Robotomata: A Framework for Approximate Pattern Matching of Big Data on an Automata Processor

Xiaodong Yu, Kaixi Hou, Hao Wang, Wu-chun Feng
Department of Computer Science, Virginia Tech
Email: {xdyu, kaixihou, hwang121, wfeng}@vt.edu

***Abstract*—Approximate pattern matching (APM) has been widely used in big data applications, e.g., genome data analysis, speech recognition, fraud detection, computer vision, etc. Although an automata-based approach is an efficient way to realize APM, the inherent sequentiality of automata deters its implementation on general-purpose parallel platforms, e.g., multicore CPUs and many-core GPUs. Recently, however, Micron has proposed its Automata Processor (AP), a processing-in-memory (PIM) architecture dedicated for non-deterministic automata (NFA) simulation. It has nominally achieved thousands-fold speedup over a multicore CPU for many big data applications. Alas, the AP ecosystem suffers from two major problems. First, the current APIs of AP require manual manipulations of all computational elements. Second, multiple rounds of time-consuming compilation are needed for large datasets. Both problems hinder programmer productivity and end-to-end performance.**

Therefore, we propose a paradigm-based approach to hierarchically generate automata on AP and use this approach to create Robotomata, a framework for APM on AP. By taking in the following inputs — the types of APM paradigms, desired pattern length, and allowed number of errors as input — our framework can generate the optimized APM-automata codes on AP, so as to improve programmer productivity. The generated codes can also maximize the reuse of pre-compiled macros and significantly reduce the time for reconfiguration. We evaluate Robotomata by comparing it to two state-of-the-art APM implementations on AP with real-world datasets. Our experimental results show that our generated codes can achieve up to 30.5x and 12.8x speedup with respect to configuration while maintaining the computational performance. Compared to the counterparts on CPU, our codes achieve up to 393x overall speedup, even when including the reconfiguration costs. We highlight the importance of counting the configuration time towards the overall performance on AP, which would provide better insight in identifying essential hardware features, specifically for large-scale problem sizes.

***Keywords*-Approximate Pattern Matching; Nondeterministic Finite Automata; Automata Processor**

I. Introduction

Approximate pattern matching (APM), also known as fuzzy search, is essential to many applications, e.g., genome data analysis, speech recognition, fraud detection, etc. It gets more attentions in big data era since it is the key to retrieve information from a sea of dirty data, where data is not only dynamic and high volume but with human errors. An typical example is the Google search. Our inputs may frequently have misspelled or abbreviated words, and the search engine has to locate the strings approximately. One solution is to index data to support fuzzy search like Google does [1]. But it demands a huge cost to prepare data in a database that is barely affordable to most businesses. Automata-based mechanism is another solution for the online APM search. It uses Finite Automata (FA) as the core and has much less time and space complexity of data preprocessing than the index-based method [2]. On the other hand, FA is inherently sequential [3], hence extremely hard to be parallelized on HPC platforms to achieve the fast speed.

Recently, a new hardware Automata Processor (AP) [4] is introduced by Micron for the non-deterministic FA (NFA) simulations. AP can perform parallel automata processing within memory arrays on SDRAM dies by leveraging memory cells to store trigger symbols and simulate NFA state transitions. However, it requires programmers to manipulate computational elements, called State Transition Elements (STEs), and inter-connections between them with a low-level language, i.e., Automata Network Markup Language (ANML). Although AP SDK provides some high-level APIs for some kinds of applications, e.g., regular expression [5] and string matching [6], the lack of customizable capability would force users to resort to ANML for their own applications. Programming on AP is still a cumbersome task, requiring considerable developer expertise on both automata theory and AP architecture.

A more severe problem is the scalability. For reconfigurable devices like AP, a series of costly processes are needed to generate the load-ready binary images. These processes include synthesis, map, place-&-route, post-route physical optimizations, etc., leading to the non-negligible configuration time. For a large-scale problem, the situation becomes worse because the multi-round reconfiguration might be involved. Most previous research on AP [7], [8], [9], [10], [11], [12], [13], [14] ignores and excludes the configuration cost and focuses on the computation. Although these studies reported hundreds or even thousands of fold speedups over multicore CPUs, the end-to-end time comparison, including configuration and computation, is not well understood.

We believe a fair comparison has to involve the configuration time, especially when the problem size is extremely large and exceeds the capacity of a single AP board. In such a case, the overhead of configuration could be very high for three reasons: (1) A large-scale problem may need multiple rounds of binary image load and flush. (2) Once a new binary image is generated, it will use a full compilation process, which time is as high as several hours. (3) During these processes, the AP device is forced to stall in an idle status and wait for the new images. Therefore, we highlight the importance of counting

the reconfiguration time towards the overall performance on AP, which would provide a better angle for researchers and developers to identify essential hardware architecture features. For example, the claimed speedups of AP-based DNA string search [15] and motif search [8] can get up to 3978x and 201x speedups over their CPU counterparts, respectively. In contrast, if their pattern sets scale out and the reconfiguration overhead is included, the speedups are reported down to 3.8x and 24.7x [16].

In this paper, we propose a hierarchical approach to build application automata on AP to improve the programming productivity and performance. Specifically, our approach includes four steps: First, we identify the basic paradigms of an application and map them to a building block. Second, we connect building blocks with an inter-block transition connecting algorithm. Third, we extend a group of building blocks to a cascadable AP macro by adding and merging input/output ports. Fourth, we generate the automata codes on AP with a macro-based construction algorithm. We use this approach to develop a framework, Robotomata, for Approximate Pattern Matching (APM) applications. Robotomata takes the types of paradigms, pattern length, and allowed errors as input, and generates the optimized ANML codes as output, so as to improve the programming productivity. Our generated codes can also maximize the reuse of pre-compiled macros, and thus significantly reduce the reconfiguration time. We evaluate Robotomata by comparing to two state-of-the-art approaches: ANML_APIs [17] and FU [18] with the real-world datasets. The results show that our generated codes can achieve up to 39.6x and 17.5x speedup in the configuration and deliver the performance closed to the manually-optimized implementation. Compared to the counterparts on the multicore CPU, our codes can achieve up to 461x speedup, even including the multi-round reconfiguration costs. The major contributions of this paper include:

- We reveal the problems of programmability and performance on Micron AP, especially the reconfiguration cost for large-scale problem size.
- We use a hierarchical approach and a framework to generate optimized low-level codes for Approximate Pattern Matching applications on AP.
- We evaluate the framework by comparing to state-of-the-art research with the real-world datasets. The experimental results illustrate the improved programming productivity and significantly reduced configuration time in our method.

II. Background and Motivations

A. Automata Processors

AP Architecture Overview: Micron's AP is a DRAM-based reconfigurable device dedicated for NFA traversal simulations. Its computational resources consist of three major programmable components: STEs, Counters, and Boolean gates. The STEs provide the capability of massive parallelism. Each STE includes a memory cell and a next-state decoder to simulate a NFA state with trigger symbols, i.e., 256-bit masks for ASCII characters. The connections between STEs simulate NFA state transitions, implemented by a programmable routing matrix. In a cycle, all active STEs simultaneously compare their trigger symbols to the input character, and those hit STEs activate all their destination STEs via activation signals emitted by next-state decoders. The counters and boolean gates are special-purpose elements to extend the capacity of AP chips beyond classical NFAs.

