Accelerating Complex Pattern Recognition Processing with In-Memory Accelerator Architectures

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Elaheh Sadredini

May 2019

Abstract

Newly available memory-centric architectures to accelerate finite automata-based pattern recognition have motivated the use of automata processing in new applications, such as bioinformatics, machine learning, and natural language processing. However, the existing automata processing solutions, including von Neumann and these new in-memory accelerators, are neither scalable nor efficient. Moreover, existing in-memory automata processing accelerators are unable to process more complex pattern structures such as tree and graph-shaped patterns.

This dissertation outlines five new contributions to improve the effectiveness of automata processing. This includes accelerating two applications using automata processing by developing novel algorithms and mapping them to in-memory automata processing accelerators: 1) frequent subtree mining, and 2) rule-based part-of-speech tagging in natural language processing.

This dissertation then presents three novel architecture studies. Based on the preliminary results from the above applications and their natural structures, and also by investigating the interconnect and scalability inefficiency in the existing in-memory automata processing architectures, 3) we propose a reconfigurable high-speed, high-density, and low-power automata processing architecture with a compact, low-overhead, and yet flexible interconnect design. To evaluate our proposed architecture, we develop a cycle-accurate automata processing simulator, flexible enough to be adopted by other researchers for automata processing research.

Motivated by our study on pattern matching in tree-shaped structures, 4) we propose a general-purpose, scalable, and reconfigurable memory-centric architecture for processing complex patterns, such as tree-like data. We take inspiration from previous automata processing architectures, but support the richer deterministic pushdown automata computational model.

Finally, we observed that fixed 8-bit symbol processing, derived from ASCII processing, restricts throughput and area efficiency of existing automata accelerators, causes resource underutilization, and limits the general adoption of applications with very large symbol-set size. To address these issues, 5) we present FlexAmata, a compiler solution that efficiently re-shapes an automaton structure to provide application compatibility with existing and future memory-centric automata processing architectures, and hardware compatibility for

big-data domain applications. The former allows execution efficiency and feasibility of applications with very large or very small symbols-sets on memory-centric automata accelerators. Inspired by this exploration, we present a comprehensive design space exploration for different bitwidth automata processing on FPGAs and in-memory approaches. This study provides several insights to design future automata processing accelerators more efficiently.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Elaheh Sadredini

This dissertation has been read and approved by the Examining Committee:

Kevin Skadron , Adviser

Ashish Venkat , Committee Chair

Worthy N. Martin, Committee co-Chair

Mircea Stan

Hongning Wang

Accepted for the School of Engineering and Applied Science:

Craig Benson, Dean, School of Engineering and Applied Science

May 2019

DEDICATION

I gratefully dedicate this dissertation to my husband, Reza. I also dedicate it to the memory of my father and grandparents.

Acknowledgements

I am grateful to numerous people who have contributed towards shaping this dissertation. I could not have reached the finish line without the influence, advice, and support of many colleagues, friends, and family.

First and foremost, I would like to express my special appreciation to my advisor, Prof. Kevin Skadron, who has been a tremendous mentor for me and has shaped my life in many profound ways. Kevin is absolutely brilliant, a clear thinker with the uncanny ability to discover and communicate crisply. His heart-warming support has given me great confidence as a researcher, and at the same time made me realize that I am only a beginner in this exciting profession. He always trusted in me and gave me tremendous freedom in pursuing my ideas, and at the same time continuing to contribute valuable feedback and encouragement. I will forever treasure the advice I have received from him over the last five years, and I am sure I have yet to learn so much more from him.

I also wish to thank the members of my Ph.D. committee; Prof. Mircea Stan, Prof. Worthy Martin, Prof. Hongning Wang, and Prof. Ashish Venkat for their many insightful discussions and suggestions on my research. I also want to express my gratitude to Prof. Sally McKee, who inspired me and helped to join UVa. Your support and kindness will always be remembered.

I am grateful for having so many friends and collaborators both inside and outside of the Computer Science department. To Jack Wadden, Tommy Tracy, Marzieh Lenjani, Viabhav Verma, Sergui Mosanu, Chunkun Bo, Deyuan Guo, Ke Wang, and Kevin Angstadt; thank you for your help and support.

Last but not least, I am deeply thankful to my family, especially my mother, for their unconditional love, support, and sacrifices. It has been extremely painful to be far from them all these years, and not be able to come back for a visit. I could not bear this pain without the support of my best friend, soul-mate, and husband, Reza. Reza has been a true and great supporter and has unconditionally loved me during my good and bad times. These past several years have not been an easy ride. I truly thank Reza for sticking by my side and always making me laugh. There are no words to convey how much I love you.

Chapter 3, in full, is a reprint of the material as it appears in proceeding of KDD 2018. Sadredini, Elaheh; Guo, Deyuan; Bo, Chunkun; Rahimi, Reza; Skadron, Kevin; Wang, Hongning, A scalable solution for

rule-based part-of-speech tagging on novel hardware accelerators, Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD), 2018. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in proceeding of ICS 2017. Sadredini, Elaheh; Rahimi, Reza; Wang, Ke; Skadron, Kevin, Frequent Subtree Mining on the Automata Processor: Challenges and Opportunities, International Conference on Supercomputing (ICS), 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in proceeding of MICRO 2018. Angstadt, Kevin; Subramaniyan, Arun; Sadredini, Elaheh; and Rahimi, Reza; Skadron, Kevin; Weimer, Westly; Das, Reetu, ASPEN: A Scalable In-SRAM Architecture for Pushdown Automata, 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'51), 2018. This work is done in collaboration with University of Michigan. We came up with a very similar idea independently and then, we teamed up and and wrote the paper. The detailed hardware design was jointly developed. The presented work in this proposal covers the dissertation author contribution.

Chapter 6, in full, is a reprint of the material as it appears in Computer Architecture Letters, 2019. The extended version of this work is under review. Sadredini, Elaheh; Rahimi, Reza; Verma, Vaibhav; Stan, Mircea; Skadron, Kevin, A Scalable and Efficient in-Memory Interconnect Architecture for Automata Processing, Computer Architecture Letters, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 7, in full, is currently under review. Sadredini, Elaheh; Rahimi, Reza; Lenjani, Marzieh; Stan, Mircea; Skadron, Kevin, FlexAmata: A Flexible Automata Processing Engine. The dissertation author was the primary investigator and author of this paper.

علم چون بر دل زند یاری شود　　علم چون بر تن زند باری شود

لیک چون این بار را نیکو کشی　　بار بر گیرند و بخشندت خوشی

هین مکش بهر هوا آن بار علم　　تا ببینی در درون انبار علم

تا که بر رهوار علم آیی سوار　　بعد از آن افتد ترا از دوش بار

مولانا

When knowledge strikes on the heart, it becomes a helper; when knowledge strikes on the body, it becomes a burden.

But when you carry this burden well, the burden will be removed and you will be given (spiritual) joy.

Beware! Do not carry that burden of knowledge for the sake of selfish desire, so that you may mortify the barn of knowledge within you.

So that you may mount the smooth-paced steed of knowledge, afterwards the burden may fall from your shoulder.

Rumi, 13th century

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The demand for high-speed mining and analysis of unstructured data is growing fast, especially in technical and business data analytics applications. Fast information extraction from textual data can be critical for business decision making. One prominent way of filtering and extracting information from collected data is to use regular expressions. They are used in identifying complex patterns and variants of base patterns, potentially the most time-consuming task in many big-data applications.

Finite automata (a form of finite state machines) are an efficient computational model for widely used pattern recognition languages such as regular expressions, with important applications in network security [3, 4], log analysis [5], and many more. There are many other applications in domains such as data-mining [6, 7, 8, 9], bioinformatics [10, 11], machine learning [12, 13], natural language processing [14, 15], big data analytics [16], and even particle physics [17], that shown to greatly benefit from accelerated automata processing. The automata structure in these applications differ significantly in static structure and dynamic behavior from existing regular expression benchmarks [2, 18]. Moreover, there is a large body of research on using finite automata in formal verification such as satisfiability and model-checking problems in temporal logics [19, 20, 21] with potentials to benefit from high-performance automata processing especially for real-time needs.

