CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching

DANIELE PARRAVICINI, Politecnico di Milano, Italy
DAVIDE CONFICCONI, Politecnico di Milano, Italy
EMANUELE DEL SOZZO, Politecnico di Milano, Italy
CHRISTIAN PILATO, Politecnico di Milano, Italy
MARCO D. SANTAMBROGIO, Politecnico di Milano, Italy

Regular Expression (RE) matching is a computational kernel used in several applications. Since RE complexity and data volumes are steadily increasing, hardware acceleration is gaining attention also for this problem. Existing approaches have limited flexibility as they require a different implementation for each RE. On the other hand, it is complex to map efficient RE representations like non-deterministic finite-state automata onto software-programmable engines or parallel architectures.

In this work, we present CICERO, an end-to-end framework composed of a domain-specific architecture and a companion compilation framework for RE matching. Our solution is suitable for many applications, such as genomics/proteomics and natural language processing. CICERO aims at exploiting the intrinsic parallelism of non-deterministic representations of the REs. CICERO can trade-off accelerators' efficiency and processors' flexibility thanks to its programmable architecture and the compilation framework. We implemented CICERO prototypes on embedded FPGA achieving up to 28.6× and 20.8× more energy efficiency than embedded and mainstream processors, respectively. Since it is a programmable architecture, it can be implemented as a custom ASIC that is orders of magnitude more energy-efficient than mainstream processors.

CCS Concepts: • **Computer systems organization → Embedded hardware**; **Multicore architectures**; *Reconfigurable computing*.

Additional Key Words and Phrases: Domain-Specific Architecture, Regular Expressions, Non-deterministic Automata, Energy Efficiency

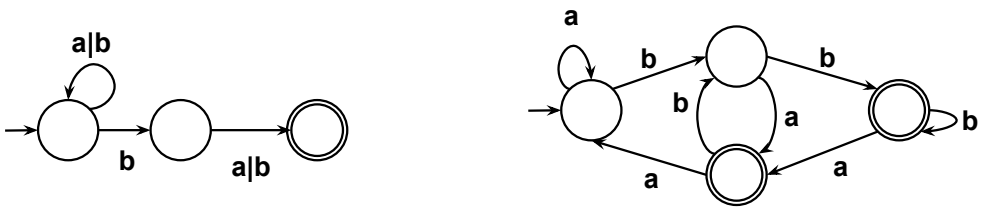
ACM Reference Format:
Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D. Santambrogio. 2021. CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2021), 24 pages. https://doi.org/10.1145/3476982

1 INTRODUCTION

Many applications rely on determining whether a string obeys a specific text-based *pattern*. A Regular Expression (RE) is a compact and specialized expressive language used to describe such

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2021.
Authors' addresses: Daniele Parravicini, Politecnico di Milano, Milano, Italy, daniele.parravicini@mail.polimi.it; Davide Conficconi, davide.conficconi@polimi.it, Politecnico di Milano, Milano, Italy; Emanuele Del Sozzo, emanuele.delsozzo@polimi.it, Politecnico di Milano, Milano, Italy; Christian Pilato, christian.pilato@polimi.it, Politecnico di Milano, Milano, Italy; Marco D. Santambrogio, marco.santambrogio@polimi.it, Politecnico di Milano, Milano, Italy.

1539-9087/2021/1-ART1 $15.00
https://doi.org/10.1145/3476982



(a) Non-deterministic Finite-state Automaton. (b) Deterministic Finite-state Automaton.

Fig. 1. Different representation of the RE "(a|b)*b(a|b)".

patterns. RE matching determines whether a sequence of input characters belongs to the set of strings described by the pattern. If so, we say the string is "accepted" by the RE.

RE matching is an essential kernel [2] for traditional computer security [32, 41] and database queries [19, 31] but also for novel domains such as natural language processing [35, 42], and genome-protein matching [8, 36]. The literature contains different algorithms to tackle RE matching. The first approach comes from theoretical computer science and employs deterministic finite-state automata (DFA) [17]. Such procedure requires only to keep track of the current state and moves into a new state every time it reads a character. However, some REs intrinsically carry a certain level of non-determinism. For example, the RE "(abbb|abab)" describes two alternative patterns that both start with the sub-string "ab". Therefore, the matching process does not know which pattern is matching until it evaluates the last two characters (either "bb" or "ab"). Even though it is always possible to move from non-deterministic finite-state automaton (NFA) into a deterministic version via the power-set construction algorithm, this conversion can exponentially increase the number of states. Figures 1a and 1b show a simple example where the NFA representation (Figure 1a) requires less states. When using an NFA, we need to adapt the matching algorithm to manage the states with alternative paths associated with the same character. A recursive implementation selects an alternative and, if wrong, it backtracks to the most recent "decision state" to evaluate a different path. This approach also requires reverting the portion of the string that was processed in the wrong path. In this way, the backtracking algorithm is simple but requires processing the input string multiple times. In the worst case, if the string does not match the RE, the algorithm must try all possible execution paths, leading to an exponential execution time [9]. An alternative approach with linear execution time has been proposed by Thompson [33] and used by Google in RE2 [11], a RE software library, which is in use in many Google products like BigQuery[1] and Google Suite[2].

Despite significant algorithmic improvements, software solutions cannot keep pace with the increasing size of the processed data (either input strings or REs). For this reason, hardware acceleration is a valid alternative for computationally-intensive kernels such as those for RE matching [3, 4, 7, 14, 26, 29, 31, 40]. In this scenario, reconfigurable FPGA devices represent a viable solution to boost the matching process while keeping a low energy profile. FPGAs can achieve a throughput of 100Gbps during intrusion prevention while a CPU with 250 cores is limited only to 400Mbps [41] (almost 250× of improvement). So, FPGAs can be used to implement specialized energy-efficient RE engines, while the device or part of it can be turned off when unused [16, 18]. However, since fixed-function accelerators embed custom RE matching logic for a given set of REs, they cannot be applied for other patterns, limiting the solution flexibility. Indeed, this approach requires to re-synthesize the logic for each new RE.

[1]https://cloud.google.com/bigquery/docs/reference/standard-sql/string_functions?hl=it#regexp_contains
[2]https://support.google.com/docs/answer/62754#regular_expression4

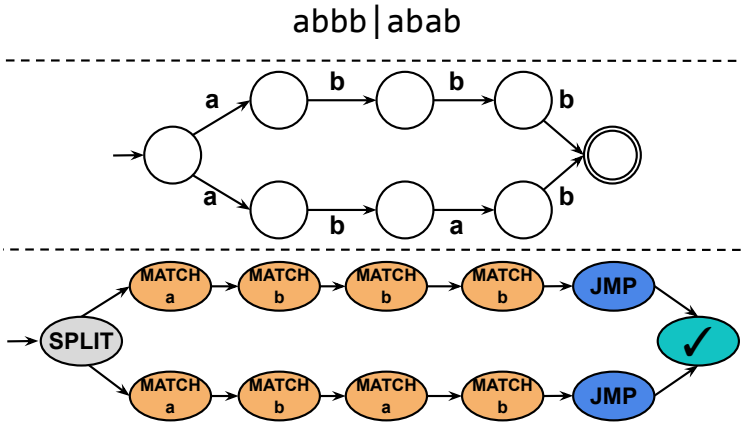


Fig. 2. From a RE "abbb|abab" (top) to its NFA (mid) and, finally, to CICERO instructions (bottom).

To overcome these limitations, we propose CICERO, a complete solution based on **domain-specific programmable engines** for RE matching. CICERO includes a *domain-specific architecture* for RE matching where each engine's execution model is based on Thompson's approach and a *compilation framework* to create the programming code of such engines. Indeed, given the input REs, the CICERO compiler translates it into our architecture machine code based on a simplified Instruction Set Architecture (ISA). Moreover, it applies optimizations to reduce the code size (i.e., number of instructions) and extract more hardware parallelism. Then, the CICERO engine executes such instructions while processing the input string. To exploit more hardware parallelism, we also describe a parallel architecture composed of multiple engines, evaluating two alternative interconnection topologies. CICERO combines the efficiency of specialized hardware accelerators and the flexibility of general-purpose processors.

After introducing the problem of RE matching (Section 2), we present our main contributions:

- the ISA (Section 3) and the associated compiler (Section 4) to generate the RE matching instruction to be executed;
- the CICERO architecture and the associated architectural optimizations to exploit the intrinsic parallelism of non-deterministic REs (Section 5);
- a comprehensive validation of the CICERO overall solution with embedded FPGA prototypes and a comparison with embedded (ARM) and mainstream (Intel) processors (Section 6).

We evaluated our single- and multi-engine FPGA prototypes using real benchmarks from the open-source AutomataZoo benchmark suite [36]. We obtained excellent results both in terms of performance and energy efficiency: our CICERO architecture is 28.6× and 20.8× more energy-efficient than ARM and Intel processors, respectively. Given the flexibility of our architecture, we present performance estimations for an ASIC implementation of CICERO that can potentially outperform even high-end, commercial processors in energy efficiency on the complete benchmarks.

2 BACKGROUND ON REGULAR EXPRESSION MATCHING

This work aims at implementing an architecture for RE matching with two conflicting goals: 1) provide the efficiency of hardware accelerators thanks to specialization and parallel execution, and 2) offer the flexibility and reusability of general-purpose processors. For this reason, we first analyze existing algorithms for the RE matching since they have a huge impact on the performance of the

architecture. In the rest of this section, we use the RE "abbb|abab" (see the top part of Figure 2) as a running example.

Software libraries, like the ones available in programming languages such as Python2.4 or Perl5.8, divide the RE into sub-expressions until the matching problem becomes manageable and then use a backtracking algorithm to evaluate the alternative paths [9].