The current generation of AP board (AP-D480) contains 32 chips organized into 4 ranks. Two half-cores reside in an AP chip, which includes a total 49,152 STEs, 768 counters, and 2,304 boolean gates. These programmable elements are organized as following: two STEs and one special-purpose element (a counter or boolean gate) form a group; 8 groups form a row; 16 rows form a block; and eventually, 192 blocks are evenly distributed into the two half-cores.

AP Programming and Pre-compiling: ANML is a XML-based language for programming on AP. Developers can define their AP automata in ANML by describing the layout of programmable elements, e.g., STEs and transitions. An AP toolchain can parse such an ANML program and compile it to binary images. At runtime, after the AP board loads a binary image, it starts to search given patterns against input streams of 8-bit characters by nominally processing one character per clock cycle at the 133 MHz frequency. AP reports the events to the host, if matches are detected.

Analogous to other reconfigurable devices, AP compilation is a time-consuming process due to the complex place-&-route calculations. To mitigate the overhead of compilation, the AP SDK supports the pre-compile of automata as *macros* and reuse macros to build larger automata in the future. That way, any subsequent automata having existing macros can be built directly on them with a lightweight relabeling, i.e., changing trigger symbols as demand. This can avoid the recompilation from the ground up.

B. Motivations of This Work

Embarrassingly Sequential: Finite State Automata is known as one of the "embarrassingly sequential" applications [3]. Two major reasons make them difficult to be parallelized. First, there are tight data dependences between consecutive steps. Second, the access patterns are input-dependent and unpredictable [19]. There are a lot of studies refactoring and optimizing the automata on CPUs and GPUs [19], [20], [21], [22]; however, as Tab. I shows, their achievements [23], [24], [25] are still far below the full or even typical throughput and utilization of the hardware [26], [27], [28], [29], [30], [31], [32], [33], [34], [35]. This is a major reason why we consider to use the new AP hardware.

TABLE I: FA traversal throughput (Gbps)

Devices	Model	Full capacity	FA throughput
CPU	Intel X5650	~ 260	~ 2.4 [19]
GPU	NVIDIA GTX480	$\sim 1.4k$	~ 0.3 [36]
FPGA	Virtex-6 family	~ 600	~ 20 [24]

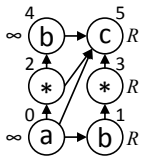


Fig. 1: A simple case of AP automaton.

Reconfiguration Issues on AP: For a large-scale problem size which exceeds the capacity of a single AP board, multi-round configurations are required. For example, the *alua* dataset used in Bioinformatics sequence search requires 20 rounds of reconfiguration. As shown in Sec. V, a single-round takes as high as 130 seconds, and around 2600 seconds should be spent on the reconfiguring in total. Obviously, excluding such cost leads to unfair performance comparison between AP and counterpart platforms. Although AP SDK provides the pre-compiling technique to alleviate the overhead of reconfiguration, it is restricted to handle an exactly same automaton structure with a pre-compiled macro. If the structure changes, a full compilation process is still required. For example, the pre-compiled macros unfortunately would fall short from reducing the cost of aforementioned Bioinformatics case, because the change of the pattern lengths leads to various automata structures. Hence, a new approach is of great necessity to help users fully explore the AP capacity, by maximizing the reuse of existing macros even if automata structures differ.

Programmability Issues on AP: Fig. 1 shows an example of Approximate Pattern Matching, which searches the pattern "abc" and allows one error at most. The STEs take characters "a", "b", "c", and "*" as the input symbols. Such a simple automaton surprisingly needs 44 lines of ANML code to be constructed: developers need to create an element for each STE and define its attributes, and then transform the errors to edges between STEs. Both the expertises of Approximate Pattern Matching and ANML programming model of AP are required. Furthermore, if the pre-compiling technique is used to optimize the configuration, the case will become more complicated and error-prone.

III. Paradigms and Building Blocks in APM

In this section, we start from the manual implementation and optimization of mapping APM on AP to further illustrate the complexity of using ANML. Then, we identify the paradigms in APM applications, organize paradigms to a building block, and then discuss the inter-block transition connecting mechanism to construct automata from blocks.

A. Approximate Pattern Matching on AP

APM is to find the strings matching a pattern with limited number of errors (insertion, deletion, and substitution). There are four most common distance types allowing various subsets of the errors, including Levenshtein, Hamming, Episode, and Longest Common Subsequence [2]. Because the Levenshtein Distance (aka. edit distance) allows all three kinds of APM errors, we use it to discuss our design. Fig. 2 shows an automaton with the edit distance that detects the pattern "object" and allows up to two errors. The ε-transitions allow the traverse reaching destination states immediately without consuming any input character in Levenshtein automaton. The $*$-transitions allow the traverse to be triggered by any input character. In this case, an asterisk-transition represents an insertion or substitution when the traverse goes vertically or diagonally. We apply two optimizations [37] on the Levenshtein automata to reduce the usage of STEs by skipping the states above the first full diagonal and below the last full diagonal (i.e. the two dotted triangles A and B in Fig. 2). They are important for mapping the Levenshtein automata on AP, considering the limited numbers of STEs of AP.

We then map the optimized automaton onto AP using ANML. The automaton in Fig. 2 will be transformed to the recognizable format on AP, as shown in Fig. 3. Two major problems have to be resolved in the mapping. First, the ANML programming model requires moving NFA trigger-character associations from transitions to states, because it can't support multiple outgoing transitions with different character sets from a single state, e.g. the transitions *s2* →*s3*, *s2*→*s9*, and *s2*→*s10* in Fig. 2. The solution is to split a state to multiple STEs. For example, the state *s2* in Fig. 2 is split to STE1 with the trigger character *o*, and the auxiliary STE2 with an asterisk. The second problem is that AP hardware can't reach the destination states in the current clock-cycle, leading to the lack of support to ε-transitions. The alternative is to add a transition from a source STE to its upper-diagonal adjacent STE. For example, the ε-transition from *s2* to *s10* in Fig. 2 is transformed to the transition from STE1 to STE8 in Fig. 3. With these two transformations, the Levenshtein automaton in Fig. 3 can be described in ANML and mapped on AP.

B. Paradigms and Blocks Construction

The transformation of Levenshtein automata illustrates programming with ANML requires advanced knowledge of both automata and AP architecture. Any parameter change, e.g., the pattern length, error type, max error number, etc., may

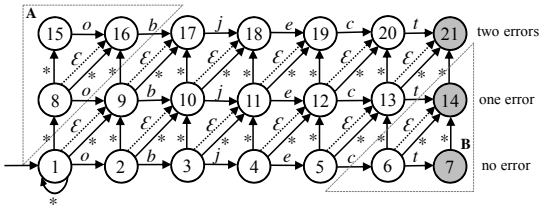


Fig. 2: Levenshtein automaton for pattern "object", allowing up to two errors. A gray colored state indicates a match with various numbers of errors.

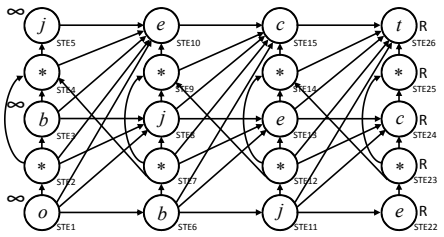


Fig. 3: Optimized STE and transition layout for Levenshtein automata for the pattern "object" allowing up to two errors.