Pattern recognition for regular languages can be computed by either deterministic finite automata (DFA) or non-deterministic finite automata (NFA), which are equivalent in computational power. On CPUs, NFAs and DFAs are represented by tables indicating each state's successor state(s) upon a rule match. DFAs are often the basis for implementing automata on CPUs, because they have predictable memory bandwidth requirements; while an NFA may have many active states and may require many state lookups to process a single input symbol (potentially leading to a very large memory bandwidth requirement), a DFA requires

just one. On the other hand, DFA tables are often too large to fit in the processor caches, because DFAs often suffer an exponential increase in the number of states relative to NFAs.

Graphics processing units (GPUs) provide a large number of parallel resources, which can help in hiding the DRAM access latency. However, highly-random access patterns in automata processing exhibit poor memory locality and increase branch divergence and need for synchronization [2]. Despite its foundations importance, the efficient implementation of finite automata processing remains a challenging open research problem and the subject of extensive research.

Researchers are increasingly exploiting memory-centric hardware accelerators to meet demanding real-time requirements as performance growth in conventional processors is slowing down. The growing demand for accelerated automata processing has motivated many efforts in designing regular expression and general automata accelerators on ASIC [22, 5], FPGAs [23, 24, 25, 26, 27], and processing-in-memory (PIM) designs [28, 29]. Spatial memory-centric accelerators, such as FPGAs and in-memory solutions, provide a reconfigurable substrate to lay out the rules in hardware by placing-and-routing automata states and connections onto a pool of hardware units in logic- or memory-based fabrics. This allows a large number of automata to be executed in parallel, up to the hardware capacity, in contrast to von Neumann architectures such as CPUs that must handle one rule at a time in each core.

To more efficiently process complex patterns, the automata accelerator should provide (1) a rich computational model to account for complex structures such as tree and graph-shaped patterns, (2) a low-overhead and yet efficient interconnect architecture to support complicated interconnect structures of real-world automata applications, and (3) a flexible bitwidth processing for better hardware utilization and compatibility.

Processing of tree-structured or recursively-nested data is intrinsic to many computational applications, such as programming languages (such as C, C++, ...) and natural languages (such as English) [30]. However, tree structure processing cannot be accomplished using finite automata. The tree structure is more complex than a sequence and cannot be described with regular languages [31]. It means that we need a richer computational model to count for the possible branches when processing a tree structure. The existing automata accelerators are incapable of processing tree-structured data.

Moreover, the interconnect design of existing automata processing accelerators are either incapable of efficient place-and-route of a highly-connected automaton or over-provision hardware resources for interconnect, sometimes at the expense of resources for state-matching [28, 29, 24]. However, real-world benchmarks are quite large in terms of the number of states, too big to fit in a single hardware unit, and thus usually need multiple rounds of reconfiguration and re-processing of the data. This incurs significant performance penalties and makes state-matching resources a scarce resource.

In addition, because regular expressions have most commonly been used for text and packet processing, existing automata accelerators are designed based on a fixed 8-bit (ASCII) symbol processing scheme, similar to software solutions [32, 33]. This means that the automata structure is modeled based on 8-bit symbols and an 8-bit input is processed in each cycle. However, we observed that the fixed-size symbol processing design can be a source of area and throughput inefficiencies and it also limits the general adoption of applications for automata processing.

1.1 Contributions

In this dissertation, **we hypothesize that an efficient interconnect architecture, a more computationally powerful automata processing design, and the right bitwidth processing granularity can unleash in-memory processing benefit for more complex pattern recognition tasks.** To evaluate this hypothesis, this dissertation proposes to unlock several previously unexplored problems in memory-centric automata accelerators by investigating the strength and limits of existing memory-centric automata processing accelerators, and proposing two new and complex use-cases and application domains in (1) data mining and (2) natural language processing for automata-based acceleration, (3) improving existing in-memory automata processing accelerators and investigate new architectures for automata processing, (4) investigating a more-powerful automata processing model and architecture that can process more complex patterns, and (5) introducing a HW/SW solution which allows execution efficiency and feasibility of applications with very large or very small symbols-sets on memory-centric automata accelerators, and also offers insights for next-generation automata processing accelerators.

1.1.1 Accelerating Tagging in Natural Language Processing on Automata Accelerators

In this chapter, using part-of-speech (POS) tagging as a case study, we show that automata processing hardware accelerators can make rule-based techniques orders of magnitude faster than statistical- or machine learning-based taggers. This allows rule-based approaches to employ more rules and achieve accuracy competitive with statistical techniques. These observations motivate a re-evaluation of rule-based approaches in natural language processing (NLP).

We study the relationship between the rule-set size and the accuracy of POS taggers, and observe that more complex rules (from more diverse template rule-sets) and larger rule sets lead to accuracy almost as good as statistical/ML-based techniques, especially with a larger training corpus. With hardware acceleration, this

sets up a new tradeoff for designers of NLP applications. We utilize spatial automata processing architectures (the Automata Processor (AP) [28] and the FPGA [23]) by transforming the rules to regular expressions. This approach is scalable in the number and complexity of rules. Increasing the number of rules up to several thousand has no overhead on the AP and a minimal overhead on the FPGA, because all the rules are laid out in space across the chip and executed in parallel.

We then compare our solution on the AP and FPGA with rule-based, statistical-based, and machine-learning-based approaches. Results show that we can achieve up to 2,600× and 1,914× speedups on the AP and on the FPGA respectively over CPU-based Brill tagging in the rule-matching stage, up to 58× speedup over the Perceptron POS tagger (CPU solution) in the total testing time, and up to 253× speedup over the LSTM tagger (GPU solution) in total testing time at the expense of approximately 1% accuracy in a large corpus.

1.1.2 Accelerating Tree Processing on Automata Accelerators

Frequency counting of complex patterns such as subtrees is more challenging than for simple itemsets and sequences, as the number of possible candidate patterns in a tree is much higher than one-dimensional data structures, with dramatically higher processing times. In this chapter, we first study difficulties of directly implementing the frequent subtree mining (FTM) problem on the AP platform. Then, we propose a multi-stage pruning framework to greatly reduce the search space of embedded FTM on the AP. This provides a scalable solution in terms of both memory and execution time on large databases and lower support thresholds.

Our solution on the AP achieves up to 353× speedup at the cost of a small reduction in accuracy, on four real-world and synthetic datasets, when compared with PatternMatcher, a practical and exact CPU solution. To provide a fully accurate and still scalable solution, we propose a hybrid method to combine out method with a CPU exact-matching approach, and achieve up to 262× speedup over PatternMatcher on a challenging database. We also develop a GPU algorithm for FTM, but show that the AP also outperforms this. The results on a synthetic database show the AP advantage grows further with larger datasets.

1.1.3 In-Memory Pushdown Automata Accelerator

The existing memory-centric automata processing engines, such as the AP and Cache Automata (CA) [29], do not support a computational model rich enough for tasks such as XML parsing or subtree mining. Inspired by our study on pattern matching in tree-shaped structures, we propose a general-purpose, scalable, and reconfigurable in-SRAM architecture for processing of tree-like data [34]. We take inspiration from previous

automata processing architectures, such as Cache Automata and AP, but support the richer deterministic pushdown automata (DPDA) computational model [34]. We propose a custom datapath capable of performing the state matching, stack manipulation, and transition routing operations of pushdown automata, all efficiently stored and computed in memory arrays.

We evaluate the run-time and energy consumption of architecture on subtree inclusion problem and we compare it to the state-of-the-art CPU and GPU solutions. Our solution shows 67.2× and 6× end-to-end performance improvement over optimized single-threaded CPU and GPU implementations of subtree mining. Moreover, it achieves 3070× and 6279× improvements on average in total energy when compared to CPU- and GPU-based implementations, respectively.

1.1.4 Efficient In-Memory Interconnect for Automata Processing

Motivated by preliminary results from the investigated applications and their natural structures, and also by investigating the interconnect and scalability inefficiency in the existing in-memory automata processing architectures, this chapter presents a *reduced-crossbar (RCB)* design, a low-overhead and yet flexible interconnect architecture that efficiently implements state-transition. RCB design is inspired by intrinsic properties of real-world automata connectivity patterns. RCB requires at least 7× fewer switches compared to the FCB design used in previous interconnect designs [29]. This in turn reduces the wire length, which results in shorter latency and lower power consumption. In addition, the area efficiency of RCB provides an opportunity to design a denser state matching resources, which can accommodate more states and results in fewer rounds of re-configuration and re-processing of data.

