LAP: A Lightweight Automata Processor for Pattern Matching Tasks

Haojun Xia, Lei Gong, Chao Wang, Xianglan Chen and Xuehai Zhou
School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
xhjustc@mail.ustc.edu.cn, {leigong0203, cswang, xlanchen, xhzhou}@ustc.edu.cn

***Abstract*—Growing applications are employing finite automata as their basic computational model. These applications match tens to thousands of patterns on a large amount of data, which brings great challenges to conventional processors. Hardware-based solutions have achieved high throughputs automata processing. However, they are too heavy to be integrated into small chips. Besides, they have to rely on DRAMs or other high capacity memories to store their underlying automata models. We focus on building a more lightweight automata processor, which can store the whole automata model into SRAMs with limited size and run independently. We propose LAP, a lightweight automata processor. Extremely high storage efficiency is achieved in LAP, leveraging a novel automata model (ADFA) and efficient packing algorithms. Besides, we exploit software-hardware co-design to achieve faster processing speed. We observe that ADFA's traversal algorithm is parallelizable. Thus, we propose novel hardware instructions to parallel the additional memory accesses in ADFA model and hide their access overhead. LAP is organized into a four-stage pipeline and prototyped into Xilinx Artix-7 FPGA at 263 MHz frequency. Evaluations show that LAP achieves extremely high storage efficiency, exceeding IBM's RegX and Micron's AP by 8×. Besides, LAP achieves significant improvements in processing speed ranging from 32% to 91% compared with previous lightweight implementations. As a result, a low-power CPU equipped with five LAP cores can achieve 9.5 Gbps processing throughput matching 400 patterns simultaneously.**

***Index Terms*—Automata Processor, Software-hardware Co-design, Pattern matching, Field Programmable Gate Array**

I. Introduction

Finite automata [1] is an important mathematical concept which has been applied to diverse fields such as textual data mining [2], network security [3], computational biology [4], and particle physics [5]. These applications utilize automata as the essential computational model, matching tens to thousands of patterns on a large amount of input data. Therefore, the performance of automata processing is crucial. However, conventional software solutions utilizing CPUs and GPUs struggle to meet current requirements for high-performance automata processing. Unpredictable memory references and control transfers in automata processing significantly reduce their performance. Besides, general-purpose instructions in CPUs and GPUs are not efficient for automata processing.

Prior hardware-based solutions have achieved high throughputs automata processing. Micron's automata processor (AP) [6] is a classic implementation based on nondeterministic finite automata (NFA) model. It is an adaptation of memory array architecture in traditional SDRAMs. Since NFAs impose high processing complexity, AP exploits massive parallelization in its spatial architecture to perform tremendous operations concurrently. In this way, AP achieves remarkable performance in NFA based processing. Furthermore, many implementations [7] [8] are proposed based on AP's architecture. However, these implementations require extremely high internal memory bandwidth and massively parallel processing units. Besides, AP-like implementations cost many hardware resources to build their complex interconnections among the processing units. Therefore, they are not suitable to be integrated into small chips. In contrast, IBM's RegX [9] is a typical deterministic finite automata (DFA) based hardware accelerator, which is part of IBM's PowerEN processor. However, it still relies on external DRAMs to store its automata model.

We focus on building a more lightweight automata processor, which can store the whole automata model into its small SRAMs and run independently. It can be integrated with CPUs supporting efficient and high-performance pattern-matching tasks without accessing external memories. One recent effort [10] has utilized efficient algorithms to achieve high storage efficiency. However, it struggles to achieve a fast processing speed. We observe that significant improvements can be obtained in storage efficiency by exploiting algorithmic innovations in automata models. Thus, we utilize ADFA model [11], which is an augmented DFA model, for higher storage efficiency. However, complex automata models require more sophisticated hardware operations, which results in slower processing speed. Therefore, we exploit software-hardware co-design to alleviate such performance losses. We propose LAP, a lightweight automata processor without relying on DRAMs or other high capacity memories. LAP achieves extremely high storage efficiency and stores the whole automata model into its SRAMs with limited size. More importantly, it achieves significant improvements in processing speed compared with previous lightweight implementations.