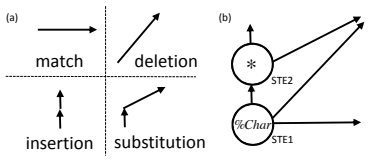


Fig. 4: (a) Four paradigms in APM applications. (b) A building block for Levenshtein automata having all four paradigms.

TABLE II: Paradigm sets for common distances

Distance	Paradigm
Levenshtein	M, S, I, D
Hamming	M, S
Episode	M, I
Longest Common Subsequence	M, I, D

lead to the code adjustment with tedious programming efforts. Hence, we introduce a hierarchical approach: it first exploits the paradigms of applications to form building blocks, then constructs block matrix and adds inter-block connections.

Paradigms and Building Blocks: APM has three types of errors: *insertion*, *deletion*, and *substitution* (denoted as *I*, *D*, and *S*). These three kinds of errors with the *match*, denoted as *M*, can be treated as the paradigms of any APM problem. They can be represented in an AP recognizable format as shown in Fig. 4a. An APM automaton takes one or more paradigms depending on the distance type as shown in Tab. II. For the Levenshtein automata that can take all three error types, a building block including two STEs and four types of transitions is shown in Fig. 4b. The STE with the asterisk is the design alternative to support multiple outgoing transitions with different character sets.

Block Matrix: With the building block, once the length of desired pattern n and the maximum number of errors m are given, building an AP automaton can be simplified to duplicate building blocks by organizing them into a $(m+1)*(n-m)$ **block matrix**. The m rows correspond to the number of allowed errors (row 0 is for the exact match), and the $(n-m)$ columns correspond to the pattern length (cutting down m is from the optimizations discussed in Sec. III-A). For example, a Levenshtein automaton allowing up to two errors for the pattern "object" having 6 characters can be built to have $(2+1)*(6-2)=12$ blocks by organizing them to a $3*4$ block matrix as shown in Fig. 3.

Inter-Block connections: The cross-row transitions between building blocks are demanded to handle consecutive errors. Three paradigms of errors: *I*, *D*, and *S* can result in 9 two-consecutive-error pairs in the Levenshtein automata. Since the order of errors doesn't affect the matching results and some pairs are functionally equivalent (e.g. DI and ID are equal to a single S), we discuss five distinguished pairs: SS, II, DD, SI, and SD.

Fig. 5 is a part of Levenshtein automaton in Fig. 3 with added inter-block transitions. We denote two STEs in a block with the exact character and the asterisk as $B_{i,j}.c$ and $B_{i,j}.a$, respectively. The inter-block transition have two attributes: *direction* (*upward*, *forward*, and *backward*) and *STE-pair type* (*character*→*asterisk*(c→a), *asterisk*→*character*(a→c), *character*→*character*(c→c), and

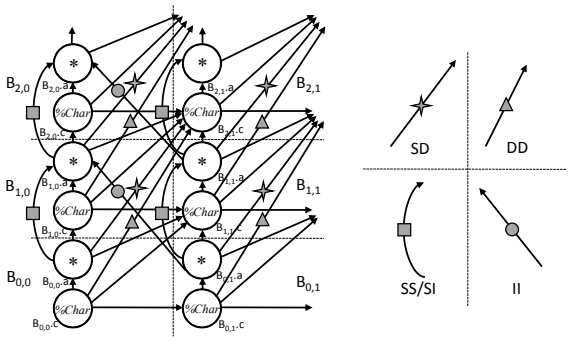


Fig. 5: Inter-block transition connection mechanism for two-consecutive errors: The right part shows five cases we consider. The left part shows a part of Levenshtein automata having six blocks and corresponding inter-block connections.

TABLE III: Inter-block transitions connecting rules

Paradigm	Transition Attributes	STE Connection
M	None	
SS/SI	*upward* a→a	$B_{i,j}.a \rightarrow B_{i+1,j}.a$
II	*backward* a→a	$B_{i,j}.a \rightarrow B_{i+1,j-1}.a$
DD	*forward* c→c	for k=2 to m: $B_{i,j}.c \rightarrow B_{i+k,j+1}.c$
SD	*forward* a→c	for k=2 to m: $B_{i,j}.a \rightarrow B_{i+k,j+1}.c$

asterisk→*asterisk*(a→a)). Some combinations of these two attributes are invalid or functionally duplicated; as shown in Fig. 5, only four combinations are needed: *forward* a→c), *forward* c→c, *upward* a→a, and *backward* a→a; they can cover all five cases of two-consecutive errors. Note that the inter-block transitions with asterisks as the destination are always for two different cases, because the traverse cannot stop at an asterisk. Tab. III summarizes the rules of adding inter-block transitions for two-consecutive errors. Any case having more than two consecutive errors can be handled as a combination of two-consecutive errors without any additional transitions.

IV. Framework Design

In this section, we introduce our formalized *building blocks* and *macros* to build automata. Then, we present the complete framework–*Robotomata* that can automatically generate automata in an optimized way (i.e., from reusable macros) to reduce the recompilation overhead.

A. From Building Blocks to Automata

Alg. 1 presents how to build automata on AP from building blocks. This algorithm accepts the types of paradigms, desired pattern length, and maximum number of allowed errors. Then, it can automatically fabricate complex AP automata. Note that this process is independent with specific APM applications so that it can work for any type of distances with paradigm combinations, as shown in Tab. II.

In Alg. 1, we use classes of `BuildingBlock` (ln. 4) and `Automation` (ln. 3) to represent building blocks and AP automata, respectively. First, we create the building blocks via ln. 5-6, where the member function `add()` in

`BuildingBlock` places and connects STEs following the rules depicted in Fig. 4b. Second, we build up the automaton with the block matrix determined by maximum error (m) and target pattern length (n) (ln. 7-8). After duplicating the blocks to fill the dimensions in ln. 9, we weave them together through the `ADD_TRANSITIONS()` to add inter-block transitions defined in Sec. III-B. Third, the starting and reporting STEs are set in ln. 13 and all STEs are labels in ln. 22-24. To this point, an AP automaton is constructed and ready to be compiled to a binary image. Note that the macro-based optimization in ln. 14-21 is an optional process, which will be discussed in Sec. IV-B and Sec. IV-C.

Algorithm 1: Paradigm-based AP Automata Construction

```
   /* Alg 1 constructs the automaton from basic building
      blocks, based on the user-defined paradigm sets,
      target pattern, and max error number.            */
1  CasMacroLib lib ;                  // Cascadable macro library
2  Procedure PRDM_Atma_Con (ParadigmSet ps, Pattern pat, int err_num)
3      Automaton atma;
4      BuildingBlock block;
5      for Paradigm p in ps do
6          block.add(p);
7      int m = err_num, n = pat.length;
8      atma.row_num = m + 1; atma.col_num = n - m;
9      atma ⟵ block.duplicate(m×n);
10     for int i ← 0 to m + 1 do
11         for int j ← 0 to n - m do
12             ADD_TRANSITIONS(atma, i,j, ps);
13     atma.set_start(); atma.set_report();
14     #ifdef ENABLE_MACRO        /* Cascadable macro constr. */
15         CasMacro cmacro;
16         for Paradigm p in ps do
17             ADD_PORTS(atma, p);
18         MERGE_PORTS(atma);
19         cmacro ⟵ AP_CompileMacros(atma);
20         lib.add(cmacro);
21     #endif
22     for int i ← 0 to m + 1 do
23         for int j ← 0 to n - m do
24             atma.re_label(i, j, pat.c(j+i));
25     return atma;
```

B. Design Cascadable Macros

Besides the execution time, AP has overhead for the time-consuming recompilation including the *place-&-route* process [16]. The pre-compiling technique can build macros in advance to hide such overhead; however, it has quite limited usage. For example, we assume the AP automaton in Fig. 3 (for the pattern "object", allowing up to 2 errors) is pre-compiled as a macro M_1 with all STEs parameterized. For any new patterns with the same number of characters and maximum errors, e.g., "gadget" and up to 2 errors, we can reuse M_1 by relabeling STEs. However, an recompilation is inevitable if any of these requirements is not met. Thus, the pattern "gadgets" with up to 2 errors or "gadget" with up to 3 errors would all fail to directly take advantage of this macro M_1.