We then present eAP (embedded Automata Processor), a high-throughput and scalable in-memory automata processing accelerator. Performance benefits of eAP are achieved by (1) exploiting subarray-level parallelism in memory, (2) designing an optimal pipeline for state matching and state transition, (3) our compact interconnect architecture, and (4) choice of memory technology.

To efficiently allow many-to-many transitions in an automaton, the underlying memory technology for eAP should be able to support logical OR functionality within memory rows in a subarray. This requires memory cells (a) to provide non-destructive read, and (b) to drive output to a "stable" state (logical OR in this case) when multiple bitcells drive a common bitline. 8T SRAM cells [35] and gain-cell embedded DRAM (GC-eDRAM) [36, 37] are examples of feasible memory technologies to implement eAP. In this paper, we evaluate eAP on both 8T and 2T1D (2 transistors 1 diode) memory cells. The 2T1D cell is a GC-eDRAM designed and fabricated by [38]. 2T1D uses substantially fewer transistors than an 8T SRAM cell and thus

incurs lower area overhead, which results in higher state density and therefore better throughput (due to the reduced rounds of reconfiguration and re-streaming of input).

1.1.5 A Flexible Automata Processing Scheme

One advantage of using memory-centric solutions is that the processing bitwidth can be customized for the application need. In this chapter aims to answer the following questions. What are the necessary hardware/software modifications to efficiently support very large or very small bitwidth sizes on memory-centric architectures? What is the best bitwidth size for automata processing on different platforms (including CPU/GPUs, FPGAs, and in-memory architectures)? How to design next-generation automata accelerators with higher throughput?

To answer the above questions, we propose FlexAmata, a compiler solution to provide *application compatibility* with existing and future memory-centric automata processing architectures, and *hardware compatibility* with the existing automata designs. FlexAmata transforms the automata shapes to support different bitwidth processing and map them to in-memory architectures and FPGAs. It offers full hardware utilization for the application with small symbol-set size, provides a feasible and low overhead solution for the applications with very large number of symbols on the existing automata accelerators, and maintains application compatibility with the future automata hardware accelerators.

Inspired by this exploration, we propose area efficient and high-throughput in-memory architectures and FPGA automata processing engine. We present a design space exploration for various bitwidth automata processing on our proposed FPGAs and in-memory solutions. Our exploration introduces hints on how to design an automaton for more efficient hardware mapping and insights for next-generation automata processing accelerators.

To evaluate FlexAmata, we develop an open-source toolkit, called APSim, for automata simulation, minimization, transformation, and performance modeling on memory-centric architectures. FlexAmata also generates HDL (Hardware Description Language) to be run on the FPGAs.

1.1.6 Summary

We propose a compact, but flexible interconnect design that enables efficient place-and-route of a complex and highly connected automaton and provides full utilization of state matching resources, which results in higher throughput and lower cost. We then present efficient in-memory hardware support for pushdown automata. This enables processing thousand of complex tree-shaped patterns in parallel with high-throughput input processing. We then introduce a compiler solution that can efficiently re-shape an automaton to a more

complex or a more simple structure. This provides a feasible solution for applications with huge symbol-set size. These set of works provide solutions to process complex patterns more efficiently, which confirms our hypothesis.

1.2 Overview of Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2: Background introduces automata processing, discusses automata-related theory, prior approaches and architectures for automata processing.

Chapter 3: Accelerating Rule-Based Part-of-Speech Tagging presents a general solution to accelerate rule-based methods in natural language processing by exploiting the highly-parallel regular-expression-matching ability of the Automata Processor and FPGAs.

Chapter 4: Accelerating Frequent Subtree Mining on the Automata Processor presents an approximate but fast solution for processing tree mining on in-memory automata accelerators.

Chapter 5: In-SRAM Accelerator for Pushdown Automata Processing presents a general-purpose, scalable, and reconfigurable in-memory solution for high-performance processing of tree-shaped data structures.

Chapter 6: Scalable and Efficient In-Memory Accelerator for Automata Processing presents a novel and compact in-memory interconnect architecture for automata processing and map it to proper memory technologies.

Chapter 7: Flexible Automata Processing presents a compiler solution to provide application compatibility with existing and future memory-centric automata processing architectures, and hardware compatibility with the existing automata designs.

Chapter 8: Conclusions summarizes the dissertation and discusses the implications of this work and potential future directions of research.

Chapter 2

Background

2.1 Finite Automata

Finite automaton (FA) is a finite state machine (FSM) that accepts or rejects strings of symbols.

A FA is a mathematical model of computing and is represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols, Δ is a transition function, q_0 are initial states, and F is a set of final or accepting states. An automaton has one or more start states that initiate computation, and one or more accept states that report a match. The transition function determines the next states using the current active states and the input symbol just read. If an input symbol causes the automata to enter into an accept state, the current position of the input symbol is reported.

A deterministic finite automaton (DFA) allows only one transition per input symbol. A non-deterministic finite automaton (NFA) has the ability to be in several states at once, meaning that transitions from a state on an input symbol can be to any set of states. DFAs and NFAs have equal computational power and can be converted to each other. However, a DFA can have exponentially more states that an equivalent NFA, which greatly increases the memory footprint. On the other hand, an NFA can have many parallel transitions, which is bounded by the limited memory bandwidth in von-neumann architectures.

2.2 Deterministic Pushdown Automata

Pushdown automata (PDAs) extend basic finite automata by including a stack memory structure. A PDA is represented by a 6-tuple, $(Q, \Sigma, \Gamma, \delta, S, F)$, where Γ is the finite alphabet of the stack, which need *not* be the same as the input symbol alphabet. The transition function, δ, is extended to consider stack operations. The transition function for a PDA considers the current state, the input symbol, and the top of the stack and

returns a new state along with a stack operation (one of: push a specified symbol, pop the top of the stack, or no operation).

Deterministic pushdown automata (DPDAs) limits the transition function to only allow a single transition for any valid configuration of the DPDA and an input symbol. This restriction prevents stack divergence, a property we leverage for efficient implementation in hardware. Some transitions perform stack operations without considering the next input symbol, and are refer as *epsilon- or ε-transitions*. To maintain the determinism, all ε-transitions take place before transitions considering the next input symbol.

Unlike basic finite automata, where non-deterministic and deterministic machines have the same representative power (any NFA has an equivalent DFA and vice versa), DPDAs are strictly *weaker* than PDAs [39].

2.3 Automata Processing on von Neumann Architectures

On CPUs, NFAs and DFAs are represented by tables indicating each state's successor state(s) upon a rule match. DFAs are often the basis for implementing automata on CPUs because they have predictable memory bandwidth requirements; while an NFA may have many active states and may require many state lookups to process a single input symbol (potentially leading to a very large memory bandwidth requirement), a DFA requires just one. On the other hand, DFA tables are often too large to fit in the processor caches, because DFAs often suffer an exponential increase in the number of states relative to NFAs.

There are several efforts for high-speed regex processing on the CPUs and GPUs [33, 40, 32]. iNFAnt [32] is a parallel engine for regular expressions on GPUs with the support for multi-striding (processing multiple input bytes in each step). Graphics processing units (GPUs) provide a large number of parallel resources, which can help in hiding the DRAM access latency. However, highly-random access patterns in automata processing exhibit poor memory locality and increase branch divergence and need for synchronization [2].

Generally, automata processing on von Neumann architectures exhibits highly irregular memory access patterns with poor temporal and spatial locality, which often leads to poor cache and memory behavior [2]. Therefore, even high-throughput off-the-shelf von Neumann architectures struggle to meet today's big-data and streaming line-rate pattern processing requirements.

2.4 Automata Processing on Memory-Centric Architectures

To cope with the memory wall problem in conventional von Neumann architectures, in-memory automata processing hardware accelerators have been proposed and have shown several orders of magnitude speedup

compared to CPUs and GPUs [41].