In summary, we make the following contributions:

- Extremely high storage efficiency is achieved in LAP, exploiting ADFA model and efficient packing algorithms. Besides, we exploit software-hardware co-design to achieve faster processing speed. We observe that ADFA's traversal algorithm is parallelizable. Thus, we propose novel hardware instructions to parallel ADFA's memory accesses in run time. Corresponding compilation algorithms are also implemented.

- We propose a four-stage pipeline to implement LAP at a higher hardware frequency. Fine-grained multithreading is utilized to deal with pipeline dependencies, which contributes to a stall-free pipeline.
- We implement LAP and prototype it into Xilinx Artix-7 FPGA at 263 MHz frequency. Evaluations show that its storage efficiency exceeds IBM's RegX and Micron's AP by 8×. Besides, each LAP core can process more than 0.9 characters per cycle, surpassing previous lightweight implementations ranging from 32% to 91%.

II. Background

A. Preliminaries of Automata Models

Finite automata is an effective algorithmic tool for pattern matching tasks. We can map thousands of patterns into a single finite automaton and feed the automaton with the character-stream that we want to process. In this way, we can match all the patterns on a large amount of data simultaneously instead of matching them by turns. Deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs) are the most classic automata models. An NFA is represented by a 5-tuple, $(Q, \Sigma, \delta, q0, F)$, where Q is a finite set of states, Σ is a finite set of characters, δ is a transition function, $q0 \in Q$ is an initial state, and $F \subset Q$ is a set of accepted states. The transition function $\delta(q, \alpha)$ defines a set of states that can be reached from state q when the character α is received as the input of the automaton. As a result, there can be more than one active state concurrently in NFA model. In contrast, the transition function $\delta(q, \alpha)$ in DFA model only maps from one state to another one instead of a set of states. Thus, there will always be only one active state in DFA model.

NFAs have limited size but require expensive per-character processing, whereas equivalent DFAs have lower processing complexity at the cost of larger size. To achieve better trade-offs between speed and automaton size, many augmented automata models have been proposed based on DFA and NFA models. Among them, we choose ADFA model [11] as the main underlying model in this paper.

B. Exisiting Hardware Solutions and Remaining Problems

Existing hardware implementations have achieved high throughputs automata processing but they have to rely on high capacity memories. Micron's automata processor (AP) [6] is a direct implementation of NFAs, adapting the spatial architecture in SDRAMs. AP consists of lots of state transition elements (STEs), which work concurrently and provide high parallelism. In this way, AP meets the high processing complexity required by NFA model. However, one-hot encoding is a must for AP, which results in low storage efficiency. Despite that AP can store lots of patterns exploiting the high capacity provided by SDRAMs, it cost large chip areas and much energy. Also, AP requires complex hardware interconnections among STEs, which also occupy much chip area. Although [7] implements AP into caches and [8] optimizes the routing architecture of AP, they also have the same problems with AP. Besides, UAP [10] supports another class of NFA implementations, which conducts all the operations serially. Thus it suffers from lower processing speed. In contrast, IBM's RegX [9] is a DFA based hardware accelerator, achieving relatively low storage efficiency. Thus, it relies on external DRAMs to stores the whole automata model. Besides, it utilizes hierarchical caches for faster memory access, which introduces complex hardware to implement and manage the caches.

III. Opportunities and Challenges

We consider ADFA model [11] as a promising choice to achieve lightweight hardware. In this section, we first present the effectiveness of its compression algorithms. Then we analyze the root cause of the performance losses in ADFAs.