> **Example**: When applied to our example, they first divide the pattern into two sub-expressions, namely "abbb" and "abab". This decomposition can be easily represented as the non-deterministic finite-state automaton shown in the mid part of Figure 2. When a character (e.g., 'a') is compatible with two or more sub-expressions (e.g., "abbb" and "abab"), the machine considers one sub-expression (e.g., "abbb"), keeping track of the possible alternatives. If the machine does not match the string (i.e., the RE does not *accept* the string), it needs to backtrack to the most recent alternative and consider other paths (e.g., sub-expression "abab").

This process repeats until the machine either accepts the string in one of the paths or rejects it after exploring all the alternatives without finding a match. This algorithm becomes extremely inefficient when the number of alternative paths grows exponentially. Consider the case of RE-based Denial of Services [27], where we may need to match the string "aaaa" and the RE "a?a?a?a?aaaa." Indeed, the backtracking approach has a time complexity of $O(2^m)$, where m is the number of alternative paths to be evaluated [9].

Thompson observed that the backtracking algorithms are inefficient mostly because they need to scan the input string multiple times [33]. To avoid this, he built a Virtual Machine (VM) implementing a multi-threaded execution model. The VM can handle simple operations like scheduling a thread, executing a thread for a certain time quantum, or a finite number of steps. Each thread executes code to match a single RE expression or sub-expression. Whenever a parallel or non-deterministic path occurs, it spawns additional threads to explore the alternatives with a breadth-first approach. In this way, the new threads do not require to analyze again parts of the string already elaborated. Moreover, we can avoid saving the whole *thread context* by executing the threads in "lockstep": all of them process the same character of the string and then move forward to the next [10].

> **Example**: When applied to our example with the input string "abbb", Thompson's algorithm creates two threads for the sub-expressions "abbb" and "abab", respectively. All threads process the same input character in parallel, so they do not need to look backward in the string. After processing the first two characters 'a' and 'b', the third character of the input string is 'b', while the second thread is expecting the character 'a'. So it fails the matching and stops. The other thread can continue, consumes the remaining characters, and accepts the string.

This approach offers an execution time that grows linearly with the number of string characters, while the degree of alternatives in the RE impacts the number of running threads per character. Similarly to this work, we address alternatives and non-determinism using parallel hardware execution flows similar to Thompson's threads. In particular, we aim at executing the threads with domain-specific engines that allow us to process the alternatives with the efficiency of hardware accelerators. To trade-off specialization and flexibility, our engines are domain-specific processors based on an Instruction Set Architecture (ISA) tailored to RE matching. Our architecture uses multiple execution flows that process the same current character in parallel with RE-specific instructions. We also provide a compiler-based framework to convert REs into such instructions.

> **Example**: The bottom of Figure 2 shows the instructions flow generated to match the RE "(abbb|abab)". Each node represents a specific instruction that can either 1) proceed

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

Instruction Class	Instruction	Operand	Description
Matching	MatchAny	-	PC+1 and cc+1.
	Match(OP)	Character	Compares OP with *cc. In case of match, PC+1 and cc+1.
	NoMatch(OP)	Character	Compares OP with *cc. In case of *no* match, *only* PC+1.
Control Flow	Split(OP)	Target Addr.	Produces two parallel execution flow: the first continues with the instruction that follows immediately after (PC+1), while a new one starts at the target address (OP).
	JMP(OP)	Target Addr.	Unconditional Jump to the target address OP.
Acceptance	Accept	-	Accepts if at the end of the string.
	AcceptPartial	-	Accepts at any point in the string.

to the next one to continue the match or 2) stop the analysis when the input string is not accepted. We also have specific instructions to spawn "threads" (Split).

In the following sections, we describe the main components of our approach. First, we describe the ISA (Section 3), which is the tailored software/hardware interface for creating domain-specific RE matching engines. Then, we present the compiler devised for lowering REs into our CICERO-compatible instructions (Section 4). Finally, we show the architectural details of the CICERO engine and its optimizations (Section 5).

3 CICERO INSTRUCTION SET ARCHITECTURE

The CICERO ISA takes inspiration from the basic operations described by [9], which employs parallel *threads* working in lockstep on a sequence of characters [10], similar to a breadth-first exploration. For this reason, the CICERO engine must be capable of performing simple operations such as matching a character, creating threads, adapting the instruction flow, or ending the execution. For example, thread creation requires generating references to the instructions indicating the alternative execution paths' beginning.

Each CICERO instruction consists of 16 bits and is divided into two parts: an *opcode* (3 bits), which identifies the instruction type, and an *operand* (13 bits). The operand may have a different interpretation based on the opcode. All instructions are stored in memory and identified by an address. The execution of each instruction takes as input a character of the string and determines the subsequent instructions to continue the matching. The ISA is divided into three main classes, as shown in Table 1: *matching* (MatchAny, Match, and NoMatch instructions), *control flow* (Split and JMP instructions), and *acceptance* (Accept and AcceptPartial instructions).

Matching instructions consider the current string character. A MatchAny instruction applies when the RE contains a wildcard (e.g., '.'). It consumes any character and moves to the next instruction. The Match instruction compares the current character with the instruction operand. If the two characters match, we move to the next instruction in the sequence. Otherwise, no further instruction is processed for this part of the flow. The NoMatch instruction represents the dual of Match operations. Indeed, it checks if the operand and the current character do not correspond. In that case, it moves to the next instruction in the sequence without consuming the current character.

Otherwise, if the characters match, it does not need to consider any further instruction, and this part of the flow is over. In this way, it is possible to check a single character multiple times (e.g., "`[^abc]`" can be represented by a sequence of three `NoMatch` instructions followed by a `MatchAny`).

Control flow instructions change the next instructions to be executed and are the basis for creating the multiple execution flows that process the alternative non-deterministic paths. A `JMP` instruction unconditionally sets a new arbitrary point to continue the execution flow. A `Split` instruction creates parallel execution flows (or threads). The first flow continues with the next instruction, while the second one starts at the address targeted by the operand.

Acceptance instructions conclude the RE matching algorithm. The `AcceptPartial` instruction affirmatively concludes the RE matching at any point of the input string, while the `Accept` instruction concludes only at the end of the string.

4 CICERO COMPILER

Since CICERO instructions are stateless, we can not take advantage of state-of-the-art algorithms, such as register allocation, available in highly optimized compiler frameworks. Therefore, we built from scratch our own custom compiler that translates REs into executable binaries, according to Section 3 format. The compiler has a standard structure with three parts: front-end, mid-end, and back-end.

The front-end elaborates the input RE with an LR parser [21] and produces an abstract syntax tree. The parser does not support any back-reference operator since the expressive power required exceeds the regular languages [9]. At that point, the front-end manipulates the abstract syntax tree to produce our architecture-agnostic intermediate representation (IR).

The mid-end applies a sequence of architecture-independent IR optimizations to enhance the RE matching code, reduce the code size, and improve parallelism. Our set of optimizations includes *code restructuring* and *redundant instruction collapsing*. These optimizations mostly target sequences of `Split` instructions. The *code restructuring* reorganizes a sequence of `Split` instructions into a tree with minimal height, while *redundant instruction collapsing* merges equivalent instructions.

Figure 3 shows an example of *code restructuring*. This optimization balances the number of instructions to reduce the abstract syntax tree height. Indeed, the left side of Figure 3 shows that the longest instruction path is three (up to the `Match 'd'`), while the path up to the `Match 'a'` contains a single instruction. Therefore, in the worst case, i.e., when the current character is 'd,' the `Match` 'd' execution happens after at least four instructions.

On the right-hand side, we can see the code after the compiler applies code restructuring. In this case, the longest path to each `Match` is equal to two. Moreover, considering a parallel architecture that can execute numerous paths simultaneously, this optimization will decrease the overall execution time. For instance, assuming each instruction is executed in a unit of time, the worst execution time with four cores is three time units.

The second optimization, i.e., *redundant instruction collapsing*, aims at identifying and merging equivalent instructions in the code. This compression reduces both code size and execution time. This is a common situation in case of non-deterministic representations, like the one in the bottom of Figure 2 where two equivalent operations (i.e., `Match 'a'`) follow a `Split` instruction. The compiler repeats this operation until a fixed point to compress equivalent CICERO code parts. Consider the example in Figure 2. We can anticipate the `Match 'a'` operations before the `Split` and collapse them into a unique equivalent instruction without modifications on the code semantics. Figure 4 shows an example of how this optimization reduces the size of the code, while Figure 6 shows the result of the optimizations to the example in Figure 2.

The back-end emits the actual machine instructions to be executed by the CICERO architecture. We perform code placement in memory and, after that, we apply another *redundant instruction*

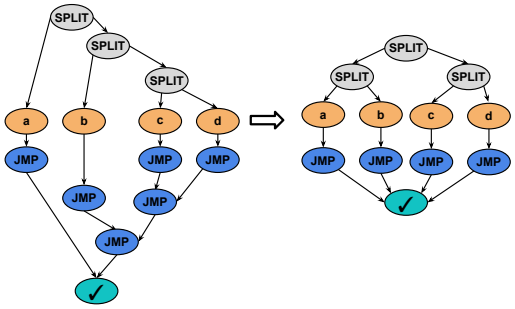


Fig. 3. Example of *code restructuring* optimization applied to the RE "`(a | (b | (c | d)))`".

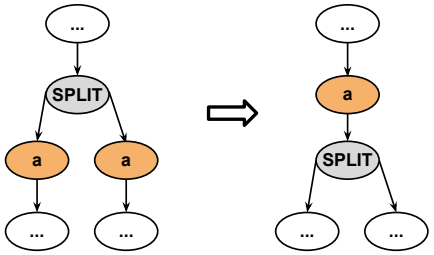


Fig. 4. Example of *code restructuring* optimization applied to the RE "`(a...|a...)`".