The Automata Processor (AP) [28] and Cache Automata (CA) [29] are two reconfigurable in-memory solutions, both directly implementing NFAs in memory. They can place-and-route automata states in a reconfigurable fabric, eliminating the need to access memory in von Neumann architectures. They also exploit the inherent bit-level parallelism of memory to support many parallel transitions in one cycle. The AP provides a DRAM-based dedicated automata processing chip, and Cache Automata proposes an on-chip solution by re-purposing a portion of the last-level cache for automata processing.

2.4.1 Automata Processor

Micron's Automata Processor (AP) [28] is an in-memory, non-von Neumann processor architecture that computes non-deterministic finite state automata (NFAs) natively in hardware. The AP allows a programmer to create NFAs and also provides a stream of input symbols to be computed on the NFAs in parallel. This is a fundamental departure from the sequential instruction/data addressing of von Neumann architectures. A benchmark repository for automata-based applications is presented in [42].

The AP re-purposes DRAM arrays for the state-matching and proposes a hierarchical FPGA-style programmable interconnect design. Each AP chip consists of two disjoint half-cores. Each half-core has 96 blocks. Each block provides 256 STEs and thus each AP chip supports 48K STEs Micron's current-generation AP-D480 boards use AP chips built on 50nm DRAM technology, running at an input symbol (8-bit) rate of 133 MHz.

Input and Output The AP takes input streams of 8-bit symbols. The double-buffer strategy for both input and output of the AP chip enables an implicit data transfer/processing overlap. Any type of element on the AP chip can be configured as a reporting element (i.e., accepting state); one reporting element generates a one-bit signal when the element matches the input symbol. If any reporting element reports on a particular cycle, the chip will generate an output vector for all the reporting elements If too many output vectors are generated, the output buffer can fill up and stall the chip. Thus, minimizing output vectors, and hence the frequency at which reporting events can happen, is an important consideration for performance optimization. To address this, we will use the structures that wait until a special end-of-input symbol is seen to generate all of its reports in the same clock cycle.

Programming and Configuration Automata Network Markup Language (ANML), an XML-like language for describing automata networks, is the most basic way to program AP chip. ANML describes the properties of each element and how they connect to each other. The Micron's AP SDK also provides C, Java

and Python interfaces to describe automata networks, create input streams, parse the output and manage computational tasks on the AP board.

Placing automata onto the AP fabric involves three stages: placement and routing compilation (PRC), routing configuration and STE symbol-set configuration. In the PRC stage, the AP compiler deduces the best element layout and generates a binary of the automata network. Depending on the complexity and the scale of the automata design, PRC takes several seconds to tens of minutes. Macros or templates can be precompiled and composed later. This shortens PRC time because only a small macro needs to be processed for PRC, and then the board can be tiled with as many of these macros as fit.

Routing configuration/reconfiguration programs the connections, and needs about 5 ms for a whole AP board. The symbol set configuration/reconfiguration writes the matching rules and initial active states for the STEs and takes 45 ms for whole board. A pre-compiled automaton only needs the last two steps. If only STE states change, only the last step is required.

2.4.2 Cache Automaton

Recently, Subramaniyan et al. [29] proposed an in-SRAM automata processing accelerator, Cache Automaton (CA), by re-purposing last-level cache for the state-matching and using 8T SRAM cells for the interconnect. The state-matching phase is based on the AP model. The crossbar interconnect adds 8T SRAM memory arrays and a 2-level hierarchical switch topology with local switches providing intra-partition connectivity and global switches providing sparse inter-partition connectivity. CA uses a full-crossbar topology for the interconnect to support full connectivity in an automaton. The design has a better state density and throughput compared to the Automata Processor.

To better understand the architecture of memory-centric models, the following example discusses a simplified two-level pipeline architecture of automata processing used in the AP and CA.

2.4.3 Working Example on In-Memory Automata Accelrators

Homogeneous Automaton: The existing in-memory automata processing architectures use the homogeneous automaton representation in their execution model (same as ANML representation in [28]). In a homogeneous automaton, all transitions entering a state must happen on the same input symbol [43]. This provides a nice property that aligns well with a hardware implementation that finds matching states in one clock cycle and allows a label-independent interconnect. Following [28], we call this element that both represents a state and performs input-symbol matching in homogeneous automata a *State Transition Element* (STE).

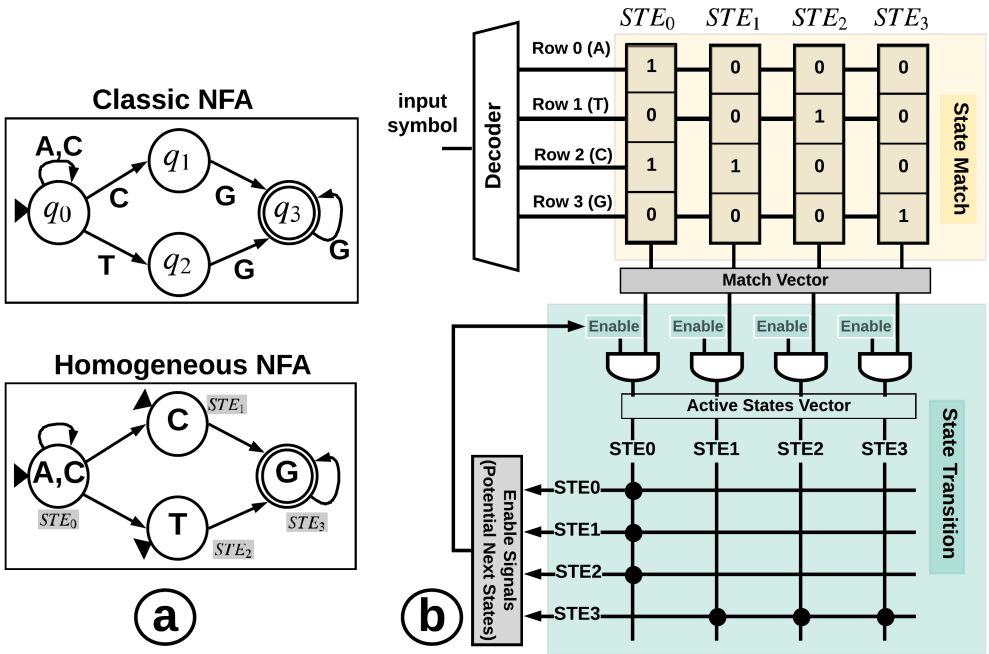

Figure 2.1: (a) Different NFA representation, (b) A simplified in-memory automata processing model

Figure 2.1 (a) shows an example on classic NFA and its equivalent homogeneous representation. Both automata in this example accept the language $(A|C)^*(C|T)(G)^+$. The alphabets are $\{A, T, C, G\}$. In the classic representation, the start state is q_0 and accepting state is q_3. In the homogeneous one, we label each STE from STE_0 to STE_3, so starting states are STE_0, $STE1$, and $STE2$, and the accepting state is STE_3. In all the architectures analyzed in this paper, any states can be starting states without incurring any extra overhead.

In Figure 2.1 (b), memory columns are configured based on the homogeneous example in Figure 2.1 (a) for STE_0-STE_3. Generally, automata processing involves two steps for each input symbol, *state match* and *state transition*. In the state match phase, the input symbol is decoded and the set of states whose rule or label matches that input symbol are detected through reading a row of memory (*match vector*). Then, the set of potentially matching states is combined with the *active state vector*, which indicates the set of states that are currently active and allowed to initiate state transitions; i.e., these two vectors are ANDed. In the state-transition phase, the potential next-cycle active states are determined for the current active states (*active state vector*) by propagating signals through the interconnect to update the active state vector for the next input symbol operation.

In the example, there are four memory rows, and each is mapped to one label (i.e., A, T, C, and G). Each state in the NFA example is mapped to one memory column, with '1' in the rows matching the label(s) assigned to those STEs. STE_0 matching symbols are A and C (Figure 2.1 (a)), and the corresponding

positions have '1', i.e., in the first and third rows (Figure 2.1 (b)). Assume STE_0 is a current active state. The potential next cycle active states (or enable signals) are the states connected to STE_0, which are STE_0, STE_1, and STE_2 (the enable signals for STE_0, STE_1, and STE_2 are '1'). Specifically, if the input symbol is 'C,' then Row2 is read into the *match vector*. Bitwise AND on the *match vector* and *potential next states* (enable signal) determines STE_0 and STE_1 as the current active states.