In the Robotomata, we propose *cascadable macros* to support reuse of macros for large-scaled AP automata. In particular, we connect one or more macro instances to compose a larger and different AP automaton through our carefully-designed interconnection algorithm. With the cascadable macros, the overhead of reconfiguration can be significantly reduced, since only the connections between instances need to be placed-&-routed. Our method generates the desired automata by connecting macros if the target block matrix can be composed by multiple smaller block matrices. For example, the AP automata in Fig. 3 is stored as a cascadable macro M_2 having a $(3 * 4)$ block matrix. Assume we will build an automata for a larger pattern "international" with 13 characters and up to 5 errors. The target block matrix is a $(5 + 1) * (13 - 5) = 6 * 8$ matrix, and thus, the AP automata can be built by connecting 4 M_2 macro instances.

Builidng Cascadable Macros: The first step to create cascadable macros is to add input/output ports. The optimal design of adding ports should minimize (1) the number of total ports, and (2) the *in-degree* of each input port, because both the number of ports and the number of signals that go into an input port are limited in AP hardware [38]. We define the port struct that has three attributes:

- ***Port role*** identifies the port is used to either input or output signals (*in*=input, *out*=output).
- ***Cascade direction*** indicates the direction of allowed cascade, and the side of two paired macros (*h*=horizontal, *v*=vertical, *d*=diagonal, *ad*=anti-diagonal).
- ***Transition scope*** represents whether connections can cross the edges of neighboring macro to link non-adjacent STEs (*e*=edge of neighbors, *ce*=cross-edge of neighbors, *e/ce*=both).

As mentioned in the previous section, multiple consecutive errors can be generated from two-consecutive errors. Therefore, we use Tab. IV to list out all STE↔port connection rules in an AP automata having $(m + 1) * (n - m)$ block matrix (corresponding to m errors at most and n pattern length). A building block is represented by $B_{i,j}$. The input/output ports always appear in pairs, representing the two sides of each connection. The STE↔port connections can be categorized into two groups. The first one is the *one-to-one* connection. This category is relatively straightforward: for example, for the paradigm M*atch*, the output of one macro can be the input of the following macro, and one character can only be connected to the following character in the given pattern. Therefore, the Robotomata adds the input and output ports to the exact-character STEs of blocks in the first column ($B_{i,0}.c$) and last column ($B_{i,n-1}.c$), respectively. The rule is shown in the table as $Iy \rightarrow B_{i,0}.c$ and $B_{i,n-1}.c \rightarrow Ox$. The second group is the *N-to-one* connection. To meet the ANML macro building requirements and optimize the port usage [38], we introduce *or* Boolean gates to this category. For example, in the case of the fifth port design of paradigm I*nsert*, rather than adding a new port as $B_{k,0}.a \rightarrow Ox$ for each row k, we use an *or* Boolean gate to allow a combination of connections as $B_{k,0}.a \rightarrow or \rightarrow Ox$, so as to reduce the number of used port to be only one. This design also bypasses a restriction of AP routing, i.e., no more than one transition is allowed to directly go to a single output port [38].

In the final step of the port design for a macro, we search for and merge the ports with inclusive STE↔port connections.

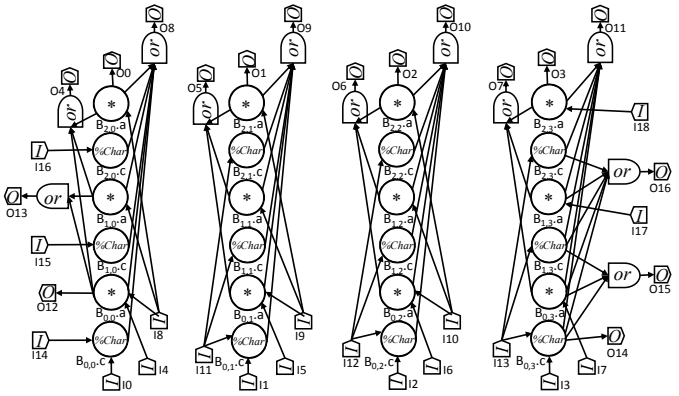


Fig. 6: Port design layout for a three layers four columns macro.

TABLE IV: Port design rules according to paradigm

Paradigm	Port Attribute			STE↔port Connection
	Dir.	Scope	Role	
M	*h*	*e*	*out*	$B_{i,n-m}.c \rightarrow Ox$
	h	*e*	*in*	$Iy \rightarrow B_{i,0}.c$
S, SS/SI	*v*	*e*	*out*	$B_{m+1,j-1}.a \rightarrow Ox$; $B_{m+1,j}.a \rightarrow Ox$
	v	*e*	*in*	$Iy \rightarrow B_{0,j+1}.c$; $Iy \rightarrow B_{0,j}.a$
	d	*e*	*out*	$B_{m+1,n-m}.a \rightarrow Ox$
	d	*e*	*in*	$Iy \rightarrow B_{0,0}.c$
	h	*e*	*out*	$B_{i,n-m}.a \rightarrow Ox$
	h	*e*	*in*	$Iy \rightarrow B_{i,0}.c$
I, II	*v*	*e*	*out*	$B_{m+1,j}.a \rightarrow Ox$; $B_{m+1,j+1}.a \rightarrow Ox$
	v	*e*	*in*	$Iy \rightarrow B_{0,j}.c$; $Iy \rightarrow B_{0,j-1}.a$
	ad	*e*	*out*	$B_{m+1,0}.a \rightarrow Ox$
	ad	*e*	*in*	$Iy \rightarrow B_{0,n-m}.a$
	h	*e*	*out*	for k=0 to $i-1$ $B_{k,0}.a \rightarrow or \rightarrow Ox$
	h	*e*	*in*	$Iy \rightarrow B_{i+1,n-m}.a$
D, DD	*v*	*e/ce*	*out*	for k=0 to $m+1$ $B_{k,j-1}.c \rightarrow or \rightarrow Ox$
	v	*e/ce*	*in*	for k=0 to $m+1$ $Iy \rightarrow B_{k,j+1}.c$
	d	*e/ce*	*out*	for k=0 to $m+1$ $B_{k,n-m}.c \rightarrow or \rightarrow Ox$
	d	*e/ce*	*in*	$Iy \rightarrow B_{i,0}.c$
	h	*e*	*out*	for k=0 to $i-1$ $B_{k,n-m}.c \rightarrow or \rightarrow Ox$
	h	*e*	*in*	$Iy \rightarrow B_{i+1,0}.c$
SD	*v*	*e/ce*	*out*	for k=0 to $m+1$ $(B_{k,j-1}.c, B_{k,j-1}.a) \rightarrow or \rightarrow Ox$
	v	*e/ce*	*in*	for k=0 to $m+1$ $Iy \rightarrow B_{k,j+1}.c$
	d	*e/ce*	*out*	for k=0 to $m+1$ $(B_{k,n-m}.c, B_{k,n-m}.a) \rightarrow or \rightarrow Ox$
	d	*e/ce*	*in*	$Iy \rightarrow B_{i,0}.c$

Then, the attribute sets of merged port equal to the union of all participant ports. This can further optimize the port usage. The procedure of constructing cascadable macros is integrated into Alg. 1 from ln. 14 to ln. 21. Two predefined functions are implemented: (1) `ADD_PORTS()` adds input/output ports to given automata based on each paradigm (ln. 16-17), followed by STE↔port connections as described in Tab. IV; and (2) `MERGE_PORTS()` optimizes the ports layout by merging equivalent and inclusive ports (ln. 18). Fig. 6 exhibits the completed ports layout for the cascadable macro M_2[1].