A. Effective Edge Compression Algorithms in ADFA model

An automaton can be modeled as a labeled graph, where each node means a state and each labeled edge means a transition between two states. Fig. 1 shows the DFA model and the ADFA model accepting the same patterns. Significant reduction in edges is achieved in ADFA model, where ”delta storage” is enabled. As the red and blue lines in Fig. 1 (a) show, S3 shares the same destination with S1 when the input is ”a”, ”b” and ”d”. In ADFA model, we remove such redundancies for higher storage efficiency. For instance, all the blue edges in Fig. 1 (a) are replaced by a single special edge in Fig. 1 (b). Such special edges are called ”default edges” and indicated by dash lines in Fig. 1 (b).

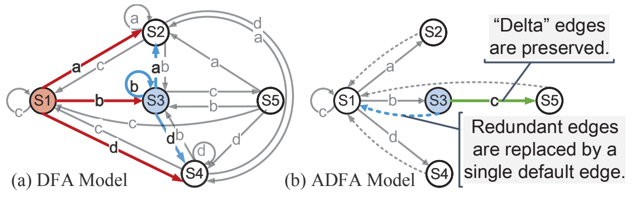

S1
S2
S3
S4
S5
a
b
c
d
(a) DFA Model
(b) ADFA Model
"Delta" edges are preserved.
Redundant edges are replaced by a single default edge.

Fig. 1. The DFA and its equivalent ADFA, accepting a+, b+c and c*d+.

B. Performance Losses Due to Additional Memory Accesses

ADFA model also introduces more complex operations in runtime, which result in performance losses. In DFA model, only one memory access is needed to process a character. For each state, one transition table is implemented storing all its outgoing edges. Each time an input is received, an entry is chosen from the transition table of the current state to determine the next state. However, the situation is different for ADFA model. As described in III-A, ”delta storage” is enabled in ADFA model and redundant edges are removed for each state. **It imposes a new requirement of conducting ”fall-backs” in runtime.** Once the current state needs to know the destination of one removed edge, it has to obtain such information from its default state. Thus, the current state will switch to its default state and the same input will be processed under this new state. For instance, assuming that S3 is the current state and the input is ”b”. No valid edge labeled

”b” will be found in S3’s transition table since this edge is redundant and removed as shown in 1 (b). As a result, S3 has to fall back to its default state, S1. Then, we try to locate the desired edge in S1’s transition table by the subsequent memory access. More memory accesses might be required if more ”fall-backs” are needed. As a result, additional memory accesses are introduced by such ”fall-backs”, which have slowed down ADFA based processing in existing implementations.

IV. LAP Architecture

We propose LAP, a lightweight automata processor, achieving both high speed and high storage efficiency. In this section, we first describe its instruction set and demonstrate its hardware pipeline. Then, we describe the software-hardware co-design in LAP, which can alleviate the performance losses described in III-B. At last, we describe LAP’s compilation software, which is also necessary for our co-design.

A. Instruction Set and Pipeline Design

a) Instruction Set: Our instructions are adapted from previous lightweight implementations [10] [12] [13]. In total, We implement 7 kinds of instructions in LAP. Each instruction is encoded into 32 bits with four fields including ”signature”, ”target”, ”type” and ”attach”. ”Signature” is used to determine if it is a valid edge for the current state. Besides, ”target” specifies the identifier of the next state. Furthermore, ”type” specifies the type of this instruction, which also specifies the usage of ”attach”. Note that ”attach” can be used to address associated instructions described in IV-B. Table I shows the types and functions of these instructions. Note that ”#Cycle1” and ”#Cycle2” describe how many cycle are needed for each instruction before and after our co-design described in IV-B.

TABLE I
Instruction Set of LAP.