2.5 Automata Processing on FPGAs

FPGA solutions for accelerating regex and general automata processing are proposed [23, 44, 25, 24]. REAPR [23] is an FPGA-based implementation of an automata processing engine, and takes advantage of the one-to-one mapping between the spatial distribution of automata states and hardware resources such as lookup tables and block RAM. REAPR can achieve approximately 2× to 4× higher clock speeds (250-500 MHz) than the AP, but lower than the estimated clock speed for CA. Large FPGA chips have approximately 2× more STE capacity than a single AP chip, but 3-6× less capacity than CA when utilizing 10-20MB of LLC. Moreover, power consumption of FPGA-based engines is higher compared to the AP and CA.

Yi-Hua et al. [25] propose a multi-symbols processing for NFAs on FPGA and utilizes both LUTs and BRAMs. Their solution is based on a spatial stacking technique, which duplicated the resources in each stride. This increases the critical path when increasing the stride value. Yamagaki [44] propose a multi-symbol state transitions solution using temporal transformation of NFAs to construct a new NFA with multi-symbol characters. This approach only utilizes LUTs, and does not scale very well due to a limited number of look-up tables in FPGAs. Besides, in their multi-striding method, the benefit of improved throughput decreases in more complex regexes (with more characters or highly connected automata) mostly due to routing congestion. Moreover, the recent FPGA-based automata processing solutions fail to map complex-to-route automata to the routing resources due to their logical interconnect complexity [24]

2.6 Automata Processing on ASIC

Several ASIC implementations have been proposed [5, 45, 46, 22] to accelerate pattern matching and automata processing. The Unified Automata Processor (UAP) [22] and HARE[5] have demonstrated line-rate automata processing and a regular matching expression on network intrusion detection and log processing benchmarks. HARE uses an array of parallel RISC processors to emulate the AHO-Corasick DFA representation of regular expression rule-sets. UAP can support many automata models using state transition packing and multi-stream processing at low area and power costs. This framework proposes new instructions to configure the transition

states, perform finite automata transitions and synchronize the operation of parallel execution. UAP uses an array of parallel processors to execute automata transitions and can emulate any automata (not limited to Aho-Corasick). In general, while ASICs provide high line rates in principle, they are limited by the number of parallel matches and state transitions. HARE incurs high area and power costs when scanning more than 16 patterns, and UAP's line rate drops for large NFAs with many parallel active states.

In general, while ASICs provide high line rates in principle, they are limited by the number of parallel matches, state transitions, and shape of the automata. HARE implements DFA and has limitations on the regex shape, and also incurs high area and power costs when processing more than 16 patterns. UAP's line rate drops for large NFAs with many parallel active states. Therefore, they do not provide general and scalable solutions.

Chapter 3

A Scalable Solution for Rule-Based Part-of-Speech Tagging on Novel Hardware Accelerators

As we are living in the era of big data and mobile computing, effective and efficient natural language processing (NLP) applications become increasingly important, and they significantly affect the quality of human-computer interaction (HCI). The most efficient and high-quality NLP applications use extensive, time-consuming statistical or neural-network models, which make them infeasible for real-time applications.

A *part-of-speech tagger* assigns part-of-speech tags (e.g., noun, verb) to words in a sentence. POS tagging is a building block for a wide range of NLP tasks. For example, in parsing, words' parts of speech determine proper word combinations [47]; in named-entity resolution, it identifies the entities and the relationships between them [48]; and in detecting sentiment contrasts, some words could have different sentiments in different parts of speech [49]. Moreover, in software engineering, POS tagging helps in recognizing essential words from software artifacts such as bug reports [50, 51, 52].

Generally in NLP, and specifically in POS tagging, statistical and neural network (NN)-based approaches have been favored over rule-based approaches because they have shown higher accuracy and the training is straightforward to automate, while early rule-based tagging required manual rule generation and execution time increased linearly with the number of rules. This limits the number of rules, thus limiting accuracy. However, rules can now be learned automatically and incorporate textual information (i.e., surrounding tags and words) [53].

In this work, using POS tagging as a case study, we show that hardware accelerators can make rule-based techniques orders of magnitude faster than statistical/ML-based taggers. This allows rule-based approaches to employ more rules and achieve accuracy competitive with statistical techniques. These observations motivate a re-evaluation of rule-based approaches in NLP.

Execution efficiency is addressed by observing that rule-based techniques map well to regular-expressions (regex), which in turn map well to "spatial" hardware that provides a reconfigurable substrate to lay out the rules in hardware. This allows a large number of patterns to be executed in parallel, in contrast to von Neumann architectures such as CPUs that must either handle one rule at a time in each core, or build large lookup tables in memory and the communication between the cores imposes a significant overhead [54]. Specifically, the Automata Processor (AP) and the Field-Programmable Gate Array (FPGA) are two spatial architectures suitable for pattern-matching. They both allow native execution of non-deterministic finite automata (NFAs), an efficient computational model for regular expressions. They achieve this with reconfigurable elements that efficiently implement automata states and matching rules, and reconfigurable routing that efficiently implements next-state activation. A single chip can implement up to tens of thousands of regular expressions, depending on rule complexity, with little or no change in throughput.

We study the relationship between the rule-set size and the accuracy of POS taggers, and observe that more complex rules (from more diverse template rule-sets) and larger rule sets lead to accuracy almost as good as statistical/ML-based techniques, especially with a larger training corpus. With hardware acceleration, this sets up a new tradeoff for designers of NLP applications. The rule-based approach can give much better testing speed, at the expense of a small drop in accuracy and longer training time.

Because the AP or FPGA can efficiently process large and complex sets of regular expressions, we propose that other NLP tasks involving rules or patterns can also be accelerated this way. For instance, in sentiment analysis, negation scope detection solutions [55][56] are typically rule-based.

In summary, this study makes the following **contributions**.

- We study the effect of different baseline taggers and different number of tagging rules for rule-based POS tagging. We show that using the unigram (i.e, one word at a time) tagger as a baseline tagger as well as larger and more complex tagging rules results in a higher testing accuracy.

- We utilize spatial architectures (the AP and the FPGA) by transforming the rules to regular expressions. This approach is scalable in number and complexity of rules. Increasing the number of rules up to several thousand has no overhead on the AP and a minimal overhead on the FPGA, because all the rules are laid out in space across the chip and executed in parallel.

- We compare our solution on the AP and FPGA with several modern statistical approaches. Results show that we can achieve up to 2,600× and 1,914× speedups on the AP and on the FPGA respectively over CPU-based Brill tagging in the rule-matching stage, up to 58× speedup over the Perceptron POS tagger (CPU solution) in the total testing time, and up to 253× speedup over the LSTM tagger (GPU solution) in total testing time at the expense of approximately 1% accuracy in a large corpus.

An important lesson learned from this research is that rule-based POS tagging on hardware accelerators can compete with the accuracy of statistical/ML-based approaches, especially in a larger corpus. As mentioned, this sets up a very interesting tradeoff to evaluate when designing an NLP application: a small decrease in testing accuracy in exchange for vastly faster testing, at the expense of slower training. This suggests that rule-based approaches are valuable for use cases where testing performance is critical, as long as training time can be tolerated.

3.1 Background on Part of Speech Tagging

Part-of-speech-tagging is the process of assigning parts of speech, such as noun, verb, etc., to each word. It has a wide range of applications in parsing, text-to-speech conversion, named entity resolution, machine translation, etc. POS tagging is generally categorized as a rule-based, statistical-based, or neural network-based model. In rule-based methods, tags are assigned based on rules that embody repeatable patterns indicating a specific tag, and in statistical methods, tags are assigned based on a probability model.

