

[6] 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).* 193–206. https://doi.org/10.1145/3649329.3657378

[7] Filippo Carloni, Leonardo Panseri, Davide Conficconi, Mattia Sironi, and Marco D Santambrogio. 2023. Enabling efficient regular expression matching at the edge through domain-specific architectures. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, 71–74. https://doi.org/10.1109/IPDPSW59300.2023.00023.

[8] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010), 20–26. https://doi.org/10.1145/1880153.1880157.

[9] Luisa Cicolini, Filippo Carloni, Marco D Santambrogio, and Davide Conficconi. 2024. One Automaton to Rule Them All: Beyond Multiple Regular Expressions Execution. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 193–206. https://doi.org/10.1109/CGO57630.2024.10444810.

[10] Cisco. 2024. ClamAV®: An Open-source Antivirus Engine for Detecting Trojans, Viruses, Malware & Other Malicious Threats. http://www.clamav.net.

[11] Cisco. 2024. Snort - Open Source Intrusion Prevention System (IPS). https://www.snort.org/.

[12] Swift Community. 2024. Swift Intermediate Language (SIL). https://github.com/apple/swift/blob/main/docs/SIL.rst. [Online; accessed 18-March-2024].

[13] Davide Conficconi, Emanuele Del Sozzo, Filippo Carloni, Alessandro Comodi, Alberto Scolari, and Marco Domenico Santambrogio. 2022. An energy-efficient domain-specific architecture for regular expressions. *IEEE Transactions on Emerging Topics in Computing* 11, 1 (2022), 3–17. https://doi.org/10.1109/TETC.2022.3157948.

[14] Russ Cox. 2007. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). http://swtch.com/rsc/regexp/regexp1.html

[15] João Paulo Cardoso de Lima, Marcelo Brandalero, Michael Hübner, and Luigi Carro. 2021. STAP: An Architecture and Design Tool for Automata Processing on Memristor TCAMs. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 18, 2 (2021), 1–22. https://doi.org/10.1145/3450769.

[16] 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* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8.

[17] 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.* 533–545. https://doi.org/10.1145/2830772.2830809.

[18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking:{SmartNICs} in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone.

[19] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngAP: Non-blocking Large-scale Automata Processing on GPUs. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3617232.3624848

[20] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 1–12. https://doi.org/10.1109/MICRO.2016.7783747.

[21] Lei Gong, Chao Wang, Haojun Xia, Xianglan Chen, Xi Li, and Xuehai Zhou. 2022. Enabling fast and memory-efficient acceleration for pattern matching workloads: The lightweight automata processing engine. *IEEE Trans. Comput.* 72, 4 (2022), 1011–1025. https://doi.org/10.1109/TC.2022.3187338.

[22] Google. 2020. Google re2. https://github.com/google/re2.

[23] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. 2012. On the locality of codeword symbols. *IEEE Transactions on Information theory* 58, 11 (2012), 6925–6934. https://doi.org/10.1109/TIT.2012.2208937

[24] Yifei He, Artur Podobas, Måns I Andersson, and Stefano Markidis. 2022. FFTc: An MLIR Dialect for Developing HPC Fast Fourier Transform Libraries. In *European Conference on Parallel Processing.* Springer, 80–92. https://doi.org/10.1007/978-3-031-31209-0_6.

[25] Sara Hooker. 2021. The hardware lottery. *Commun. ACM* 64, 12 (2021), 58–65. https://doi.org/10.1145/3467017.

[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA. https://dl.acm.org/doi/10.5555/1196416.

[27] W-m W Hwu and Pohua P Chang. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture.* 242–251. https://doi.org/10.1145/74925.74953

[28] Zsolt István, David Sidler, and Gustavo Alonso. 2016. Runtime parameterizable regular expression operators for databases. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on.* IEEE, 204–211. https://doi.org/10.1109/FCCM.2016.61.

[29] Tian Jin, Gheorghe-Teodor Bercea, Tung D Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. 2020. Compiling onnx neural network models using mlir. *arXiv preprint arXiv:2008.08272* (2020). https://arxiv.org/abs/2008.08272.

[30] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM computer communication review* 36, 4 (2006), 339–350. https://doi.org/10.1145/1151659.1159952.

[31] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308.

[32] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why gpus are slow at executing NFAs and how to make them faster. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 251–265. https://doi.org/10.1145/3373376.3378471.

[33] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1 (2023), 1–27. https://doi.org/10.1145/3579453.

[34] Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, and Simone Campanoni. 2024. Representing Data Collections in an SSA Form. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 308–321. https://doi.org/10.1109/CGO57630.2024.10444817.

[35] 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. https://doi.org/10.1145/3079079.3079100.