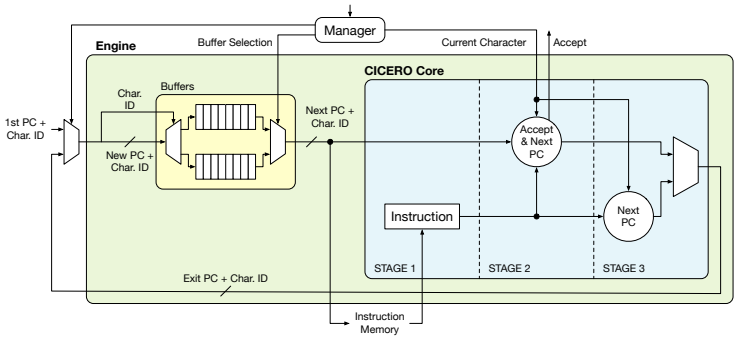


Fig. 5. CICERO base engine architecture.

collapsing to `JMP` instructions. Since a chain of `JMP` instructions is inefficient (e.g., left-hand side Figure 3), we replace this chain with a unique `JMP`. In this way, we reduce the number of subsequent `JMP` instructions to be processed.

5 CICERO ARCHITECTURE

This section describes the fundamental building blocks of our CICERO architecture. First, we describe the CICERO base engine that elaborates the instructions over a single character at a time (Section 5.1). Then, we increase the degree of parallelism in the CICERO engine enabling the ability of processing multiple characters (Section 5.2). Finally, we aim at further increasing the parallelism in instruction processing with a multi-engine architecture (Section 5.3). In this context, we also explore two different interconnection topologies that offer different scalability models.

5.1 CICERO Base Engine

The fundamental block of the CICERO architecture is the CICERO engine, which processes the RE instructions with a minimal amount of resources. The CICERO engine has two main components: the *CICERO Core* and the *Buffers*. As shown in Figure 5, we combine the CICERO engine with an *Instruction Memory* and a *Manager* module to obtain a platform that executes the instructions described in Section 3. The CICERO RE matching code requires executing all instructions related to a string character before moving to the next one until the string is either accepted or rejected.

The CICERO Core is a 3-stage pipelined processor that executes the instructions stored in the *Instruction Memory*. The *Program Counter (PC)* refers to this memory and indicates the next

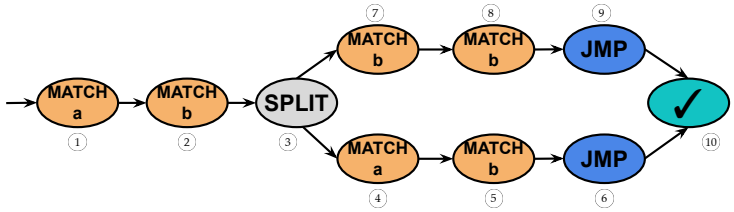


Fig. 6. Optimized CICERO instructions for example RE in Figure 2.

string: a b a b c d

		Clock cycles														
PC	Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	...
1	match(a)	S1	S2														
2	match(b)			S1	S2												
3	split(7)					S1	S2	S3									
4	match(a)							S1	S2								
5	match(b)										S1	S2					
6	jmp(10)												S1	S2			
7	match(b)								S1	S2							
8	match(b)																
9	jmp(10)																
10	accept_partial														S1	S2	

Fig. 7. Execution timing diagram of CICERO code. S1, S2, and S3 indicate the stages of CICERO core.

instruction to be executed[3]. The first pipeline stage uses the PC signal to address the *Instruction Memory* and loads the next instruction. Both the remaining stages decode and execute the instruction to either indicate the next instruction (by producing a new PC) or conclude the RE matching algorithm by raising the *Accept* signal. The CICERO Core requires these two additional stages since the *Split* instruction produces two PCs (corresponding to the beginning of the two threads), while the engine has only one output port. Clearly, the third stage is executed only for this type of instruction. The output port includes the PC and one extra bit to specify whether the thread has to continue with the current character or proceed with the next. This bit redirects the CICERO core output into the proper first-in-first-out (FIFO) of the *Buffers*. Furthermore, adding a multiplexer to the CICERO core input allows us to insert the first thread, i.e., first instruction to be executed and first character to be processed. The *Buffers* are composed of two FIFOs (or more as in Section 5.2). We employ them as a ping-pong buffer that contains instructions related to the current character and the other PCs for the following one.

The *Manager* selects from which FIFO the CICERO Core gets the following operation to be processed. Therefore, the *Manager* alternates the content of the FIFOs among current character PCs and following character PCs. Moreover, the *Manager* controls the overall execution of the RE matching algorithm. Once the CICERO Core has consumed all instructions related to the current character, the *Manager* provides the new character and changes the FIFO for the CICERO Core. The FIFO that is currently empty becomes the FIFO for the new next character. When the CICERO Core reaches an *Accept* instruction, the CICERO engine notifies that the string is accepted. Otherwise, when both queues are empty, the *Manager* concludes that the string does not match the RE.

[3]In the following, we use the term Program Counter (PC) and instruction, interchangeably. Indeed, the PC is the memory reference to the corresponding instruction to be executed.

Running Example. Consider the RE "abbb|abab" in Figure 2 and the corresponding optimized CICERO code in Figure 6, together with the input string "ababcd". The engine initialization starts with the first thread, which has PC equal to 1, and the current character is the first 'a'. The first instruction is a `Match` 'a', and it is stored in the first FIFO (let us call it FIFO 0), while the other FIFO (FIFO 1) is empty. CICERO fetches the first input character, i.e., 'a'. Then, it executes the first instruction (i.e., Match 'a'), consumes the first 'a' of the input string, and produces the reference to the second instruction (i.e., Match 'b'). Since this instruction refers to the next character, the *Manager* adds it in the FIFO 1. FIFO 0, which is the FIFO of the current character, i.e., 'a', is now empty since all corresponding instructions are executed; hence, we can move to the following character of the input string, i.e., the first 'b', and switch to FIFO 1. CICERO executes the second instruction (i.e., `Match` 'b') and produces the `Split` instruction, i.e., number 3 in Figure 6. Given that there are no more instructions for the current character 'b', we move to the following one, i.e., the second 'a', and swap FIFO 1 for FIFO 0. The core executes the `Split` instruction and produces two instructions: instruction 4 (`Match` 'a') and instruction 7 (`Match` 'b') . Since both refer to the current character, the *Manager* adds them in the current FIFO, i.e., FIFO 0. CICERO starts executing instructions 4 and 7, but only instruction 4 (i.e., `Match` 'a') matches and produces a new instruction , i.e., number 5 `Match` 'b', which the Manager places on FIFO 1. We move ahead of one character in the string, i.e., the last 'b', and switch to FIFO 1. CICERO executes instruction 5, i.e., `Match` 'b', which produces the `JMP` instruction, number 6. As for the previous case, there are no more instructions referring to the current character, and we move forward in the input string fetching the 'c' character, and we swap the FIFOs, i.e. FIFO 1 for FIFO 0. Since the `JMP` does not read any character, instruction 10 is pushed into the current FIFO, i.e., FIFO 0. Finally, CICERO takes instruction 10 from the queue and executes the `AcceptPartial`, ending the overall matching procedure. Figure 7 shows the execution timing diagram of the described running example.

As the reader can see from this diagram, there are no instructions with different colors (i.e., referring to different characters) executed in the same clock cycle, even though they may be ready to execute. For instance, instruction 5 is ready to be executed at clock cycle 8, however, its execution is postponed at the end of all instruction related to third character. This execution delay will play a crucial role in the following section of the manuscript.

In standard processors, supporting threads requires that the thread context is saved when it moves to the idle state and reloaded once the thread is resumed. In CICERO, the threads refer to the parallel flows processing the current character. Since the CICERO Core does not produce any temporary values nor stores value in a register file, the CICERO *context* includes only the PC and the current string pointer. The current character is shared among all active threads; hence, the PC provides enough information to restart the corresponding thread.

5.2 CICERO Multi-Character Engine

The engine described in the previous section has an architecture able to process a single character with multiple threads working in lockstep on a sequence of instructions (i.e., it consumes a character for each possible instruction flow). In this way, CICERO works in a breadth-first style that consumes a character at a time without backtracking, similar to a single-stride NFA (i.e., single character [40]), with two buffers. However, though it has noteworthy abilities, the single character consumption rate limits the achievable throughput of characters processed per second.

Though we adopt an algorithmic approach that is more efficient than backtracking, the system considers all the current character's execution paths before moving to the next. The effectiveness of this approach is high whenever compared to backtracking or processing workloads containing several parallel sub-expression to evaluate. As the engine utilization increases, we extract the best from CICERO. However, this is not always the kind of workloads in the RE world [35], and

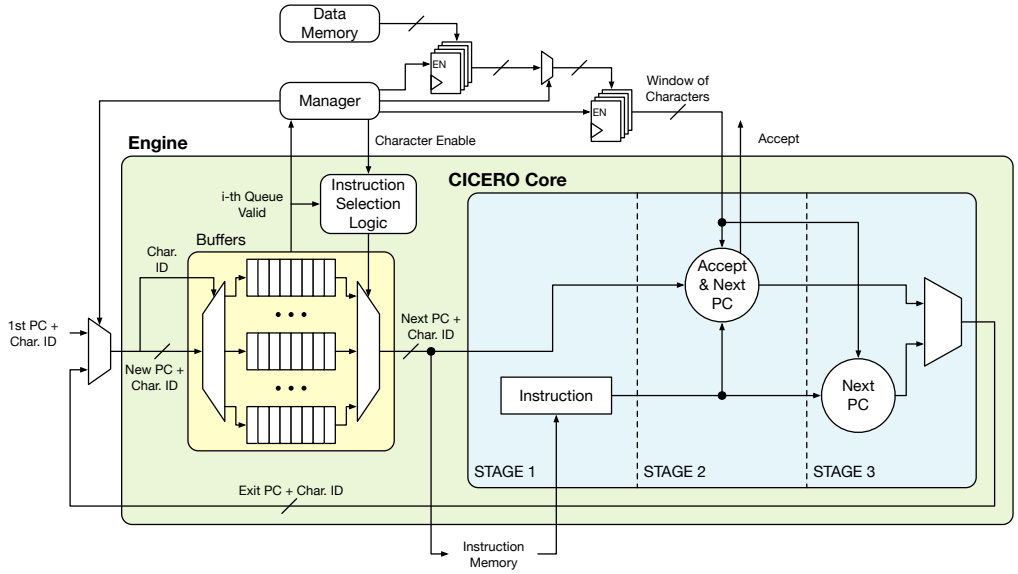


Fig. 8. CICERO base engine extended with multi-character support.

sequential execution is inevitable [2]. There are two possible approaches: dealing with thread accumulation in the next character buffer or increasing the character processing rate.