Rule-based POS tagging: The rule-based approach is the earliest POS tagging system, where a set of rules is constructed and applied to the text. The rule-based POS tagging identifies the most appropriate tag for each input token based on contextual rules learned in the training phase. A *transformation-based POS tagger* (TBT) [53] is a rule-based tagger that assigns POS tags to words based on linguistic knowledge learned from a training corpus. Then it uses the training information to tag new untagged corpora in the testing phase in two stages. In the first stage, it uses a simple statistical tagger, called a *baseline tagger* or *back-off tagger*, to assign an initial tag (usually the most frequent POS) for a word. In the second stage, the initial tags are updated based on the contextual rules (the learned rules update the tag if the baseline tag is incorrect in this context). The Brill tagger [57] is a rule-based approach that is the most widely used POS tagger for English texts. The same authors also propose an unsupervised approach [58] that assigns all possible tags to the words in the initial step, and in the next step, uses rules to reduce the number of tags to remove the ambiguity. Mohammed et al. [59] improve the original TBT-based approach and propose *guaranteed pre-tagging*, which fixes the initial tags of the words that are known to be correct. This approach works well if prior information is known.

Statistical-based POS tagging: The *Unigram Tagger* is a statistical tagger that assigns the most likely tag to the word based on the training corpus. To identify the most likely tag for each word, a unigram tagger counts the frequency of tags for each word in the training corpus. The default tag *noun* is used for unseen words. The unigram POS tagger is simple and fast, and it is usually used as a baseline tagger for rule-based approaches.

A *Hidden Markov Model* (HMM) tagger assigns POS tags by searching for the most likely tag for each word in a sentence (similar to a unigram tagger). Unlike with the unigram tagger, an HMM tagger detects a tag sequence for a sentence as a whole, instead of assigning a tag for each word independently. First-order and second-order HMM taggers are usually called Bigram and Trigram taggers, respectively. Given a sentence $w_1...w_n$, an HMM-based tagger chooses a tag sequence $t_1...t_n$ that maximizes the following joint probability:

$$P(t_1...t_n, w_1...w_n) = P(w_1...w_n|t_1...t_n)P(t_1...t_n)$$

The *TnT [60] Tagger* (also known as a trigram POS tagger) uses second-order Markov models and considers triples of consecutive words to simplify the probability computation. In TnT, the tag of a word is determined by the POS tags of the two previous words.

The *Maximum Entropy (ME) Tagger* incorporates more complex features into probabilistic models [61]. Given a sentence $w_1...w_n$, an ME-based tagger models the conditional probability of a tag sequence $t_1...t_n$ as:

$$P(t_1...t_n|w_1...w_n) \approx \prod_{i=1}^{N} P(t_i|c_i)$$

where $C_1, ...C_n$ are contexts for each word $w_1...w_n$ in the sentence. An ME-based tagger models features as binary-valued functions representing constraints to compute $P(t_i|c_i)$. It will learn the weights of the features that can maximize the entropy of the probability model using the training corpus. The Stanford POS-tagger [62] is an example of ME-based tagger.

Neural network-based POS tagging: POS tagging simply can be seen as a supervised classification problem where the input of classifier is a word (or a feature-based representation of a word) and the output is the score of belonging to each class (tag). In Avarage Perceptron tagger [63], a huge set of hand crafted features is extracted and provided for a single layer perceptron with linear activation function to classify the word based on its tag. This method selects the class with highest score as the potential tag. Deep neural network solves the potential drawback of designing handcrafted features by letting the network to pick the features by itself. Bi-directional recurrent neural networks-based taggers [64] (e.g., LSTM) perform the tag classification for the whole sentence as a single decision problem and provide the opportunity of utilizing information coming from left and right-side at the same time. However, these benefits come with the computational cost of training and testing a deep neural network.

3.2 Related Work

Regex matching is an important computation kernel in many applications. However, the efficiency on CPUs is not satisfying due to memory bandwidth limitations. Therefore, several regular-expression processing hardware acceleration methods have been proposed. Micron's AP is an efficient semiconductor architecture for parallel automata processing [28]. It uses a non-von-Neumann reconfigurable architecture, which directly implements NFAs in hardware, to match complex regular expressions in parallel. REAPR is a reconfigurable engine for automata processing on the FPGA. It provides a flexible framework which synthesizes RTL for applications involving automata processing with high throughput [23]. iNFAnt2, an optimized version of iNFAnt, is a prototype framework for NFA-based automata processing on NVIDIA CUDA-enabled GPU cards [2], and DFAGE[1][2] is a DFA-based automata processing on GPU. However, neither GPU automata processing engine provides clear advantages over CPU, let alone AP/FPGA [2] [10]; therefore, we focus on the AP and FPGA in our paper. Both the AP and FPGA have been proved their strengths in many different applications [7, 65, 66, 6, 16, 12].

We are aware of very little work to accelerate POS tagging. A recent study [15] of Brill tagging on the AP shows 30x to 270x speedup over the CPU solution. However, [15] only uses 218 rules in Brill and only evaluates them on a small dataset (e.g. picking 5 sample files from the Brown corpus). Furthermore, they just present the speedup for the second stage of the testing phase, which is simply the rule-matching. However, in our paper, we focus on scalability with number of rules and on accuracy, generating more complicated rule template sets and creating up to 4,000 newly-learned rules using the entire Brown corpus (containing 500 sample files), and achieve a better accuracy than [15]. We also study the performance improvements for the whole testing phase (both the baseline tagger and rule-matching stages). Moreover, [15] uses a character position array to process various look-ahead steps among rules, while we propose a padding technique to synchronize the reports from the AP, which works much faster and significantly simplifies the post-processing of the match-reports.

3.3 Methodology

Rule-based part-of-speech tagging is a challenging task on CPUs when the tagging rule set becomes larger and more complex, while the AP and FPGA excel in parallel rule matching even for large numbers of rules. In this section, we show how to implement the tagging task as a parallel regex matching task on these hardware accelerators, including converting tagging rules to NFAs and encoding the input string. Then, we describe

[1] https://github.com/vqd8a/DFAGE
[2] https://github.com/vqd8a/iNFAnt2

how to prepare tagging rules and input text for applying to the AP and the FPGA. Furthermore, we discuss matching results in the post-processing phase and method of tagging rules with character-level features.

3.3.1 Tagging Rules and Input Text Preparation

We create POS tagging rules based on the fnTBL [67] rule-set template. A rule template defines a dependency pattern in the tagging context without assigning specific tags or words. For example, a rule template w_{-1}, w_0 (previous tag and current word) means that the tagger will check the previous part-of-speech tag and current word to determine if the current tag needs to be corrected. Specific rules can be derived from rule templates by filling in specific part-of-speech tags and words. The fnTBL rule-set with 37 templates is shown in Table 3.1. For comparison, we also show the original Brill tagging rule-set with 24 templates.

Table 3.1: The fnTBL Template Set (37 Templates)

w_0, w_1, w_2	* w_0	* (t_1, t_2, t_3)
w_{-1}, w_0, w_1	* w_1	* (t_{-3}, t_{-2}, t_{-1})
w_0, w_1	* w_{-1}	* t_1, w_0, w_1
w_0, w_{-1}	* w_2	t_1, w_0, w_{-1}
w_0, w_2	* w_{-2}	* t_{-1}, w_{-1}, w_0
w_0, w_{-2}	* t_{-1}, t_1	t_{-1}, w_0, w_1
* (w_1, w_2)	* t_1, t_2	t_{-2}, t_{-1}
* (w_{-2}, w_{-1})	* t_{-1}, t_{-2}	t_1, t_2
(w_1, w_2, w_3)	* t_1	t_1, t_2, w_1
(w_{-3}, w_{-2}, w_{-1})	* t_{-1}	—
w_0, t_1	* t_2	** (w_{-1}, w_0)
w_0, t_{-1}	* t_{-2}	** (w_0, w_1)
w_0, t_2	* (t_1, t_2)	** w_{-1}, t_{-1}
w_0, t_{-2}	* (t_{-2}, t_{-1})	** w_1, t_1

* Original Brill templates (24 templates) in the fnTBL sets
** Original Brill templates that are not in the fnTBL sets
() If a specific tag or word is contained in this range

We use the Brill tagger to learn tagging rules, which requires tagged training data and a set of rule templates. We choose the Penn Treebank corpus and the Brown corpus as training data. During training, the Brill tagger generates specific tagging rules based on the rule templates, ranks learned rules by score and picks the top-k rules as learned results. The score of a tagging rule is defined as Equation 3.1, where N_{fixed} is the number of places that a rule can change an incorrect tag to a correct tag, and N_{broken} is the number of places that a rule changes a correct tag to an incorrect tag. A match will not be counted if a rule changes an incorrect tag to another incorrect tag. A higher score means the the rule can correct more tags in training data while limiting incorrect tag changes.

$$Score(rule) = N_{fixed} - N_{broken} \tag{3.1}$$

There are two steps for performing a typical rule-based POS tagging on testing data. The first step is to tag the input text initially with a light-weight tagger such as the unigram or bigram tagger. The second step is to correct initial POS tags using learned tagging rules. We use various baseline taggers for the first step, which can be done fast but may have low tagging accuracy. Then we extract the outputs of the first step, which contains the original text and initial part-of-speech tags, and use them as the input data for our rule matching experiments on the hardware accelerators.