Instruction Type	Function	#Cycles1	#Cycles2
NULL	Doing nothing	1	1
BASIC	Supporting labeled edges	1	1
DEFAULT_OPT1	Supporting ADFA model	2 or 1	1
DEFAULT_OPT2	Supporting ADFA model	2 or 1	1
MAJORITY	Compressing model size	2 or 1	1
EPSILON	Supporting NFA model	2	1
PERSIST	Optimizing NFA model	1	1

b) Key Elements: As described in II-A, the execution of an automaton starts with a single active state $q0$. In our design, each state $q \in Q$ is assigned a unique identifier encoded with 12 bits while each input $\alpha \in \Sigma$ is an 8-bit ASCII character. With the first character input to the automaton, $q0$ switches to one or more active states according to the transition function δ. Since then, each time an input is received, each active state switches to a set of new states independently according to δ. To run an automaton in hardware, we have to record all its active states during the whole process. In LAP, the active states are stored in **Active State Stack (ASS)** and dynamically updated in runtime. To support both ADFAs and NFAs, ASS manages multiple active states and outputs them serially. Thus, the transition procedure for each active state can be conducted in turns. Besides, **Stream Prefetch Unit (SPU)** loads the input data, a stream of characters defined in Σ, from the input buffer. It outputs these characters serially and it will not output a new character until the old one has been consumed by the automaton. As for **Instruction Memory and Auxiliary Memory**, they store LAP’s binary programs consisting of instructions described in IV-A. Their contents, which describe the behavior of the transition function δ, are generated according to the automaton we plan to run. What’s more, **Instruction Decoder** is the controller of LAP. It updates ASS according to the information it gathered and reports the accepted states defined in F when they are activated.

c) Pipelined Data Paths: We propose a four-stage pipeline to implement LAP for high hardware frequency. The simplified structure of LAP is demonstrated in Fig. 2, where data paths are partitioned into four stages. In the first stage, ASS outputs one active state (the current state) while SPU outputs one character (the current input). In the second stage, two physical addresses are generated according to the state and the input. Due to the careful arrangement of our compilers, the address of the instruction to be fetched can be calculated with a simple operation (integer addition as shown in Fig. 2 (a)). In the third stage, two instruction words are fetched simultaneously. Finally, the Instruction Decoder updates ASS and reports the current state if it is an accepted state. Note that the circuit delays for these stages are balanced as we synthesize LAP into FPGAs. What’s more, fine-grained multithreading is applied here to solve the dependencies in the pipeline. Filling this pipeline with four independent contexts makes our pipeline stall-free. Each context has its input stream but all contexts share the same binary programs. Thus, LAP can find a set of patterns among 4 input streams simultaneously.

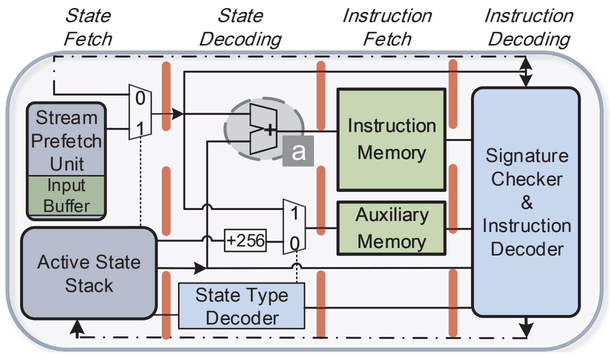


Fig. 2. Pipelined architecture of LAP.

B. Co-design for Faster ADFA based Processing

As described in III-B, additional memory accesses are introduced in ADFAs. Even worse, it is not easy to parallel these accesses due to the dependencies among them. Thus, previous works [10] [12] deal with these memory accesses serially, which significantly limits their processing speed. Fortunately, we observe that the conventional ADFA traversal algorithms

[14] [13] are parallelizable. We propose a **paralleled traversal algorithm (PTA)** for ADFA model, where the time for additional memory accesses is reduced. Based on *PTA*, two novel hardware instructions are implemented in LAP for faster ADFA traverse. Exploiting such software-hardware co-design, two optimizations for two kinds of "fall-backs" are enabled.

1) Falling back to the initial state (Optimization-1): As Fig. 1 (b) shows, the transition diagram of the ADFA shows a tree structure where S1, the initial state of the automaton, is the final destination for the "fall-backs". We have observed that S1's transition table is frequently accessed due to such "fall-backs" in pattern-matching tasks. Thus, we try to hide the overhead accessing the transition table of the initial state.