Considering the second approach, we can enhance the architecture by analyzing a character window of 2^W characters, i.e., 2^W-stride NFA, with parallel threads in lockstep, as shown in Figure 8. In this way, we can keep code portability among different windows of engines (i.e., the modification is not visible at the ISA level), but we increase the engine character consumption rate that can now run on $2^W - 1$ parallel characters. However, the thread context has to be updated to keep track of the consumption pointer of the input string. CICERO handles this optimization by employing a W-bit ID, called `CC_ID`, that refers to the current character in the window analyzed. Moreover, whenever we encounter `Match` or a `MatchAny` instructions, we should update `CC_ID` to reflect the fact that we moved to the next character. Considering that the `CC_ID` is a natural number modulo 2^W, CICERO keeps the threads of the last character of the window in a non-ready state. Indeed, their execution might conflict with a thread with the same `CC_ID` that refers to a newer character.

For instance, consider the input string "abcde", $W = 2$, and a current window "abcd" within CICERO, with running threads with `CC_ID`'s 0, 1, 2, 3. If the thread with `CC_ID` 3 goes first and finds an instruction that consumes a character (e.g., `Match` or `MatchAny`), the thread `CC_ID` increases and become 0, i.e., $(3 + 1) \bmod 2^2$. At this point, this thread would wrongly target the 'a' in the buffer instead of the proper 'e'. For this reason, we stall threads related to the last character of the window that explains the minus one in the $2^W - 1$ parallel characters.

To summarize, the proposed optimization is a *sliding window* that the *Manager* handles as a circular buffer. Indeed, we need to add a FIFO for each character of the sliding window (i.e., if $W = 3$ we need a total of 8 FIFOs) to handle separate thread contexts and adopt a policy that executes always the oldest thread. An arbiter with programmable priority will let the proper instruction execute. Therefore, we can consider the CICERO base engine described in the previous section as a particular case with a window $W = 1$.

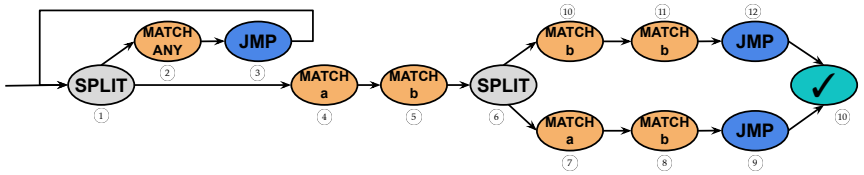


Fig. 9. CICERO code corresponding to ".*(abab|abbb)".

Finally, we need to account for the logic required to slide the window ahead, i.e., moving ahead of one character. Indeed, the enhanced architecture tracks the number of threads per `CC_ID` in flight in every architecture component (e.g., *Buffers*, CICERO Core, engine). Practically, there is a 2^W-bit wide bitmap that has an i^{th} active bit if there exists at least an active thread with `CC_ID` $= i^{th}$ in the architecture. The bitmaps are then combined with bitwise OR operations to hint the *Manager* on sliding the window or not. Indeed, if the character bit closest to the beginning of the window, i.e., the oldest one, is unset, the *Manager* fetches another character and slides the window.

Running Example. To better illustrate the mechanism behind CICERO with Multi-Character Engine and how it takes advantage of non-determinism, we consider an extension to the example of Section 5.1. Consider the RE ".*(abab|abbb)", which is shown in Figure 9 in the form of a CICERO code, and consider as input string "abaababd". We chose this RE because it also highlights how CICERO manages non-determinism conversely to a backtracking approach. Indeed, CICERO adopts a breadth-first like execution model that explores all the alternatives at the same level. The inherent non-determinism in the considered RE leads CICERO to execute instructions 1, 2, 3, and 4 for every character in the string to test the actual starting point of the matching procedure, which begins at instruction 4. Moreover, since CICERO instructions do not rely on state, it can start considering new instructions that are ready to execute while running instructions 1,2,3 and 4. The reader can appreciate a graphical representation of this effect in Figure 10, which compares the pipeline of Single-Character CICERO (i.e., $W = 1$), on the top, with a Multi-Character (i.e., $W = 2$) CICERO, at the bottom. This part of Figure 10 shows that by supporting up to four (i.e., 2^W, where $W = 2$) parallel characters, CICERO avoids waiting for the pipeline flush before processing the new character in the input string. For instance, instruction 6 can execute at clock cycle 7 (bottom of the Figure), instead of waiting for the end of the processing of character 'b' at clock cycle 12 (top of the Figure). Thanks to this improvement, we can increase the pipeline occupancy (in the example, we move from an instruction per clock of 11/17=64% to 16/18=88%). Consequently, the proposed optimization reduces execution times and increases the character processing rate.

5.3 CICERO Multi-Engine Architecture

In the previous sections, we described the base design of the CICERO engine together with the sliding window implementation. However, with our execution model we can exploit a further degree of parallelism related to the instructions of the threads. Since CICERO instructions do not have side effects, they can be safely executed in parallel by multiple CICERO engines to increase the parallelism. The parallel version of CICERO features multiple CICERO engines with a centralized *Manager* and a distributed *Load Balancing Infrastructure* as shown in Figure 11.

As discussed above, the *Manager* supplies the current character to the engines and makes decisions on the overall matching process. It decides when to move to the next string character (or slide the window), accept a RE if one of the engines notifies an accept signal, and reject the RE matching after consuming all the instructions. To support parallel execution, we add a private

string: a b a a b a b d

	W=1	Clock cycles																	
PC	Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	...
1	split(4)	S1	S2	S3					S1	S2	S3									
2	matchany			S1	S2						S1	S2								
3	jmp(1)						S1	S2							S1	S2				
4	match(a)				S1	S2						S1	S2							
5	match(b)							S1	S2											
6	split(10)													S1	S2	S3				
7	match(a)																S1	S2		
8	match(b)																			
9	jmp(13)																			
10	match(b)																			
11	match(b)																			
12	jmp(13)																			
13	accept_partial																			

string: a b a a b a b d

	W=2	Clock cycles																	
PC	Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	...
1	split(4)	S1	S2	S3				S1	S2	S3					S1	S2	S3			
2	matchany			S1	S2					S1	S2						S1	S2		
3	jmp(1)					S1	S2						S1	S2						
4	match(a)				S1	S2					S1	S2						S1	S2	
5	match(b)						S1	S2												
6	split(10)								S1	S2	S3									
7	match(a)											S1	S2							
8	match(b)															S1	S2			
9	jmp(13)																			
10	match(b)													S1	S2					
11	match(b)																			
12	jmp(13)																			
13	accept_partial																			

Fig. 10. Comparison of CICERO execution timing diagrams with Single character Engine (W=1) and with Multi-Character Engine (W=2). S1, S2, and S3 indicate the stages of CICERO core.

block-based instruction cache to each CICERO engine. Since the instruction memory is read-only, no coherency protocol is needed. If a cache miss occurs, a round-robin *Arbiter* regulates the access to the *Instruction Memory*.

The *Load Balancing Infrastructure* handles the thread's execution on different engines without affecting the critical path. Each CICERO engine features an instance of the *Load Balancing Infrastructure*. It consists of a *Station* and a latency insensitive *channel* [1]. At each clock cycle, the Station receives as input the engine output (i.e., the thread information composed of PC and a `CC_ID`), the number of instructions inside the local engine, and information from the nearby stations. At the same time, the *Station* obtains the expected latency of running an instruction that might flows to the next *Load Balancing* instances. Moreover, the *Station* can receive threads to execute from the

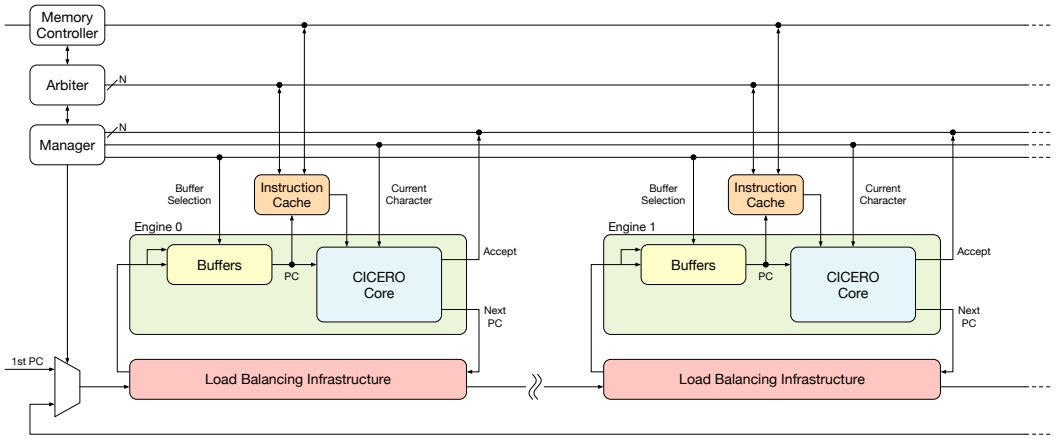
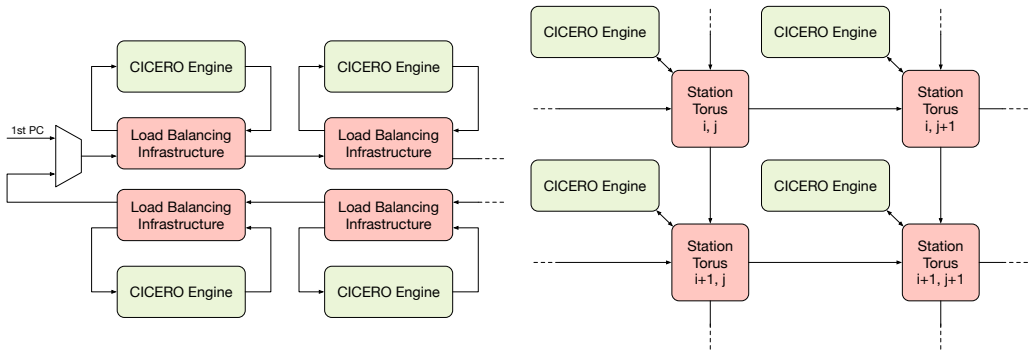


Fig. 11. An overview of CICERO multi-engine architecture showing the overall infrastructure that wraps CICERO engines.