3.3.2 Accelerating Rule-Based POS Tagging on the AP and FPGA

In order to to accelerate the rule-based POS tagging using the AP and the FPGA, we implement the multi-rule tagging task as a parallel regex matching task on the hardware accelerators. Figure 3.1 represents the workflow of rule matching for updating the baseline tags on the AP and FPGA. We first convert all learned tagging rules to regular expressions and then convert regular expressions to ANML representation [28], which is an XML-based file format expressing finite state machines and connections, used on the AP and FPGA. For the AP, we compile these rules directly onto the hardware using Micron's compiler, and for the FPGA, we use REAPR to generate an FPGA configuration as an xclbin and use Amazon's toolchain to create an Amazon FPGA Image (AFI) for the xclbin file. If users use their own FPGAs on local nodes, they do not need to create the AFI. Then, for both the AP and the FPGA platforms, we stream in the encoded input text (e.g., encoded Treebank) to match against all rules. The matching results can tell us which tags match a learned rule and thus needs to be corrected. The hardware will report these results back and the CPU can correct the corresponding tags. For the AP and FPGA, we apply padding in both NFAs and input text to support different degrees of look ahead in tagging rules, so that we can get matching results from the AP synchronously (see below).

Tagging Rules to Regular Expressions Conversion:

Table 3.2 shows how to convert each tagging rule to a regular expression. For rules with fixed words or tags, we directly fill in the words and tags into regex templates. For rules that check if a word or a tag is in a range, we use the regex string-OR operation to represent them. An example rule with ranges derived from template '(w_{-3}, w_{-2}, w_{-1})' is shown below. It means that, if the word '*hadn't*' is shown in last three words, we need to correct the current tag from VBD to VBN, i.e. from past-tense verb to past-participle verb.

```
VBD → VBN: if Word:hadn't@[-3,-2,-1]
```

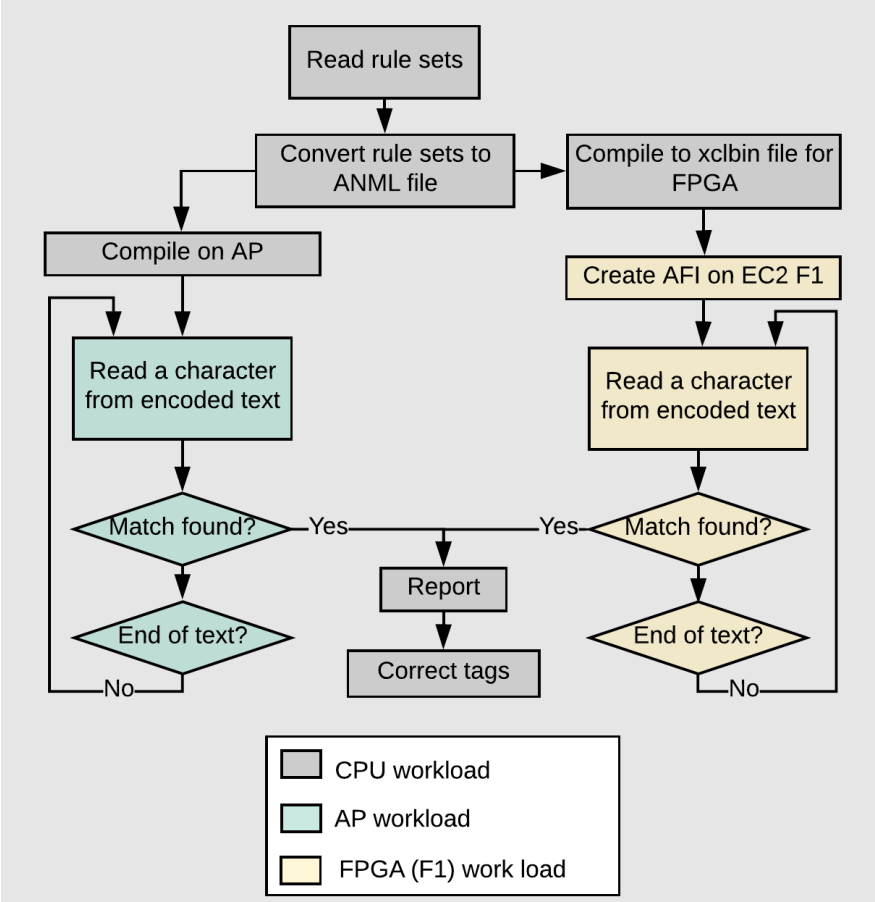

Figure 3.1: POS Tagging workflow on the AP and FPGA.

We use string-OR regex syntax to capture the cases when such a word may appear at any places in the range. A regex for the above range rule is shown below. The string-OR regex syntax can be efficiently converted to NFAs with branches, which the AP and FPGA support.

```
/\s+(hadn't\/[^\s]+\s+[^\s]+\s+[^\s]+|
^\s+\s+hadn't\/[^\s]+\s+[^\s]+|
^\s+\s+[^\s]+\s+hadn't\/[^\s]+)
\s+[^\s]+/VBD\s/
```

Padding Technique: In the fnTBL template set, there are rules with 0 to 3 look-ahead steps, i.e. the tag of a word may depend on some words ahead of itself. If we directly convert these rules to NFAs and match them with the input word sequence on the hardware, we may get matching results of the same word asynchronously, i.e., for a specific word, some matching results may appear at the end of this word, while

Table 3.2: Converting Tagging Rules to Regexes

Rules	**Regex**
Regex rule templates	...word$_{-1}$\/tag$_{-1}$\s+word$_0$\/tag$_0$\s+word$_1$\/tag$_1$\s+word$_2$\/tag$_2$\s+word$_3$\/tag$_3$\s
A known word "word"	word
An unknown word	[^\s]+
A known tag "NN"	NN
An unknown tag	[^\s]+
Word tag delimiter	\/
An unknown word-tag pair	[^\s]+
A word "word" in a range 1 to 3	(word\/[^\s]+\s+[^\s]+\s+[^\s]+|[^\s]+\s+word\/[^\s]+\s+[^\s]+| [^\s]+\s+[^\s]+\s+word\/[^\s]+)

some matching results may appear after streaming in one or two or three more words. This would introduce more overhead when applying the results to update tags. For example, this could occur in a rule derived from template 'w_0, w_1, w_2', which looks two words ahead to determine whether to correct current tag.

To solve such problems, we use a padding technique. We first analyze the maximum look-ahead step in the rule template set, then pad all rules to this maximum look-ahead, so that we can delay some early matching results. The padding technique only consumes a marginal amount of hardware resources for each tagging rule, e.g., two extra NFA state elements for each look-ahead padding step. With this padding technique, the hardware accelerators can conduct regex matching in parallel for consecutive words and tag the words in a pipelined fashion. This technique can generate matching results for each word synchronously, which improves throughput and simplifies the following step. With the padding technique, the output is a vector of 0s and 1s, from which we know which tagging rules are triggered and whether the input POS tag needs to be corrected.

3.3.3 A Working Example

In the training phase of rule-based POS tagging, the following rule has been learned from the Treebank corpus.

```
NN → JJ  if  Word:the@[-1]  &  Word:future@[0]
                  &  Word:growth@[1]
```

It means if $word[-1]$ == the, and $word[0]$ == $future$, and $word[1]$ == $growth$, and $tag[0]$ == NN, then, replace $tag[0]$ which is NN with JJ. The rule is then converted to the following regular expression (according to Table 3.2):

```
/\s+the\/[^\s]+\s+future\/NN\s+growth\/[^\s]+\s/
```

Assume that the maximum look-ahead step is 3, this regular expression is padded to three look-ahead

words as follows:

```
/\s+the\/[^\s]+\s+future\/NN\s+growth\/[^\s]+\s/
+[^\s]+\s/+[^\s]+\s/
```

Figure 3.2 shows the generated automaton on the AP/FPGA for the padded regular expression. The automata generate a report (in the "report" state shown in the figure) when the input stream matches the padded regular expression.