Fig. 3 (a) shows the traditional way of conducting "fall-backs" to the initial state. An instruction word is fetched in the first cycle as the blue lines show. If this instruction passes the signature check, it will be used to update the current state. Otherwise, another memory access to the transition table of the initial state is required. For example, if the current state is S3 in Fig. 1 (b) and the current input is "b", the automaton will fall back to S1 and find the desired edge from S1's transition table in the second cycle. In brief, the second instruction is fetched optionally depending on the signature of the first instruction, which requires us to perform these memory accesses serially.

Fortunately, this dependency can be broken. We propose a new instruction, "DEFAULT_OPT1", to implement our *PTA*. As shown in Fig. 3 (b), LAP does not trigger the second memory access optionally. Instead, LAP fetches these two instructions in parallel and chooses one of the instructions according to the result of the signature check. In this way, the second memory access can be overlapped with the former one.

2) Falling back to non-initial states (Optimization-2): Situations are different when falling back to non-initial states. Each non-initial state has one default state and has an incomplete transition table where redundant edges are removed. Thus, the instruction fetched from their transition tables might not be a valid one, which makes optimization-1 not applicable.

However, such kind of "fall-backs" can also be optimized. Different from the "fall-backs" to the initial state, the complete information of the non-initial state should be acquired before we fall back to it. Such information including "target", "type" and "attach" is encoded into one instruction word, which is called the associated instruction in this paper. Fig. 3 (c) shows the traditional way of conducting "fall-backs" to the non-initial states, where the associated instructions are fetched optionally in the second cycle. We observe that the associated instructions are usually fetched immediately after the completion of the instructions they are associated with. Thus, we separate the associated instructions into the *auxiliary memory* so they can be fetched from the *auxiliary memory* at the same time when the instructions they associated to are accessing the *instruction memory*. This optimization is enabled by our novel instruction "DEFAULT_OPT2" as shown in Fig. 3 (d). In this way, the overhead of fetching associated instructions is hidden by exploiting the free bandwidth of the *auxiliary memory*.

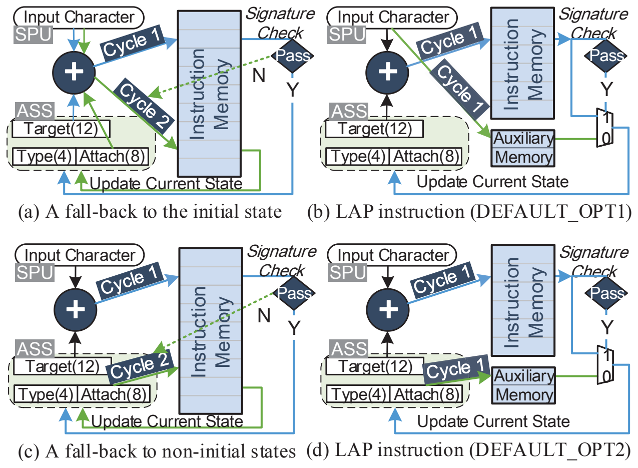

Input Character
SPU
Cycle 1
Cycle 2
Instruction Memory
Signature Check
Pass
N
Y
ASS
Target(12)
Type(4)
Attach(8)
Update Current State
(a) A fall-back to the initial state
Auxiliary Memory
(b) LAP instruction (DEFAULT_OPT1)
(c) A fall-back to non-initial states
(d) LAP instruction (DEFAULT_OPT2)

Fig. 3. Two kinds of "fall-backs" in ADFAs and corresponding optimizations.

C. Compilation Software

Compilation software is developed to run pattern matching applications on LAP. We adapt the open-source C++ software [15] to transform patterns (regular expressions) into a specific automata model. Then, the LAP programs are generated using the Python scripts we developed based on that automata model.