(a) Ring topology.

(b) Torus topology.

Fig. 12. CICERO interconnection topologies.

previous stations. Based on the number of buffered instructions in the local engine (i.e., the local latency), and the latency coming from the next Station, each *Load Balancing Infrastructure* decides where to move the CICERO output and the threads by computing the minimum latency among the possible paths. As the last step, the *Load Balancing Infrastructure* computes its input latency as the minimum between the number of threads to execute in the CICERO engine and the latency coming from the next Station. The latency information is then adjusted to consider the number of threads in flight along the channel. To avoid any combinational path, registers separate the latency on every channel.

We devise two different topologies for the multi-engine architecture. The first topology is a ring where each engine connects with the other two engines at most, as in Figure 12a. The second topology is a torus where each engine connects with at most four other engines, as in Figure 12b. While the ring is a simple topology but with limited scalability, the torus ideally provides a more scalable interconnection topology since each engine has more alternative where to send the threads. In both cases, we need a multiplexer to initialize the multi-engine architecture with the first thread.

Considering the **ring topology** (Figure 12a), the *Station* modules and the *Load Balancing Infrastructure* ones are equivalent to the ones described in the previous section. *Station* modules are connected through latency-insensitive queues to form a ring. This protocol guarantees correct execution in all cases. In this way, we aim at evenly distributing the number of threads to elaborate across the engines.

For the **torus topology** (Figure 12b), we can reuse the ring topology's interconnection components to design an XY-distributed *Load Balancing* station (called *Torus Station*) on top of the *Ring Station*. The X_{in} input flows into a ring-based *Station*, and the link with CICERO engines remains the same. The first ring-based *Station*'s output then passes to the second ring-based *Station* via a latency-insensitive queue. In the second *Station*, we have the additional input, Y_{in}, which produces two outputs, namely X_{out} and Y_{out}. In this way, we can link the *Ring Stations* as in Figure 12b.

6 EXPERIMENTAL VALIDATION

We implemented CICERO in SystemVerilog with a standard AXI interface and created FPGA prototypes exploiting the Xilinx Vivado HLx 2019.2 toolchain. We targeted an embedded FPGA board, namely the Xilinx Ultra96v2 (Zynq Ultrascale+ MPSoC XCZU3EG A484), on which we employ the PYNQ framework [39]. We also generated an ASIC implementation targeting an open-source 32nm technology to evaluate a potential custom chip.

At first, we analyzed the impact of compiler and architectural optimizations on CICERO performance (Section 6.1). In particular, we measured the code size and clock cycle reductions that the implemented compiler optimizations enabled. Similarly, we investigated the matching time and energy efficiency benefits that the multi-character and multi-engine approaches provide when targeting FPGA with running frequency of 200MHz. Then, we compared our best FPGA prototype against Google RE2 [15], an optimized multi-threaded C++ library for RE, in terms of matching time and energy efficiency (see Section 6.2). Finally, we estimated the performance and area of the ASIC implementation (see Section 6.3).

In all the experiments, we used Protomata [28] and Brill [42] benchmarks from the AutomataZoo suite [36], which represent proteomics and natural language processing applications, respectively. We considered Protomata and Brill since they both belong to the family of "Regex" benchmarks of the original ANMLZoo suite [35]. Therefore, their RE representation is ready to use [36], and they target novel compelling research fields, i.e., bioinformatics and natural language processing. Moreover, we believe that these two benchmarks represent two opposite use cases: one more suitable to CICERO features, i.e., with a high number of alternatives (Protomata), against an unsuitable one with a wide variety of sequential REs (Brill). AutomataZoo reports an active set (i.e., the average number of active states in the NFA) of 712 for Protomata against 78 for Brill [36]. Indeed, most Protomata REs include many non-contiguous character sets to test. In this way, a generic architecture has to evaluate a more significant number of alternative paths/sub-expressions, and partial matches (part of the string matches the initial part of the RE) are more likely to happen. If adopting a backtracking approach, the target platform will most likely suffer from it and obtain poor performance on Protomata. Instead, Brill contains a mix of contiguous character sets and sequential matches. This second benchmark brings more advantages to traditional von Neumann architectures, which can handle these sets with simple arithmetic differences, sequential executions, and aggressive approaches similar to backtracking. We exclude other benchmarks since they either provide the automaton instead of the RE or contain unsupported features of non-regular languages such as backreferences. We evaluated these benchmarks on the same suite's input and applied the RE matching to at most 1,024 characters if the input string was bigger. We also combined some REs in the two benchmarks to increase the RE complexity. To do so, we used up to four operators '|' to create parallel alternatives. These combined REs increase the number of alternative

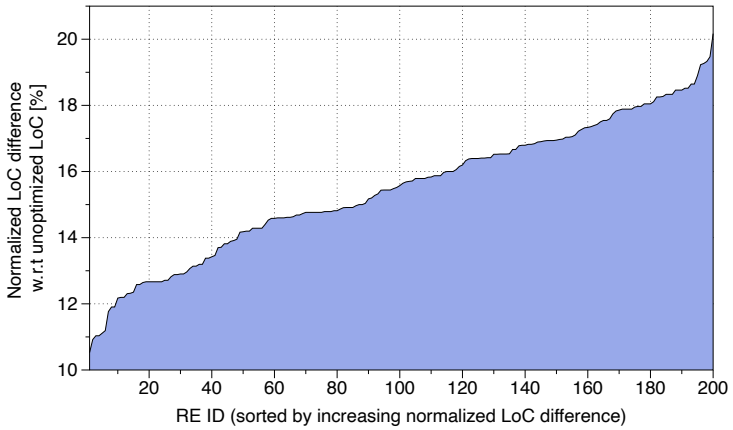


Fig. 13. Unoptimized vs optimized Lines of Code (LoC) difference normalized w.r.t. the unoptimized LoC size ($\frac{Unopt_{LoC}-Opt_{LoC}}{Unopt_{LoC}}$) on REs sampled from Protomata4, showing the improvements w.r.t. the original LoC.

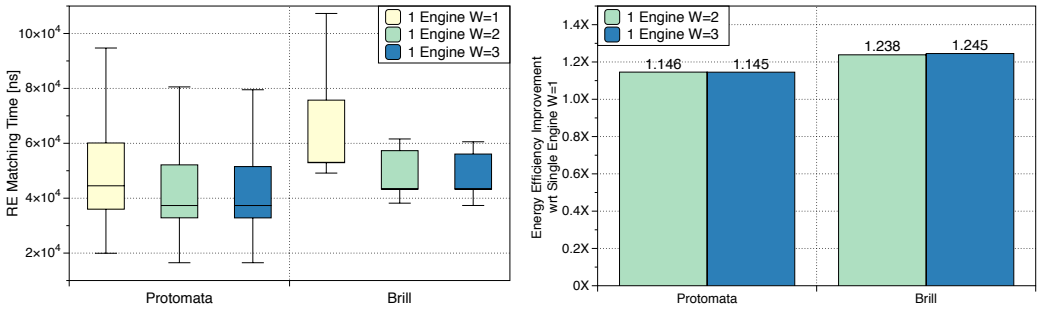
paths simultaneously active and provide a scenario where the final user aims to match a set of REs in a single input pass. We randomly sampled 1,000 REs from both Protomata and Brill, and combined four different random REs together in a combinatorial manner, i.e., providing all the possible combinations, as previously described. Then, we randomly sampled 200 combined REs from this new set of REs and 200 possible input strings from the original AutomataZoo. We will call these combined versions *Protomata4* and *Brill4* benchmarks.

Throughout the evaluation, we employ three different sets of tests for the considered benchmarks. The first one is a subset of REs and inputs randomly sampled with a uniform distribution from the original benchmarks. The second subset contains 200 REs randomly sampled with a uniform distribution from the combined benchmarks (i.e., Protomata4 and Brill4) to increase the parallelism degree and better highlight the benefits of a multi-engine CICERO. The third set comprises the complete benchmark tests as published by the suite authors to provide a fair comparison with other approaches employing established test suites.

6.1 Evaluation of Compiler and Architectural Optimizations

This Section describes the impacts of compiler and architectural optimizations on CICERO performance. First of all, we analyze the benefits that the compiler optimizations introduce in terms of code size and processing time (Section 6.1.1). Then, we evaluate how CICERO performance scales according to the character window (Section 6.1.2) and the number of engines (Section 6.1.3). Finally, we examine which is the most suitable interconnection topology for the CICERO multi-engine architecture (Section 6.1.4).

6.1.1 Compiler Optimizations. We start evaluating the impact of compiler optimizations on the code size. Figure 13 shows the reduction in terms of Lines of Code (LoC), or instructions, among the code sizes of the Protomata's REs compiled with and without the optimizations. In particular, we normalized the difference between unoptimized and optimized LoC by the unoptimized size ($\frac{Unopt_{LoC}-Opt_{LoC}}{Unopt_{LoC}}$). On average, the optimizations save 15.48% instructions for the Protomata4 combined REs, while the combined Brill4 has an average reduction of 1 instruction, and hence it is not plotted. Protomata code size reduction leads to a geometric mean (geomean) speedup of 1.3× compared to the unoptimized code.