[36] OISF. 2024. Suricata - Open Source Network Analysis and Threat Detection Software. https://suricata.io/.

[37] Marc Paolieri, Ivano Bonesana, and Marco Domenico Santambrogio. 2008. An adaptable FPGA-based System for Regular Expression Matching. (2008). https://doi.org/10.1145/1403375.1403681.

[38] Terence Parr. 2013. The definitive ANTLR 4 reference. *The Definitive ANTLR 4 Reference* (2013), 1–326. https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/.

[39] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D Santambrogio. 2021. Cicero: A domain-specific architecture for efficient regular expression matching. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–24. https://doi.org/10.1145/3476982.

[40] Karl Pettis and Robert C Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 16–27. https://doi.org/10.1145/93548.93550

[41] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. 2021. Scalable FSM parallelization via path fusion and higher-order speculation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 887–901. https://doi.org/10.1145/3445814.3446705.

[42] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. Microspec: Speculation-centric fine-grained parallelization for fsm computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 221–233. https://doi.org/10.1145/2967938.2967965.

[43] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 2020. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE.

[44] Rust Foundation. 2024. Rust Compiler Development Guide - Overview of the compiler. https://rustc-dev-guide.rust-lang.org/overview.html#intermediate-representations. [Online; accessed 18-March-2024].

[45] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. 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*. 219–234. https://doi.org/10.1145/3373376.3378459.

[46] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 403–415. https://doi.org/10.1145/3035918.3035954

[47] Andrea Somaini, Filippo Carloni, Giovanni Agosta, Marco D Santambrogio, and Davide Conficconi. 2025. Artifact Repository of Combining MLIR Dialects with Domain-Specific Architecture for Efficient Regular Expression Matching. https://doi.org/10.5281/zenodo.13345346

[48] 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*. 259–272. https://doi.org/10.1145/3123939.3123986.

[49] Mingqian Sun, Guangwei Xie, Fan Zhang, Wei Guo, Xitian Fan, Tianyang Li, Li Chen, and Jiayu Du. 2024. PTME: A Regular Expression Matching Engine Based on Speculation and Enumerative Computation on FPGA. *ACM Transactions on Reconfigurable Technology and Systems* (2024). https://doi.org/10.1145/3655626

[50] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387.

[51] Josep Torrellas, Chun Xia, and Russell L Daigle. 1998. Optimizing the instruction cache performance of the operating system. *IEEE Trans. Comput.* 47, 12 (1998), 1363–1381. https://doi.org/10.1109/12.737683

[52] 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 *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 461–472. https://doi.org/10.1109/MICRO.2012.49.

[53] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, et al. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–12. https://doi.org/10.1109/IISWC.2016.7581271

[54] J. Wadden, T. Tracy, E. Sadredini, L. Wu, C. Bo, J. Du, Y. Wei, J. Udall, M. Wallace, M. Stan, and K. Skadron. 2018. AutomataZoo: A modern automata processing benchmark suite. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 13–24. https://doi.org/10.1109/IISWC.2018.8573482.

[55] Xuan Wang, Lei Gong, Jing Cao, Wenqi Lou, Weiya Wang, Chao Wang, and Xuehai Zhou. 2023. HAP: A spatial-von neumann heterogeneous automata processor with optimized resource and IO overhead on FPGA. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 185–196. https://doi.org/10.1145/3543622.3573190.

[56] 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 *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang.

[57] Xiang-Yu Wang, Jian-Yu Shen, and Shih-Wei Liao. 2023. Accelerating Halide framework by automatic Affine scheduling using the stochastic optimization algorithm on MLIR. *Research Square Preprint* (2023). https://doi.org/10.21203/rs.3.rs-3274775/v1.

[58] Jackson Woodruff and Michael FP O'Boyle. 2021. New regular expressions on old accelerators. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 343–348. https://doi.org/10.1109/DAC18074.2021.9586095.

[59] Haojun Xia, Lei Gong, Chao Wang, Xianglan Chen, and Xuehai Zhou. 2021. Lap: A lightweight automata processor for pattern matching tasks. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 844–849. https://doi.org/10.23919/DATE51398.2021.9474249.

[60] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8. https://doi.org/10.23919/FPL.2017.8056759.

[61] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. 2016. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials* 18, 4 (2016), 2991–3029. https://doi.org/10.1109/COMST.2016.2566669.

[62] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 1083–1100. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[63] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the" embarrassingly sequential" parallelizing finite state machine-based computations through principled speculation. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 543–558. https://doi.org/10.1145/2541940.2541989.

[64] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 129–140. https://doi.org/10.1145/2145816.2145833.

Received 2024-05-30; accepted 2024-07-22