C. Cascadable Macro based Construction Algorithm

After the pre-compilation, the cascadable macros are inserted into a macro library for reuse (ln. 19-20). In our current design, the library contains macros whose pattern lengths (columns) are in three groups of {1,2, ..., 9} {10, 20, ..., 90} {100, 200, ..., 900} with each one allowing up to 3 errors (layers). The reasons of choosing these $27 \times 3 = 81$ macros are two-fold: (1) It is inefficient and even impractical to save all possible situations for different combinations of pattern sizes and errors. (2) Many real-world cases focus on pattern sizes $<$1000 characters and errors $\leqslant$3 (see Sec. V). For the rare cases with longer patterns or larger error allowance, we can handle them by using more macro instances.

Based on the cascadable macro library, we can effectively reduce the construction and compilation overhead and efficiently build AP automata. Alg. 2 shows our macro-based AP automata construction. In this algorithm, we search the library for a hit macro, which has the same structure with the desired automaton. If found, the macro can be directly instantiated (ln. 5-6). Otherwise, we use multiple macros to form the desired automaton. First, we select and instantiate proper macros according to the dimensions of the desired automaton (ln. 8-11). Second, these macro instances are organized to a lattice (ln. 12-16). Third, we link these instances to generate the complete AP automaton (ln. 17-27). The function `CONN_INST(ParadigmSet, src, dst, dir, scope)` is predefined to make cascade links of input-output port pairs from the source instance `src` to the destination instance `dst`. The arguments of `dir` and `scope` are used as a filter, meaning only the ports with provided attribute values are qualified for linking. Finally, the AP automaton can be constructed after the relabeling process (ln. 29-31).

V. Evaluation

We evaluate our Robotomata by using the Levenshtein automata construction since it includes a full paradigm set of M, S, I, D. In contrast, the other distances (e.g., Hamming) only need to use a subset of the paradigms and simpler routing, so that have less compilation overhead than Levenshtein. Notice that changing distance type for a given dataset in the Robotomata is lightweight since it can automatically generate AP automata for a given dataset with different distance types through the simple paradigm set change. We run our experiments on a platform equipped with Intel Xeon E5-2637 CPU @ 3.5 GHz and 256 GB main host memory. The installed AP SDK is in version 1.6.5. Currently, the Micron AP is not ready for production; thus, we use an emulator[2] to estimate the runtime performance. We enable the cascadable macro library in our Robotomata and generate AP automata using Alg. 2. The size of the library follows the scheme in Sec. IV-C. We also set the macros in the library to accept up to three errors. That way, the higher error number (e.g., $>$ 3) will need not only horizontal instance cascading but also vertical cascading.

A. Synthetic Patterns

To evaluate the Robotomata on processing different patterns and error allowances, we use a set of six synthetic patterns of

[1]Transitions between STEs inside a macro are skipped to highlight the port connections.

[2]http://www.micronautomata.com

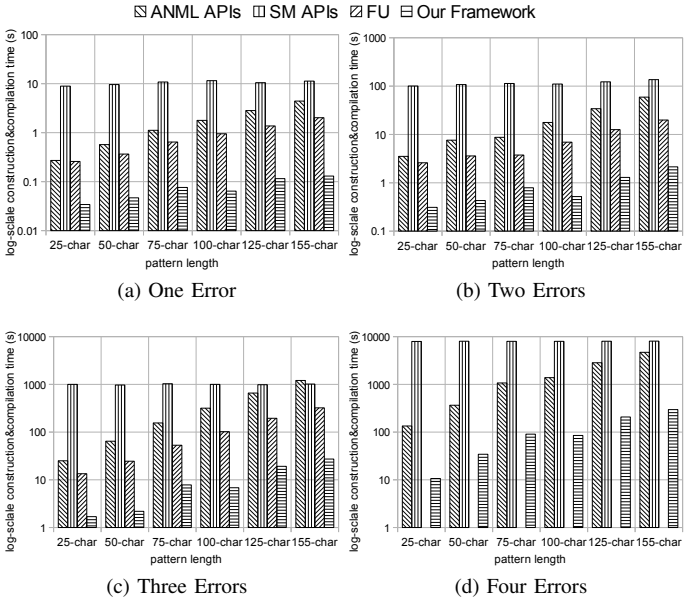
Algorithm 2: Cascadable Macro-based AP Automata Construction

```
/* Alg. 2 constructs automata based on our cascadable
   macro library, whose macros are built by ln. 14-21 in
   Alg. 1.                                              */
1  MacroLib lib;
2  Procedure CASM_Atma_Con(ParadigmSet ps, Pattern pat, int err_num)
3      Automaton atma;
4      int m = err_num, n = pat.length;
5      if lib.search(ps, m+1, n−m) = true then
6          Instantiate(ps, atma, m+1, n−m, true, true);
7      else
8          int digits[] = {(n−m) /100, (n−m) %100/10, (n−m) %10};
9          int quo = (m+1) / lib.max_err, rem = (m+1) % lib.max_err;
10         int ins_col = (bool)digits[0] + (bool)digits[1] + (bool)digits[2];
11         Automaton inst[quo+(bool)rem][ins_col];
12         if quo then
13             for int i ← 0 to (quo−1) do
14                 Gen_inst_row(ps, inst[i], lib.max_err, digits);
15         if rem then
16             Gen_ins_row(ps, inst[quo], rem, digits);
17         for int i ← 0 to (quo+(bool)rem−1) do
18             for int j ← 0 to (ins_col−1) do
19                 if j then CONN_INST(ps, inst[i][j−1], inst[i][j], h, e);
20                 for int k ← 0 to (i−1) do
21                     TransitionRange scope;
22                     if k=(i−1) then scope=e else scope=ce;
23                     CONN_INST(ps, inst[k][j], inst[i][j], v, scope);
24                     if j≠(ins_col−1) then
25                         CONN_INST(ps, inst[k][j], inst[i][j+1], d,
                               scope);
26                     if j≠0 then
27                         CONN_INST(ps, inst[k][j−1], inst[i][j], ad,
                               scope);
28         atma ⟵ inst;
29     for int i ← 0 to m+1 do
30         for int j ← 0 to n−m do
31             atma.re_label(i, j, pat.c(j+i−1));
32     return atma;
33 Function Instantiate(ParadigmSet ps, Automaton am, int row, int col, bool
     start, bool report)
34     CasMacro cmacro;
35     cmacro ⟵ lib.pick(ps, row, col, start, report);
36     am ⟵ cmacro.instantiate();
37 Function Gen_inst_row(ParadigmSet ps, Automaton ams[], int err, int
     digits[])
38     int j = 0;
39     for int i ← 0 to (ams.size−1) do
40         while j<digits.size do
41             if digits[j] then
42                 Instantiate(ps, ams[i], err, digits[j], !i, !(digits.size−1−j));
43                 j++; break;
44             j++;
```

lengths from 25 to 155 in steps of 25. The error allowances range from one to four. In Fig. 7, the construction and compilation time of our framework is compared with other three AP automata construction approaches: Functional Units (FU) [18], String Matching APIs (SM_API) [6], and basic ANML APIs (ANML). The first one uses partially optimized macros, while the latter two use conventional macros. The AP runtime performance is not considered in this section, due to its lock-step execution mode leading to linear relation between runtime cost and input length and independence to automata construction approaches.