(a) Weighted average distributions of matching times. (b) Geomean energy efficiency improvements.

Fig. 14. The effects on queue scaling on a CICERO single engine (experiments performed on 200 inputs and 200 REs sampled from Protomata and Brill benchmarks).

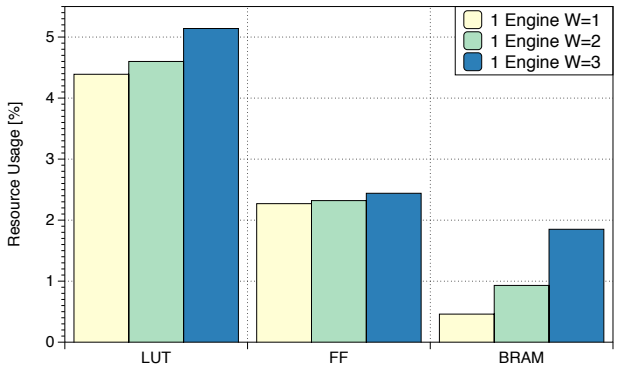


Fig. 15. Resource usage percentage of CICERO single engine, with different window sizes.

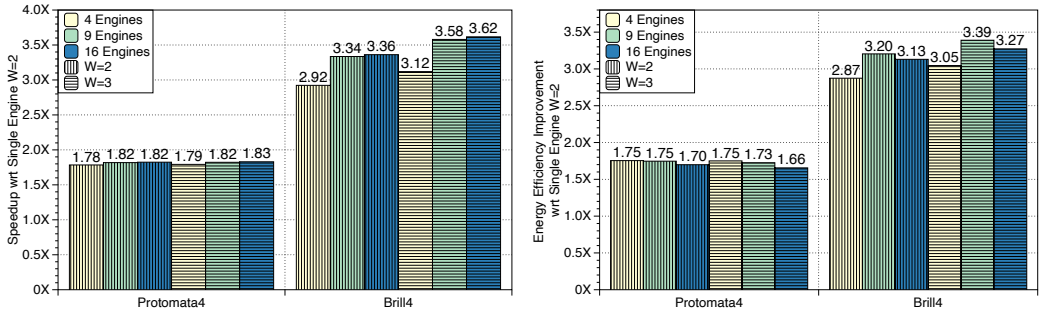
6.1.2 Character Window Scaling. Moving to the architectural enhancements, we start evaluating the impact of the increment in character processing rate, i.e., the character window (Section 5.2), against the base engine (Section 5.1). For this analysis, we employ the standard Protomata and Brill benchmarks, and randomly sampled 200 REs and 200 possible input strings to showcase the behavior on random REs. We measured the matching times of the FPGA prototypes through CICERO performance counters after loading the code and the string to match on CICERO memory. Besides, we extracted the board-level power consumption with the Voltcraft Energy-Logger 4000, which measures the board voltage, current, and power directly from the plug, and then computed the power consumption geometric means. Figure 14a shows the boxplot distributions of weighted average matching times of a CICERO single engine with windows of 2^W characters, where W is equal to 1, 2, 3. We chose the weighted average because it assesses the RE processing times better than raw runtimes as it also accounts for the processed characters. For each RE, we compute the weighted average as follows: $\frac{\sum_{i=0}^{N} MatchingTime_i \cdot l_i}{\sum_{i=0}^{N} l_i}$, where N is the number of input strings, and l_i the string length. In particular, according to the RE matching process outcome, we weigh the matching times with different input string lengths. If the RE does not match, we pick the whole string length; otherwise, as a match may occur at an arbitrary point of the input string, we approximate the number of processed characters with half of the string length.

The reader can notice from Figure 14a that moving from $W = 1$ to $W = 2$ reduces both the median (the black line within the colored box) and the height of the box, decreasing the average case and achieving steadier matching times. Comparing $W = 2$ and $W = 3$, the boxplot of the greater window (the blue one, $W = 3$) has a smaller length than the green one ($W = 2$). Therefore, it provides a steadier weighted matching time. However, this chart displays minimal differences among single CICERO engines with $W = 2$ and $W = 3$; therefore, we will still consider both configurations. Then, we computed the geomean of energy efficiency improvements when increasing the character window against the base engine with $W = 1$. To do so, we used the previously employed weighted matching time per RE, computed the energy efficiency as $1/(WeightedMatchingTime[ms] \times PowerConsumption[W])$, and finally the geomean. Figure 14b highlights that the CICERO single engine with $W = 2$ (i.e., the green one) is slightly more efficient than the one with $W = 3$ on Protomata, while it is slightly worse on Brill. Both Figure 14a and Figure 14b show that the increase in the window dimension gains practical improvements in median matching time, achieving steadier matching times and better energy efficiency. Figure 15 shows the resource utilization of the entire Ultra96v2 board, including both the CICERO engine and the additional logic that connects the engine to the ARM processor. We can notice that the CICERO engine is mainly LUT and BRAM demanding. Indeed, their usage grows according to W, as the engine needs further logic to manage the additional number of alternative paths. On the other hand, FF growth is more restrained since CICERO mainly employs FFs for the Manager state machine. Finally, the chart indicates that a CICERO engine has a low resource usage; indeed, even when W=3, the engine requires at most 5% of LUTs.

6.1.3 Engine Scaling. We analyzed the scaling effectiveness of CICERO multi-engine architecture. We employ the randomly sampled combined version of the two benchmarks (i.e., Protomata4 and Brill4) to increase the cores utilization. In this way, we aim at showcasing the impact of scaling to multiple engines with a ring topology. Figure 16a shows the geomean of the speedups achieved by 4, 9, and 16 cores with $W = 2$ (vertical lines) and $W = 3$ (horizontal lines) against the CICERO single engine with $W = 2$. For Both Protomata4 and Brill4, we obtain a speedup that scales with the core number. Conversely, Figure 16b displays the geomean of energy efficiency improvements at the core scaling on the same benchmarks and shows how the efficiency improvements do not reflect the speedups. Indeed, considering Protomata4, the most energy-efficient architecture has four engines with both W = 2 and W = 3. However, Brill4 indicates that the most energy-efficient architecture has nine engines with W = 3. These results prove how different architectures provide different trade-offs from both matching time and energy efficiency perspectives, depending not only on the kind of REs, but also on the input string. For these reasons, from now on, we will consider only the architecture with nine engines and $W = 3$ as the reference one, being the optimal trade-off among matching time and energy efficiency.

Figure 17 reports how the resource usage scales according to the number of engines and W. While the FF utilization remains relatively low (almost 8% in the worst case), the number of LUTs and BRAMs significantly increases due to the additional logic and memory required by both the engines and the load balancing infrastructure. This behavior is particularly evident when considering the sixteen-engine configuration. However, since such a configuration does not provide relevant performance benefits compared to a nine-engine one, there is no point in selecting it. On the other hand, even though $W = 3$ requires more resources than $W = 2$, the higher speedup and energy efficiency (especially on Brill4) compensates for the additional resources. This analysis further supports our choice of the nine-engine architecture with $W = 3$ as the reference one.

6.1.4 Topology Analysis. Before diving into the comparison with other literature approaches, we compare the ring topology against the torus one for our reference architecture of nine engines and



(a) Multi-engine geomean speedups. (b) Multi-engine geomean efficiency improvements.

Fig. 16. The effects on CICERO engine scaling (experiments performed on 200 inputs and 200 REs sampled from Protomata4 and Brill4 benchmarks).

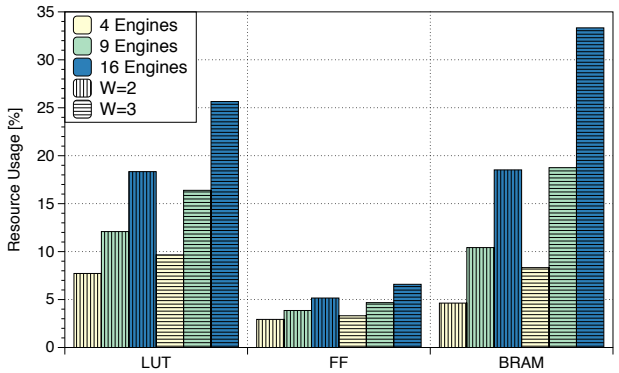
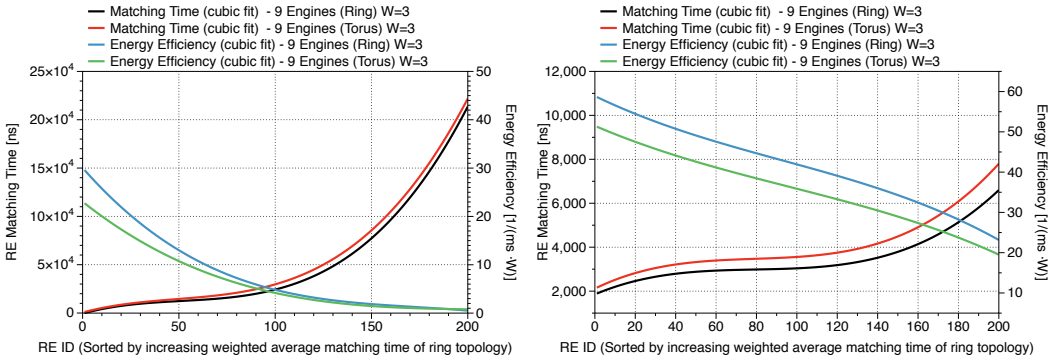


Fig. 17. Resource usage percentage when scaling the number of engines (from 4 to 16) considering W=2 (vertical lines) and W=3 (horizontal lines).