1) Transforming patterns into equivalent automata: In this paper, regular expressions can be transformed into ADFAs or NFAs. Fig. 4 gives an example of such transformations and visualizes the target ADFA model hierarchically. Labeled edges indicate the transitions that are triggered when the signature check passes while the default edges indicate the "fall-back" paths. We set the parameter *"Depth Bound" (DB)* to 2 when generating the ADFA model. Note that DB means the max depth allowed in Fig. 4 hierarchy-2. In this way, we ensure that each state at most experiences two "fall-backs" before reaching the initial state. More importantly, it keeps the associated instructions in a small number so that they only cost neglectable storage overhead for the *auxiliary memory*.

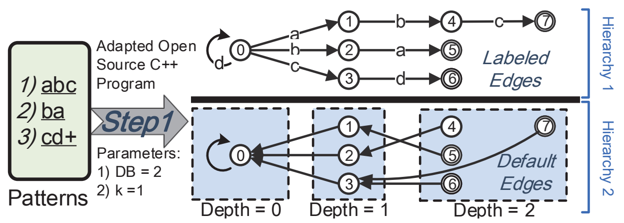


Fig. 4. Hierarchical visualization of the ADFA, accepting abc, ba and cd+.

2) The split EffCLiP representation of automata models: All transition tables are stored in an overlapped fashion exploiting Efficient Coupled-Linear Packing algorithm (EffCLiP [14]), which contributes to highly compact storage. In order to support optimizations described in IV-B, we propose *split EffCLiP representation*, storing automata models compactly into two separated memories. As shown in Fig. 5, the labeled edges in Fig. 4 are encoded into linear memory space using EffCLiP algorithms. Besides, the transition table of the initial state, which is marked as "initial table", is stored in the *auxiliary memory* to support optimization-1. Besides, the associated instructions, which are marked as "associated edges", are also stored in the *auxiliary memory* to support optimization-2.

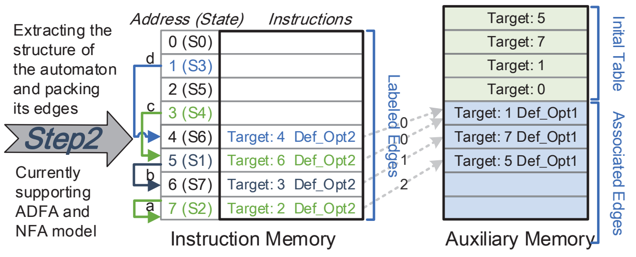


Fig. 5. Split EffCLiP representation of the ADFA model.

V. Experiments and Evaluations

A. Experimental Setup

LAP is implemented with Verilog and prototyped into Xilinx Artix-7 FPGA at 263 MHz frequency using Vivado Design Suite 2017. The *Instruction memorie* (16KB) and the *auxiliary memories* (0.6KB) are implemented with BRAMs. Each LAP core utilizes 724 LUTs (7%), 2199 FFs (11%) and 5 BRAMs (20%) with 0.12W overall dynamic power.

Two classes of standard workloads are used for evaluations. PowerEN simple is a series of workloads, which are used in IBM's PowerEN processor to evaluate its regular expression accelerator [9]. Besides, Exact-Match (EM) and range1 are synthetic workloads generated by the tool described in [15]. We evaluate LAP's processing speed with line rate and evaluate its storage efficiency with pattern density. The line rate (symbol/cycle) indicates the average number of characters each LAP core can process per cycle, while the pattern density (#patterns/KB) is measured by the number of patterns divided by the memory size they occupy. We compile these workloads into LAP programs and simulate their executions on LAP using the Vivado Simulator, a cycle-accurate simulator for Verilog. The number of cycles used to process each workload is obtained from the Vivado Simulator while the pattern densities are calculated by our compilers.