The compilation data of FU is not available for four errors [18]. In Fig. 7, we first observe that the compilation costs of all approaches increase exponentially as the error number rises. This is because more errors require higher STE usage and more complicated placement-&-routing. SM_API is a "black-box" approach provided by Micron and its performance is insensitive to the pattern length, meaning high compilation time is needed even for small patterns. The FU and ANML approaches show positively correlated pattern length and compilation cost, for the similar reason with aforementioned error-cost relationship. In contrast, the compilation cost of our framework is more relates to the instance number than the pattern length. Notice that the instantiation of a larger pre-compiled macro usually causes higher cost.

Overall, the SM_API approach provides higher abstraction and is easier for developers to use with the compromised more expensive construction and compilation. FU approach can achieve up to 3.7x speedups against the ANML approach. On the other hand, our Robotomata can achieve up to 46.5x speedups over the ANML. For four error case, since the error number exceeds library's threshold, we should apply both vertical and horizontal instance cascades that cause higher compilation overhead; even so the Robotomata can still provide up to 16.3x speedups over the ANML baseline. In other word, our Robotomata with cascadable macros can better explore the AP capacity than other designs.



(a) One Error (b) Two Errors

(c) Three Errors (d) Four Errors

Fig. 7: Construction and compilation time vs. pattern lengths. The highest speedups of Robotomata over FU, ANML_APIs, SM_APIs are highlighted.

B. Real-world Dataset

We also evaluate the Robotomata with two real-world datasets. The "Bio" is the BLAST *igseqprot* dataset[3] for sequence alignment, and "IR" is the NHTSA *Complaints* dataset[4] for information retrieval. Tab. VI describes the characteristics of the two datasets. The "Bio" contains ˜85K patterns ranging from 32 to 686 characters, while the "IR" has ˜100K patterns from 10 to 959 characters. Apparently, the overall demand for STEs of either dataset exceeds the capacity of a single AP board. For example, the "Bio" dataset with one error requires over 4M STEs while single AP board supplies only 1.5M STEs. This supply-demand imbalance forces us to use multiple flush/load reconfigurations to completely process the

[3]https://www-odi.nhtsa.dot.gov/downloads/flatfiles.cfm
[4]ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/

TABLE V: Compilation Profile

#Err	Approach	Bioinformatics (Bio)					Information Retrieval (IR)				
		Recompile round	Relabel round	Mean STE usage(%)	Mean BR alloc(%)	Total compile time (s)	Recompile round	Relabel round	Mean STE usage(%)	Mean BR alloc (%)	Total compile time (s)
1 err	ANML	1	34	71.2	73.5	103	2	79	70.3	75.8	329
	FU	1	56	44.7	80.1	53.8	3	131	42.4	80.7	162
	Robotomata	1	36	68.3	79.2	5.83	2	89	62.6	80.4	12.8
2 errs	ANML	1	61	61.2	80.9	665	3	148	56.3	81.4	3763
	FU	2	95	39.6	89.3	335	4	228	36.8	92.2	996
	Robotomata	1	71	52.8	82.7	48.1	3	174	48.3	86.4	110
3 errs	ANML	2	105	47.5	81.7	7585	4	242	46.2	84.2	42340
	FU	3	149	33.5	91.4	3634	6	323	34.6	95.1	18780
	Robotomata	2	119	42.2	85.9	381	5	281	39.5	89.4	1070
4 errs	ANML	3	141	44.2	86.3	43201	6	353	39.6	87.9	135437
	FU	Beyond the capacity									
	Robotomata	3	168	37.4	88.6	3607	8	460	30.4	90.5	12760

dataset. Since the constructed automata can be dynamically stored as a macro in the library, we can directly reuse them for all the patterns with the same length and errors. Therefore, we only consider the patterns with different lengths.

TABLE VI: Dataset Characteristics

Pattern Sets	Pattern#	Length (min.)	Length (max.)	Diff_lengths#
Bio	85389	32	686	375
IR	$100K$	10	959	846

Compilation Profiling:

We compare our Robotomata to two other AP construction approaches: FU and ANML-APIs. The SM_APIs method is not used because it hides the compiling profile (e.g., STE usage) so that the rounds of reconfiguration are uncomputable. Tab. V shows the profiles of the three approaches. We test the two datasets "Bio" and "IR" with up to 4 maximum error allowances. Notice the FU supports up to three errors. In the table, the "STE usage" is used to estimate the number of reconfiguration rounds. Generally, higher STE usage indicates less reconfiguration rounds. On the other hand, the STE usage is negatively correlated to the placement-&-routing complexity, which is represented by "BR allocation". Obviously, longer patterns and more errors will result in larger automata, thereby giving rise to higher BR allocation (i.e. more complex compilation) and lower STE usage.

For reconfiguration rounds, we distinguish the "recompile" rounds from the light-weight "relabel" rounds. Relabeling the entire AP board costs 45 ms [7], while the recompiling time depends on AP automata complexity. The "total compile time" columns include the cost from both recompile and relabel rounds. ANML-APIs show the best STE usage and thus the least reconfiguration rounds because they use the low-level interfaces to accurately manipulate AP computational resources. However, the ANML-APIs take the longest overall compile time since they use only conventional macros resulting in highest reconfiguration costs. In contrast, the FU approach oftentimes presents the lowest STE usage with the highest reconfiguration rounds, due to its large number of functional unit copies and associated complex inter-connections. Nevertheless, the total compile time of FU approach can be still shorter than ANML-APIs since the functional units can reduce more placement-&-routing costs than conventional macros. On the other hand, our Robotomata shows slightly lower STE usage than ANML-APIs but achieves the fastest total compile time thanks to our cascadable macro design. In particular, it can achieve up to 39.6x and 17.5x speedups against ANML-APIs and FU approaches respectively.

Performance Comparison:

In Fig. 8 and Fig. 9, we compare the performance of AP approaches to an automata-based CPU implementation *PatMaN* [39] over the two datasets. PatMaN allows all error types with no upper-bound error number. We first compare the pure computational time between AP and CPU. As we discussed, after loading the binary image to AP board, AP executes in a lock-step style making the runtime performance linear to input length and reconfiguration rounds and independent of automata construction approach. Fig. 8 shows our APM codes on AP can achieve up to 4370x speedups, which are within the same order of magnitude with the other AP-related work [7], [10].