$W = 3$. We exploited the same benchmarks as before, i.e., Protomata4 and Brill4, and compare the matching times and energy efficiencies of both topologies. Figure 18a and Figure 18b show the cubic fits of these measures. In this way, the reader can see that, on Protomata4 benchmark, the torus curve (the red line) always remains above the ring one (the black line) as the RE matching time increases. Considering the efficiency curves, even though the ring (the blue line) generally shows a better energy efficiency, the torus (the green line) performs slightly better over the most time-demanding subset of REs, though it is a very restricted subset. Moving to the Brill4 results in Figure 18b, the cubic fits of matching times and energy efficiencies demonstrate that the ring gains better matching time (i.e., the black line stays below the red one) and achieves a higher efficiency (i.e., the blue line stays above the green one) for all the considered REs. In conclusion, although the torus represents a better solution in terms of scalability and a good candidate for a more suitable layout design, these results prove that the simplicity of ring topology is enough to keep up with both the matching time and energy efficiency of Protomata4 and Brill4 benchmarks. Besides, as shown in Figure 19, the ring topology is also less resource-demanding than the torus one.



(a) Matching times and energy efficiencies on Protomata4 benchmark.

(b) Matching times and energy efficiencies on Brill4 benchmark.

Fig. 18. Weighted average matching time and energy efficiency of ring and torus topologies (experiments performed on 200 inputs and 200 REs sampled from Protomata4 and Brill4 benchmarks).

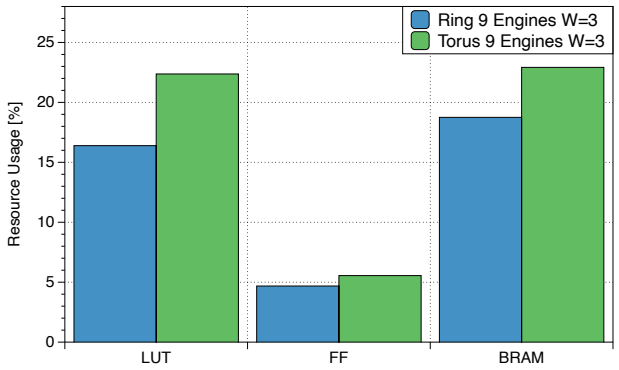
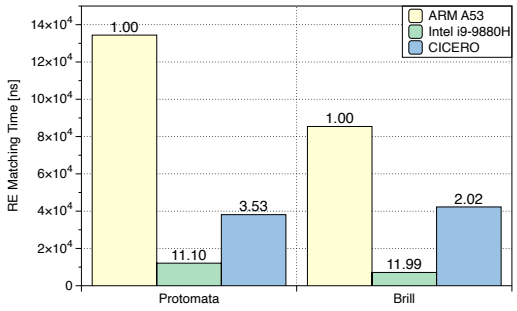


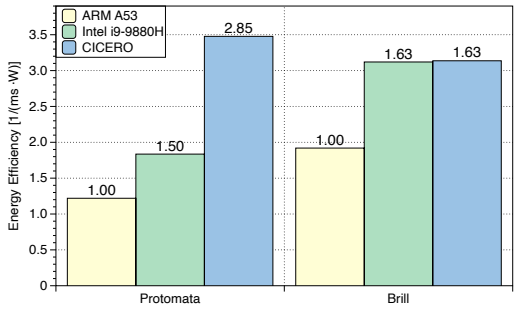
Fig. 19. Resource usage when considering a 9-Engine architecture with W=3 but with different topologies.

6.2 Comparison Against Google RE2

As mentioned before, our analysis identified the ring-based nine-engine architecture with $W = 3$ as the most efficient one. This implementation requires 11,563 (16.39%) LUTs, 6,600 (4.68%) FFs, and 81 (18.75%) BRAMs on the Ultra96v2 FPGA. We compared our implementation with Google RE2 executed on two candidate processors: an embedded solution, the ARM Cortex A53 (mounted on the Ultra96v2), and a mainstream one, the Intel i9-9880H. The RE2 library was built from sources [15] with `-O3` optimizations. We set the comparison on partial match operation in cold-start conditions and measured matching time and energy efficiency for the matching process only. As previously stated, we relied on the CICERO performance counters to measure the matching time after loading the code and the string to match on CICERO memory. We used the C++ chrono library to measure the execution time of a RE2 code snippet that interprets the RE by creating an `RE2::RE2` object and calling the `RE2::PartialMatch` function. We repeated the entire procedure 30 times for both Intel and ARM CPUs to mitigate cache effects and acquire statistically relevant results. As before, we measured and weighted the average execution times on the string length analyzed. Finally, we selected the Thermal Design Power (TDP) of the Intel CPUs as reference power consumption. For

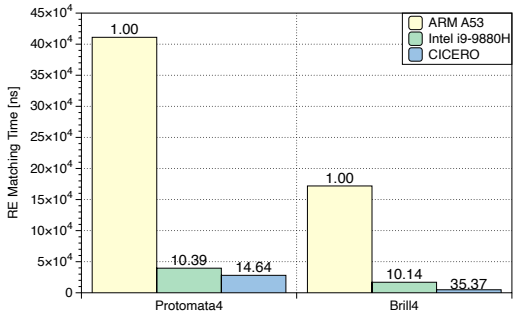


(a) Matching times (the lower the better) and speedups wrt ARM A53.

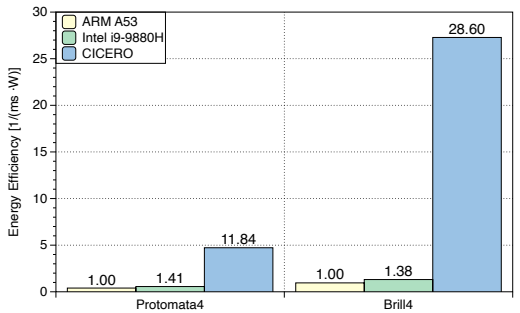


(b) Energy efficiencies (the higher the better) and improvements wrt ARM A53.

Fig. 20. Geomean of matching times and energy efficiency comparison between ARM A53, Intel i9, and CICERO on the complete Protomata and Brill benchmarks.



(a) Matching times (the lower the better) and speedups wrt ARM A53.



(b) Energy efficiencies (the higher the better) and improvements wrt ARM A53.

Fig. 21. Geomean of matching times and energy efficiency comparison between ARM A53, Intel i9, and CICERO on 200 inputs and 200 REs sampled from Protomata4 and Brill4 benchmarks.

the ARM CPUs and the CICERO FPGA prototype, we employed the Voltcraft Energy-Logger 4000 to extract the board-level power consumption.

Figure 20 shows the geomean values of the matching times (left-hand side) and energy efficiency results (right-hand side) achieved by the A53, i9, and CICERO on all the possible RE-input couples from Protomata and Brill benchmarks. The results in Figure 20a show that CICERO achieves lower matching times than the embedded processor (i.e., the A53) with speedups of 3.526× and 2.021× on the Protomata and Brill, respectively. However, the i9 shows matching times even better than ours in both benchmarks. Instead, considering the energy efficiencies achieved, Figure 20b highlights comparable results of CICERO and the mainstream processor (i.e., the i9) on the Brill benchmark, i.e., values are around 3.136 and 3.119 $[\frac{1}{ms \cdot W}]$, respectively. However, Protomata benchmark shows the CICERO advantage of tailoring the architecture for delivering a higher energy-efficient computation. Indeed, CICERO delivers 1.89× and 2.851× energy efficiency improvements than the i9 and the A53, respectively.

Overall, these results exhibit remarkable matching times and higher efficiency with the standard benchmarks as they are. However, our approach, built on Thompson's algorithm, which can scale

Table 2. Related Work Summary.

Work	Target Scenario	Device	Architecture Type	Execution Mode	Compilation Framework	Compilation Time Required
CICERO-FPGA	Embedded	XCZU3EG (16nm)	Software-Programmable	NFA	Yes	Software like
CICERO-ASIC	Embedded	ASIC (32nm)	Software-Programmable	NFA	Yes	Software like
FPGA [40]	Datacenter	XC7VX690T (28nm)	Stream-Dataflow	DFA	No	Bistream like
FPGA [20]	Datacenter	XC7VX1140T (28nm)	Stream-Dataflow	DFA	No	Bistream like
FPGA/ASIC [14]	Embedded	Arria V SoC (28nm)/ ASIC (45nm)	Stream-Dataflow	DFA	No	Bistream like/N.A.
FPGA [38]	Datacenter	ADM-KU3 (20nm)	Stream-Dataflow	NFA	Yes	Bistream like
FPGA [6]	Datacenter	XCVU9P (16nm)	Stream-Dataflow	NFA	Yes	Bistream like
FPGA[26, 29]	Datacenter	XCVU9P (16nm)	Stream-Dataflow	NFA	Yes	Bistream like
TCAM [23]	Datacenter	N.A.	Software-Programmable	DFA	N.A.	Software like
PowerEN [7, 34]	Datacenter	ASIC (45nm)	Software-Programmable	DFA	Yes	Software like
FPGA [8]	Datacenter	XC7VX485T (28nm)	Software-Programmable	DFA	Yes	Software like
AP [6, 13, 29, 35]	Datacenter	DRAM (22nm)	Software-Programmable	NFA	Yes	Software like

linearly in the string length without paying the cost of alternative paths backtracking. For this reason, we collect the results of a combined version of the benchmarks as previously described. This combination increases the alternative paths simultaneously active and better mimics a real scenario where the final user aims to search all the REs in a single input pass. Indeed, considering the analysis of gigabytes of data, an optimized search wants to reduce as much as possible the number of times to scan the input data. Figure 21 presents the results of these combined experiments on the considered architectures. While the i9 matching times hold the same magnitude order as the standard benchmark and the A53 shows deterioration, CICERO reveals dramatic improvements as in Figure 21a. Indeed, the speedups achieved by the i9 and CICERO over the A53 are 10.386× and 14.642× on Protomata and 10.144× and 35.370× on Brill. While comparing the i9 with CICERO, our architecture delivers speedups of 6.173× and 3.487× against the mainstream architecture on Protomata and Brill, respectively. Moving to Figure 21b and the associated energy efficiency results, CICERO FPGA-based implementation presents outstanding results. Especially, CICERO achieves 8.409× and 20.798× energy efficiency improvements than the most energy-efficient CPU (i.e., the i9) and 11.839× and 28.600× than the A53 over Protomata and Brill, respectively. The domain specialization of the architecture combined with Thompson's approach leads to remarkable speedups and energy-efficient computations against both an embedded and a mainstream processor.