B. Experimental Results

a) Overall Line Rates: Fig. 6 shows LAP's overall line rates across a variety of workloads. Although our work mainly aims to achieve faster ADFA-based processing, performance for NFA model is also evaluated for comprehensive evaluations. Besides, the performance of UAP [10] is included here. Since LAP and UAP use similar instruction sets, direct comparisons between them can show the effectiveness of our software-hardware co-design fairly. As Fig. 6 shows, LAP achieves comparable line rates with UAP from 0.49 to 0.66 symbols/cycle using NFA model. As for using ADFA model, LAP achieves line rates from 0.91 to 0.97 across different workloads. Meanwhile, "ADFA is a more complicated model, so each transition requires several sequential references, producing a lower line rate, ranging from 0.47 to 0.75." (UAP [10], 2015) In summary, LAP shows significant performance improvements ranging from 32% to 91% in ADFA based processing compared with UAP. As a result, the prototype FPGA system equipped with five LAP cores can achieve 9.5 Gbps processing throughput using ADFA model.

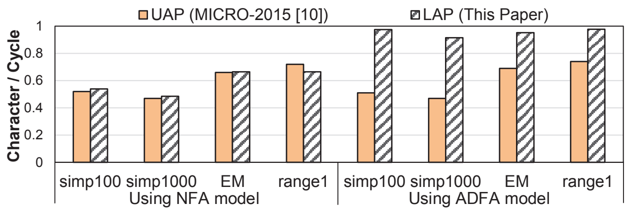


Fig. 6. Overall line rates of LAP and UAP.

b) Detailed Analysis of the Line Rates: As described in III-B, additional memory accesses in ADFA model result in poor processing speed. As a result, UAP [10] achieves relatively low processing speed using ADFA model. In contrast, RegX [9] has achieved the ideal line rate of DFA based processing. Therefore, we compare LAP's performance with these two typical DFA based implementations in Fig. 7, where UAP serves as a baseline and RegX serves as the ceiling. Also, we evaluate the performance of LAP with or without optimizations described in IV-B to identify the sources of the performance improvements. Fig. 7 shows that LAP with no optimization achieves similar performance with UAP. However, LAP equipped with optimization-1 achieves significant improvements. It infers that the main overhead, which limits the speed of ADFA based processing, is caused by the additional accesses to the initial state due to the "fall-backs". Furthermore, optimization-2 further improves the performance of LAP utilizing the free bandwidth of the auxiliary memory. Finally, LAP achieves the state of the art performance.

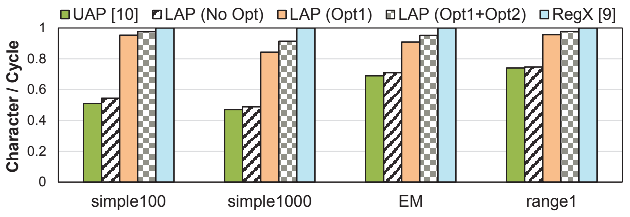
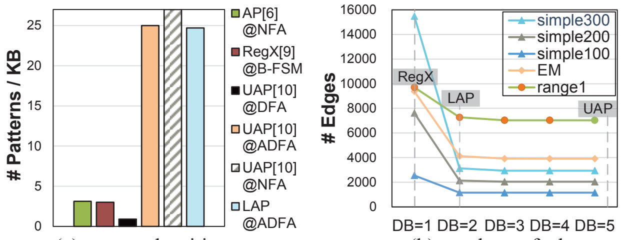


Fig. 7. Detailed performance evaluations of LAP.

c) Pattern Densities: Fig. 8 (a) shows the pattern densities achieved by LAP and existing hardware solutions over simple400 workload [9]. AP represents a class of implementations based on Micron's Automata Processor [6], where one-hot encoding is used and similar pattern densities are achieved. RegX [9] achieves similar pattern density with AP utilizing B-FSM model, which is an augmented DFA model. What's more, UAP [10] supports different automata models. It achieves bad pattern density using DFA model. However, it achieves the state of the art pattern densities using NFA model or ADFA model despite that its processing speed is unsatisfactory. Most importantly, LAP also achieves the state of the art pattern density, which is 8× higher than RegX and AP.