In Fig. 9, we conduct a more fair comparison using the overall execution time (i.e., runtime and compiling time) of AP. We can observe that these AP approaches can generally outperform the CPU implementation in the cases of large error number. However, ANML-APIs may be slower than CPU version for small error number (e.g. in "One Error Bio"). The FU approach can provide better performance than ANML-APIs but still fails to outperform the PatMaN in some cases (e.g., in "One Error IR"). This shows the significance of the reconfiguration overhead when processing large-scale datasets. On the other hand, our Robotomata can outperform the CPU implementations and other AP approaches in all cases. Specifically, it is able to give an improvement of 2x to 461x speedups over CPU PatMaN and up to 33.1x and 14.8x speedup over ANML-APIs and FU respectively. In summary, with the Robotomata, we can conduct performance comparison being fair to both CPU and AP by including the ultimately optimized AP reconfiguration costs.

VI. Related Work

Large effort has been invested to map automata on various parallel architectures, including CPUs [40], [41], [42],

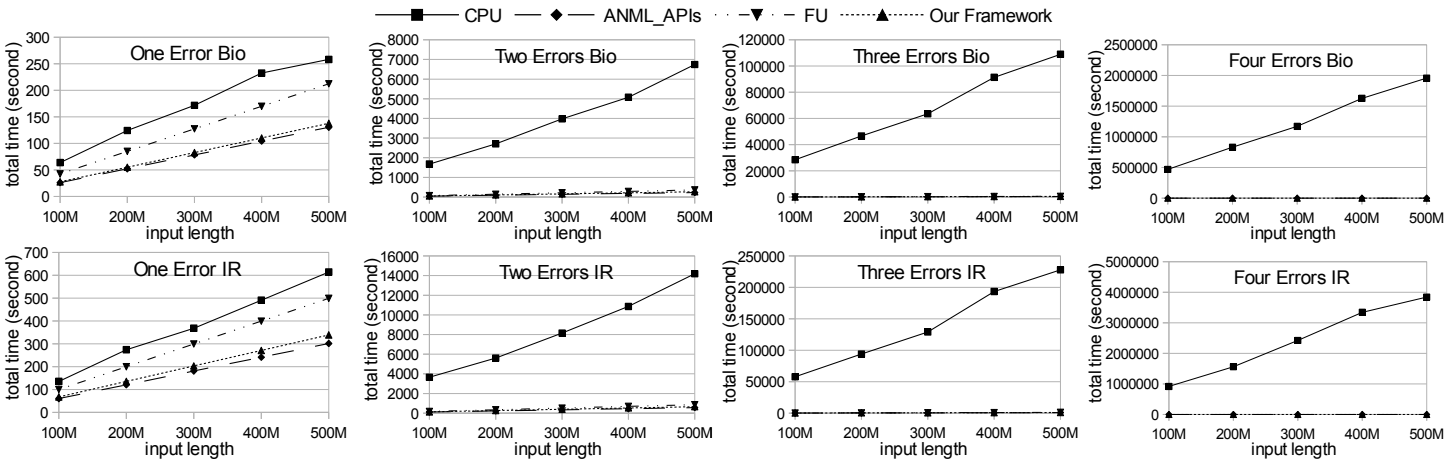


Fig. 8: Computational time comparison between AP (with three different construction approaches) and the CPU counterpart. Notice that FU approach can't support more than three errors.

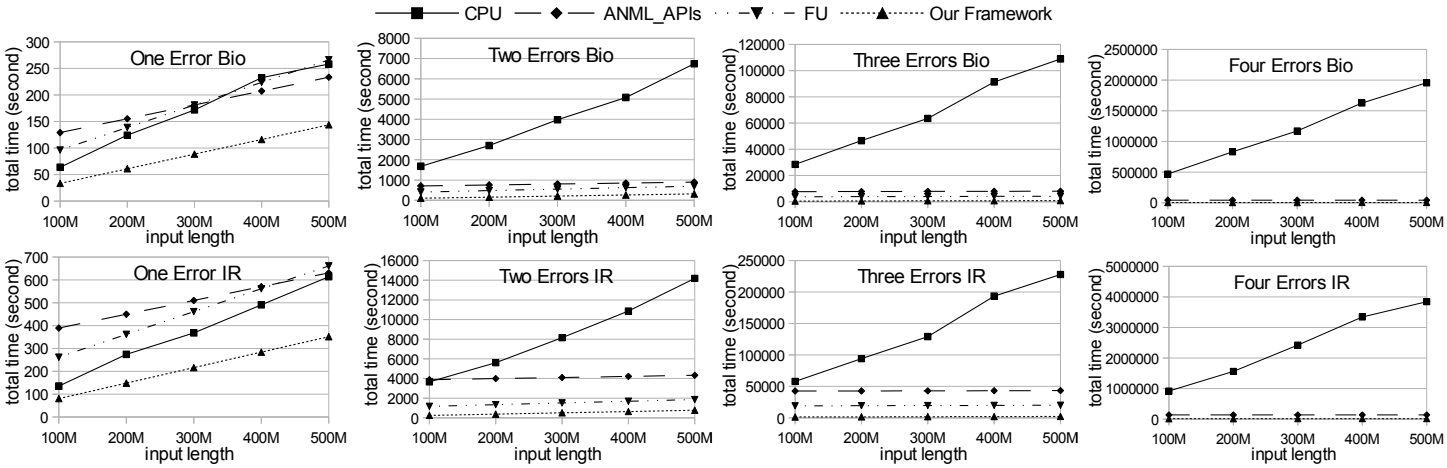


Fig. 9: Overall time comparison between AP (with three different construction approaches) and CPU implementation. Notice that FU approach can't support more than three errors.

GPUs [23], [43], FPGAs [25], and ASICs [44]. Various speedups against implementations on single-threaded processors, however, are either not fully leverage the merits of employed hardware or based on an architecture that still in design stage.

Recent studies have revealed Micron AP can improve performance for many applications from data mining [7], [45], machine learning [9], bioinformatics [8], [15], [46], intrusion detection [10], graph analysis [11], and so on[12], [13], [14]. Among them, FU [18], SM APIs [6], RAPID [16], and ANMLZoo [47] are highly related with this work.

Tracy et al. [18] propose a functional unit (FU) approach to accelerate APM problems. They decompose a Levenshtein automata to 8 FUs, then represent FUs in ANML and pre-compile&save them as macros. This approach can benefit all Levenshtein automata by reducing placement&routing overhead to some extent. However, it still requires significant inter-instance routing and compiling time for large-scale automata as shown in our experiments.

String Matching (SM) APIs [6] are included in AP SDK recently to provide high-level abstracts for APM. Taking user-provided patterns and distances, SM APIs can generate binary images accordingly. However, if either the given distance or the pattern length doesn't fit any template in the library of SM APIs, it is necessary to construct automata and place-and-route them from scratch during the compilation. As shown in our experiments, SM APIs cannot support some large-scale problem sizes.

Angstadt et al. [16] propose RAPID, a high-level programming model for AP. RAPID bypasses the pre-compiled macros; instead, RAPID places and routes a small repetitive portion of the whole program, saves it as a small binary image, and loads this portion as many times as need on the AP board. However, this scheme may underperform pre-compiling strategy in the case that the desired pattern is large and its pattern lengths vary, because the repetitive portion of program is usually smaller than a macro. Another drawback of this method is losing the flexibility of cascadable macros once the portion of program is saved as a binary image which can be only loaded and executed.