6.3 Analysis of ASIC Implementation

We synthesized CICERO with Synopsys Design Compiler Q-2019.12-SP4 targeting the Synopsys SAED 32nm technology at typical conditions. We estimated the resulting total silicon area, power consumption, and maximum target frequency. SRAM synthesis limits the maximum target frequency to 1 GHz. When targeting this frequency, we obtained an implementation that occupies $0.794mm^2$ – which is compatible with modern accelerators at 32nm [22] – and consumes $9\mu W$. We used the number of clock cycles from the best FPGA prototype to estimate the efficiency of the custom chip implementation. On average, the resulting design is at least 3,900,000× more energy-efficient than the Intel CPU on the complete benchmarks.

7 RELATED WORK

Modern automata processing benchmarks, such as ANMLZoo [35] and AutomataZoo [36], allowed designers to demonstrate the efficiency of FPGA implementations over general-purpose and spatial architectures [24]. This Section discusses existing accelerators for RE matching available in the

literature, excluding inexact matching solutions, such as [5, 12, 30], that are out of the scope of this work. Since the approaches are extremely different, a quantitative comparison would not be fair, as discussed later in this Section. Thus, we provide a qualitative analysis, as reported in Table 2.

Stream-Dataflow Architectures usually exploit spatial architectures, particularly reconfigurable ones. The works that mainly rely on deterministic finite automata (DFA) focus their efforts on DFA compression. However, they require hardware re-synthesis for each new set of automata [14, 20, 40], which may demand from few hours to days. On the other hand, REAPR [38] is a tool that translates non-deterministic finite automata (NFA) into RTL implementations. The authors expand their work to support AWS F1 instances [25] and to allow a fast reconfiguration of different REs that exploit the same NFA structure [6]. On top of these approaches, other authors propose a compiler framework called FlexAmata that aims at optimizing the automata representation also considering different alphabet symbols bitwidth [29], further extended to exploit either LUT- or BRAM-based designs [26]. CICERO is, instead, a specialized but flexible architecture that can support several different REs thanks to the custom ISA and the compiler-based framework. Therefore, our approach enables generating a new binary code with software compilation time instead of waiting hours for generating a bitstream. Thus, comparing these two diverse approaches would not result in a fair apple-to-apple comparison.

In-Memory or Software-Programmable Architectures (SP) are platforms more similar to CICERO. Indeed, they can update the set of REs without changing the underlining architecture, nor the bitstream. Meiners *et al.* [23] exploit Ternary Content Addressable Memories (TCAM) for high-speed REs matching through efficient DFA representation, but they do not provide insight on the compilation framework. Van Lunteren and Guanell [7] present an extension of the IBM Power Edge of Network (PowerEn). They encode the DFA as a set of rules into the memory of a software-programmable DFA engine by exploiting the B-FSM concept for reprogrammable DFA [34]. Comodi *et al.* [8] present an architecture that exploits a VLIW-like approach for a DFA with backtracking execution mode. All these methods are based on the DFA representation, which grows exponentially with the complexity of the RE. Instead, we focus on an NFA implementation to allow parallelization of the alternatives. The algorithmic approaches are fundamentally different, though semantically equivalent, and the NFA implementation proved to lead execution times linear in the string length.

The Automata Processor (AP) was an outstanding spatial reconfigurable architecture that embedded a target automaton into the reconfigurable fabric [13, 37]. While it was a promising solution with high performance [29, 42] and no FPGA bitstream overhead [24, 38], only simulation results of the AP were reported while we show a prototype executed on FPGA [6][4].

8 CONCLUSIONS AND FUTURE WORK

We presented CICERO, a software-programmable, domain-specific architecture for Regular Expression (RE) matching. CICERO exploits Thompson's algorithm to create a non-deterministic RE representation that can execute on multiple engines in parallel without backtracking. We also provide an end-to-end framework for translating REs into optimized code. We validate CICERO architectural optimizations on an embedded FPGA on benchmarks from AutomataZoo [36], showing increasing benefits in the proposed solution, e.g., from the code size to the matching times and energy efficiencies. CICERO multi-engine and multi-character architecture shows up to 28.6× and 20.80× energy efficiency improvements against the highly optimized Google RE2 library onto an embedded processor (the ARM A53) and a mainstream processor (the Intel i9), respectively. We provide an estimation of a software-programmable ASIC prototype on the same benchmarks projecting remarkable advantages.

[4]At the time of writing the AP SDK is not available anymore

In our research plan, we aim at analyzing new interconnection topologies along with different load balancing distributions. Moreover, we envision a vectorized CICERO engine that will potentially provide further advantages. Finally, the benefits of the RE combination show that there is room for improving the preprocessing step. Indeed, by deeply analyzing available benchmarks, a run-time manager could determine the optimal combination degree for the given REs and reconfigure the FPGA with the architecture that most fits the workload.

ACKNOWLEDGEMENT

We would like to thank the Xilinx University Program for providing the Ultra96 used in this paper.

REFERENCES

[1] M. Abbas and V. Betz. 2018. Latency Insensitive Design Styles for FPGAs. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*. 360–3607.

[2] K. Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. The landscape of parallel computing research: A view from Berkeley. (2006).

[3] Michela Becchi and Patrick Crowley. 2007. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 1–12.

[4] Michela Becchi and Patrick Crowley. 2008. Efficient regular expression evaluation: Theory to practice. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 50–59.

[5] Andreas Becher, Stefan Wildermann, and Jürgen Teich. 2018. Optimistic regular expression matching on FPGAs for near-data processing. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*. 1–3.

[6] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. 2019. Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems* 12, 2 (2019), 1–25.

[7] Jeffrey Brown, Sandra Woodward, Brian Bass, and Charlie Johnson. 2011. IBM power edge of network processor: A wire-speed system on a chip. *IEEE Micro* 31, 2 (2011), 76–85.

[8] Alessandro Comodi, Davide Conficconi, Alberto Scolari, and Marco D. Santambrogio. 2018. TiReX: Tiled regular expression matching architecture. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 131–137.

[9] 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

[10] Russ Cox. 2009. Regular expression matching: the virtual machine approach. http://swtch.com/rsc/regexp/regexp2.html

[11] Russ Cox. 2012. Regular Expression Matching with a Trigram Index or How Google Code Search Worked. https://swtch.com/~rsc/regexp/regexp4.html

[12] Lorenzo Di Tucci, Davide Conficconi, Alessandro Comodi, Steven Hofmeyr, David Donofrio, and Marco D Santambrogio. 2018. A Parallel, Energy Efficient Hardware Architecture for the merAligner on FPGA Using Chisel HCL. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 214–217.

[13] Paul Dlugosch et al. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098.

[14] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

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

[16] John L Hennessy and David A Patterson. 2018. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

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

[18] M. Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*. 10–14.

[19] Zsolt István, David Sidler, and Gustavo Alonso. 2016. Runtime parameterizable regular expression operators for databases. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 204–211.

[20] Lei Jiang, Qiong Dai, Qiu Tang, Jianlong Tan, and Binxing Fang. 2014. A fast regular expression matching engine for NIDS applying prediction scheme. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*.

[21] Donald E Knuth. 1965. On the translation of languages from left to right. *Information and control* 8, 6 (1965), 607–639.
[22] P. Mantovani, E. G. Cota, K. Tien, C. Pilato, G. Di Guglielmo, K. Shepard, and L. P. Carlon. 2016. An FPGA-Based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*.
[23] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. 2010. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of the USENIX conference on Security*.
[24] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP?. In *Proceedings of the International Conference on Supercomputing (ICS)*. 1–11.
[25] David Pellerin. 2017. Fpga accelerated computing using aws f1 instances. *AWS Public Sector Summit* (2017).
[26] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 2020. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 138–147.
[27] Alex Roichman and Adar Weidman. 2012. Regular Expression Denial of Service.
[28] Indranil Roy. 2015. *Algorithmic techniques for the micron automata processor.* Ph.D. Dissertation. Georgia Institute of Technology.
[29] 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 ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 219–234.
[30] Davide Sampietro, Chiara Crippa, Lorenzo Di Tucci, Emanuele Del Sozzo, and Marco D Santambrogio. 2018. Fpga-based pairhmm forward algorithm for dna variant calling. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–8.
[31] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 403–415.
[32] Shreyas G Singapura, Yi-Hua E. Yang, Anand Panangadan, Tamas Nemeth, and Viktor K. Prasanna. 2015. *FPGA Based Accelerator for Pattern Matching in YARA Framework.* Technical Report. CE, Los Angeles, CA.
[33] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422.
[34] Jan van Lunteren and Alexis Guanella. 2012. Hardware-accelerated regular expression matching at multiple tens of Gb/s. In *Proceedings of IEEE INFOCOM*. IEEE, 1737–1745.
[35] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 1–12.
[36] 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.
[37] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An overview of micron's automata processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 1–3.
[38] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.
[39] Xilinx. 2016. PYNQ: Python for Productivity for ZYNQ. http://www.pynq.io/.
[40] J. Yang, L. Jiang, Q. Tang, Q. Dai, and J. Tan. 2016. PiDFA: A practical multi-stride regular expression matching engine based on FPGA. In *Proceedings of the IEEE International Conference on Communications (ICC)*. 1–7.
[41] 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.
[42] Keira Zhou, Jack Wadden, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *Proceedings of the IEEE International Conference on Big Data (BigData)*. 355–360.

Received April 2021; revised June 2021; accepted July 2021