d) Discussions about the Pattern Densities: In general, NFA models have fewer states and fewer edges than the equivalent DFA models. As a result, UAP [10] achieves the highest pattern density using NFA model utilizing efficient packing algorithms. However, AP-like implementations [6] [8] [7] achieve relatively low pattern density based on NFA model since one-hot encoding is used. Besides, UAP achieves bad pattern densities using DFA model in Fig. 8 since DFAs contain tremendous redundant edges. In contrast, "delta storage" is enabled in RegX [9], UAP [10] and LAP, utilizing default edges. Thus, they achieve better pattern densities. It's worth noting that the key difference among these implementations is how many fall-backs are allowed at most before reaching the initial state, which is determined by the parameter *"Depth Bound" (DB)*. As Fig. 8 (b) shows, the numbers of edges keep reducing as the value of DB increases. RegX only supports single depth default edges, which means DB equals 1 in RegX. Besides, we set DB to 2 in LAP as described in IV-C. We can see a sharp decrease in Fig. 8 (b) as DB goes from 1 to 2. As a result, LAP achieves much higher pattern density than RegX. It is worth noting that a more efficient packing algorithm and encoding methods also contribute to our higher pattern density. What's more, DB in UAP is unlimited. As a result, UAP stores fewer edges compared with LAP, resulting in slightly higher pattern density. However, it is a worthwhile trade-off since much higher speed is obtained by LAP.



(a) pattern densities (b) numbers of edges

Fig. 8. Pattern densities and the numbers of edges.

VI. Conclusion

In this paper, we propose LAP, a lightweight automata processor. Extremely high storage efficiency is achieved in LAP, leveraging algorithmic innovations. Besides, faster processing speed is achieved exploiting the software-hardware co-design over the instructions, the architecture and the compilers. LAP is prototyped into Xilinx Artix-7 FPGA at 263 MHz frequency based on our four-stage pipeline. Evaluations show that LAP's storage efficiency exceeds IBM's RegX and Micron's AP by 8×. Besides, LAP achieves significant improvements in processing speed ranging from 32% to 91% compared with previous lightweight implementations.

VII. Acknowledgement

This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFA0700900, 2017YFA0700903), National Science Foundation of China (No. 61976200), Jiangsu Provincial Natural Science Foundation (No. BK20181193), Youth Innovation Promotion Association CAS (No. 2017497), and the 2020 Youth Innovation Fund of University of Science and Technology of China. Lei Gong is the corresponding author.

References

[1] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*. USA: Addison-Wesley Longman, 1969.

[2] K. Wang, Y. Qi, J. J. Fox, M. R. Stan, and K. Skadron, "Association rule mining with the micron automata processor," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pp. 689–699, 2015.

[3] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2006, San Jose, California, USA, December 3-5, 2006*, pp. 93–102, 2006.

[4] I. Roy and S. Aluru, "Finding motifs in biological sequences using the micron automata processor," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pp. 415–424, 2014.

[5] M. H. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda, "Using the automata processor for fast pattern recognition in high energy physics experiments—a proof of concept," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 832, pp. 219–230, 2016.

[6] 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.*, vol. 25, no. 12, pp. 3088–3098, 2014.

[7] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. T. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18*, pp. 259–272, 2017.

[8] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eap: A scalable and efficient in-memory accelerator for automata processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pp. 87–99, 2019.

[9] J. van Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a programmable wire-speed regular-expression matching accelerator," in *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pp. 461–472, 2012.

[10] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9*, pp. 533–545, 2015.

[11] M. Becchi and P. Crowley, "A-DFA: A time- and space-efficient DFA compression algorithm for fast regular expression evaluation," *TACO*, vol. 10, no. 1, pp. 4:1–4:26, 2013.

[12] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "UDP: a programmable accelerator for extract-transform-load workloads and more," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pp. 55–68, 2017.

[13] Y. Fang and A. A. Chien, "Udp system interface and lane isa definition," tech. rep., https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2017-05, 2017.

[14] Y. Fang, A. Lehane, and A. A. Chien, "Effclip: Efficient coupled-linear packing for finite automata," tech. rep., University of Chicago Technical Report, TR-2015-05, 2015.

[15] M. Becchi, M. A. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pp. 79–89, 2008.