Wadden et al . [47] propose ANMLZoo, a benchmark suite for automata processors. It has a subset targeting on APM, and explores how different distance types, error numbers and pattern lengths affect fan-in/fan-out of each STE then further affect configuration complexity. However, it doesn't fully explore the AP's capacity since doesn't leverage any pre-compiled information. Moreover, users still need to manually manipulate the AP automata when extending the benchmarks.

VII. Conclusions

In this work, we reveal the problems of programmability and performance on AP, especially the reconfiguration cost for large-scale problem sizes. We use a hierarchical approach to design and implement Robotomata, a framework that can generate optimized low-level codes for Approximate Pattern Matching applications on AP. We evaluate Robotomata by comparing to state-of-the-art research with the real-world datasets. The experimental results illustrate the improved programming productivity and significantly reduced configuration time.

Acknowledgements

This work was supported in part by NSF I/UCRC IIP-1266245 via NSF CHREC and the SEEC Center via the Institute for Critical Technology and Applied Science (ICTAS).

References

[1] P. Mcgarry. (2016) Outsmarting google search: making fuzzy search fast and easy without indexing. [Online]. Available: https://www.ryft.com/blog/

[2] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*

[3] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, 2009.

[4] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 2014.

[5] *PCRE Programmers Reference*, Micron Technology. [Online]. Available: http://www.micronautomata.com/apsdk_documentation/latest/h1_pcre.html

[6] *String Matching Programmers Reference*, Micron Technology. [Online]. Available: http://www.micronautomata.com/apsdk_documentation/latest/h1_string.html

[7] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the micron automata processor," in *ACM Int. Conf. Comput. Front. (CF)*, 2016.

[8] I. Roy and S. Aluru, "Finding motifs in biological sequences using the micron automata processor," in *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2014.

[9] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the automata processor," in *Int. Conf. High Perf. Comput. (ISC), Springer*, 2016.

[10] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, "High performance pattern matching using the automata processor," in *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2016.

[11] I. Roy, N. Jammula, and S. Aluru, "Algorithmic techniques for solving graph problems on the automata processor," in *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2016.

[12] M. H. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda, "Fast track pattern recognition in high energy physics experiments with the automata processor," *arXiv preprint arXiv:1602.08524*, 2016.

[13] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron, "Generating efficient and high-quality pseudo-random behavior on automata processors," in *IEEE Int. Conf. Computer Des. (ICCD)*, 2016.

[14] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity resolution acceleration using the automata processor," in *IEEE Int. Conf. Big Data (BigData)*, 2016.

[15] C. Bo, K. Wang, Y. Qi, and K. Skadron, "String kernel testing acceleration using the micron automata processor," in *Int. Workshop Computer Archit. Mach. Learn. (CAMEL)*, 2015.

[16] K. Angstadt, W. Weimer, and K. Skadron, "Rapid programming of pattern-recognition processors," in *ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2016.

[17] *ANML Programmers Reference*, Micron Technology. [Online]. Available: http://www.micronautomata.com/apsdk_documentation/latest/h1_anml.html

[18] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins, "Nondeterministic finite automata in hardware : the case of the levenshtein automaton," in *Int. Workshop Archit. Syst. Big Data (ASBD)*, 2015.

[19] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2014.

[20] X. Yu, B. Lin, and M. Becchi, "Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence," *IEEE J. Selected Areas in Commun.*, 2014.

[21] X. Yu, W.-c. Feng, D. Yao, and M. Becchi, "O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection," in *ACM/IEEE Symp. Archit. Networking Commun. Syst. (ANCS)*, 2016.

[22] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *ACM/IEEE Symp. Archit. Networking Commun. Syst. (ANCS)*, 2007.

[23] X. Yu, *Deep packet inspection on large datasets: algorithmic and parallelization techniques for accelerating regular expression matching on many-core processors*. University of Missouri-Columbia, 2013.

[24] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying automata processing: Gpus, fpgas or microns ap?" in *ACM Int. Conf. Supercomput. (ICS)*, 2017.

[25] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling pcre to fpga for accelerating snort ids," in *ACM/IEEE Symp. Archit. Networking Commun. Syst. (ANCS)*, 2007.

[26] X. Cui, T. R. Scogland, B. R. de Supinski, and W.-c. Feng, "Directive-based partitioning and pipelining for graphics processing units," in *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2017.

[27] J. Zhang, S. Misra, H. Wang, and W. chun Feng, "Eliminating irregularities of protein sequence search on multicore architectures," in *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2017.

[28] X. Yu, H. Wang, W. C. Feng, H. Gong, and G. Cao, "cuart: Fine-grained algebraic reconstruction technique for computed tomography images on gpus," in *IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, 2016.

[29] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "An enhanced image reconstruction tool for computed tomography on gpus," in *ACM Int. Conf. Comput. Front. (CF)*, 2017.

[30] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on gpus," in *ACM Int. Conf. Supercomput. (ICS)*, 2017.

[31] K. Hou, H. Wang, and W.-c. Feng, "Gpu-unicache: Automatic code generation of spatial blocking for stencils on gpus," in *ACM Int. Conf. Comput. Front. (CF)*, 2017.

[32] J. Zhang, H. Wang, and W.-c. Feng, "cublastp: Fine-grained parallelization of protein sequence search on cpu+gpu," *IEEE/ACM Trans. Comput. Bio. Bioinf. (TCBB)*, 2015.

[33] *Virtex-6 Family Overview*, Xilinx, Inc. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[34] K. Hou, H. Wang, and W.-c. Feng, "Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors," in *ACM Int. Conf. Supercomput. (ICS)*, 2015.

[35] J. Zhang, H. Wang, H. Lin, and W.-c. Feng, "cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU," in *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2014.

[36] X. Yu and M. Becchi, "Gpu acceleration of regular expression matching for large datasets: Exploring the implementation space," in *ACM Int. Conf. Comput. Front. (CF)*, 2013.

[37] Baeza-Yates and R. G. Navarro, "Faster approximate string matching," *Algorithmica*, 1999.

[38] *Guidelines for Building a Macro*, Micron Technology. [Online]. Available: http://www.micronautomata.com/documentation/anml_documentation/c_macros.html

[39] K. Prfer, U. Stenzel, M. Dannemann, R. E. Green, M. Lachmann, and J. Kelso, "Patman: rapid alignment of short sequences to large databases," *Oxford Bioinf.*, 2008.

[40] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *ACM/IEEE Symp. Archit. Networking Commun. Syst. (ANCS)*, 2009.

[41] J. Qiu, Z. Zhao, and B. Ren, "Microspec: Speculation-centric fine-grained parallelization for fsm computations," in *Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2016.

[42] J. Qiu, Z. Zhao, B. Wu, A. Vishnu, and S. L. Song, "Enabling scalability-sensitive speculative parallelization for fsm computations," in *ACM Int. Conf. Supercomput. (ICS)*, 2017.

[43] X. Yu and M. Becchi, "Exploring different automata representations for efficient regular expression matching on gpus," *ACM SIGPLAN Not.*, 2013.

[44] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: The unified automata processor," in *ACM Int. Symp. Microarchit. (MICRO)*, 2015.

[45] E. Sadredini, R. Rahimi, K. Wang, and K. Skadron, "Frequent subtree mining on the automata processor: Challenges and opportunities," in *ACM Int. Conf. Supercomput. (ICS)*, 2017.

[46] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms," in *IEEE Int. Symp. High-Perf. Computer Archit. (HPCA)*, 2018.

[47] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *IEEE Int. Symp. Workload Charact. (IISWC)*, 2016.