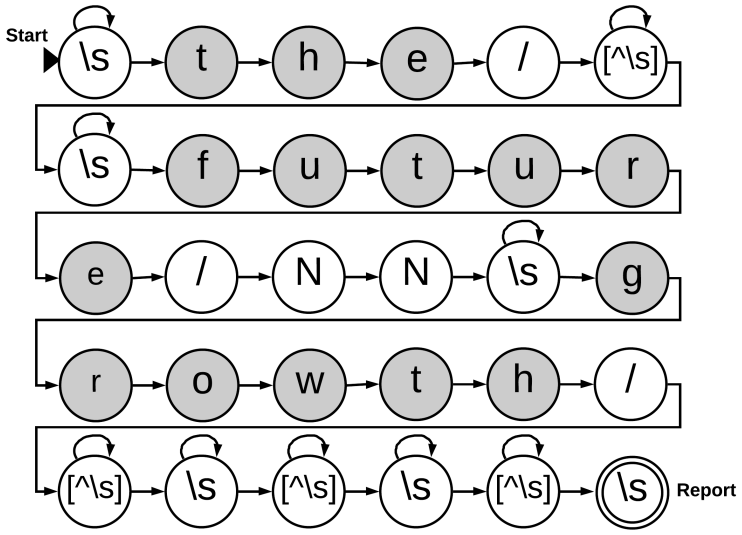

Figure 3.2: An example automaton for a regex rule with padding.

Such automata are stored on the AP/FPGA and the input sequence will be streamed into the hardware. The input string is the baseline-tagged word sequence with dummy word-tag pairs between sentences.[3]

Assume the sentence we intend to tag is "*the future growth of our economy*". After applying the baseline tagger, the words are initially tagged as follows:

```
the/DT future/NN growth/NN of/IN our/PRP economy/NN
```

We encoded the input string with sentence delimiters as follows:

[3] An example input sentence can be separated by space using word-tag pairs, e.g. "$word_0/tag_0$ $word_1/tag_1$ $word_2/tag_2$ ...". If the tagging rules are learned at the sentence-level, we need to use dummy word-tag pairs to separate adjacent sentences, so that the boundary words will not affect other sentences during regex matching, e.g. "$sentence_0$ ./. ./. ./. $sentence_1$ ...". The number of dummy word-tag pairs depends on the maximum look-ahead steps in the rule template set.

```
the/DT future/NN growth/NN of/IN our/PRP economy/NN ./.  ./.  ./.
```

The regex rule mentioned above matches the input string at the space character right after '*our*/*PRP*', so we need to correct the tag for *future* to JJ.

The encoded text is matched against all the tagging rules in parallel. If multiple tagging rules match the input, the tag is updated using the rule with higher score.

One can find the details of our implementations here.[4]

3.3.4 Character-Level Regex Features

Character-level regex can capture features inside a word, which can be more discriminative in POS tagging task. Some example character-level features that can be important include hyphens, uppercase letters, specific prefixes or suffixes, root words, or words with mixed digits and letters. Since the AP and FPGA excel at general regex matching, it will be interesting future work to extend the tagging rule set to include rules with character-level features without significant performance overhead.

3.4 Experimental Results

3.4.1 Execution Environment and Data Sets

We perform experiments on a Linux server with a 3.3GHz Intel Core-i7 5820k CPU and 32GB DDR4 RAM. GPU experiments use an NVIDIA K80 in this same system. We use taggers in NLTK 3.2.1 in Python 2.7 as our baseline taggers. In addition, NLTK contains Java interface for running the Stanford log-linear tagger (3.6.0). For all measurements, I/O times are excluded, assuming data are already loaded.

We use the Penn Treebank and Brown corpora, the built-in corpora in NLTK, as our datasets. The Penn Treebank corpus contains 199 tagged documents (wsj_0001 to wsj_0199), 3,914 sentences and 100,676 words. The Brown corpus contains 500 documents, 57,340 sentences and 1,161,192 words. Some experiments are performed using just the news category of the Brown corpus, which has 44 documents, 4,623 sentences and 100,554 words.

On both the AP and FPGA, because all regexes are processed in parallel, a new input symbol can be processed every clock cycle. The kernel execution time of the AP is estimated, because fully-functional AP hardware is not yet available. But it is simple to estimate, because with the input processing rate fixed at one character per cycle at 133 MHz, throughput is 133MB/s. The kernel execution time on FPGA is evaluated on

[4] https://github.com/elaheh-sadredini/BrillPlusPlus

the Amazon EC2 F1 instance equipped with a Xilinx Virtex UltraScale VU9P FPGA and four memory banks. The synthesized clock rate can vary with rule complexity, with a maximum of 250 MHz. The deployment on F1 also allows other users who have access to EC2 to easily use our proposed method or reproduce the results.

Table 3.3: Testing accuracy of the Brill tagger with different baseline taggers with 5-fold cross validation on the Treebank corpus and Brown news corpus.

Baseline	Treebank			Brown (News)			Entire Brown		
	Baseline	Brill (24)	fnTBL (37)	Baseline	Brill (24)	fnTBL (37)	Baseline	Brill (24)	fnTBL (37)
RU	91.37	93.76	**93.82**	87.58	91.03	**91.14**	92.60	94.36	94.55
RUB	92.26	92.60	92.65	88.59	89.55	89.71	92.69	94.45	**94.59**
RUBT	92.16	92.32	92.37	88.51	89.28	89.37	92.74	94.18	94.31

3.4.2 Accuracy of the Brill Tagger

In order to study how baseline taggers affect the overall testing accuracy of a rule-based tagger like Brill, we evaluate the accuracy of Brill tagger using unigram tagger (U), bigram tagger (B), and trigram tagger (T) as the baseline on Treebank, Brown News, and entire Brown corpus with 5-fold cross validation on the datasets. We also test the Brill tagging testing accuracy with both the original 24 rule templates and the fnTBL 37 rule templates.

Results are shown in Table 3.3. For each corpus, the first column (Baseline) represents the baseline testing accuracy for the corresponding baseline tagger. The second and third columns show testing accuracy when using the corresponding baseline tagger as their back-off tagger for 24 rule-templates and fnTBL 37 rule-templates respectively. The maximum number of rules generated for Brill is 500 for this experiment.

For the unigram tagger, we use the regex tagger as its backoff tagger (denoted as RU). The regex tagger (R) can assign tags to words based on common rules, such as defining ".*able" as adjective and defining ".*ness" as noun. Since we only use 9 common rules, the accuracy of pure regex tagger is very low (23.92% on the Treebank corpus, 30.41% on the Brown news corpus, and 29.61% on the entire brown corpus). For the bigram tagger, we use the unigram tagger as the backoff of the bigram tagger (denoted as RUB). Finally, for the trigram tagger, we use the bigram tagger as the backoff of the trigram tagger (denoted as RUBT).

Results show that by choosing the unigram tagger (RU) as the baseline, the Brill tagger achieves the highest testing accuracy for Treebank and Brown (news) corpora, which is 1% more than choosing other taggers as baseline taggers. The reason for this is that Treebank corpus and Brown (news) are small, so there are many unseen words in the bigram and trigram taggers, and they may overfit the training data. Interestingly for entire Brown corpus, by using bigram tagger (RUB) as the baseline tagger, the brill tagger achieves the highest testing accuracy. This is because there are fewer unseen words in the bigram tagger model for larger datasets. Furthermore, for all corpora, the larger rule template set (fnTBL 37) helps to improve

Table 3.4: Testing accuracy (%) and testing time (in seconds) for Brill++ on CPU, AP, and FPGA for Treebank and Brown (news) corpora while increasing the number of tagging rules.

#Rules	Treebank				Brown (News)			
	Test Acc (%)	Test Time (second)			Test Acc (%)	Test Time (second)		
		CPU	AP	FPGA		CPU	AP	FPGA
100	93.57	0.23	0.0015	0.0008	90.36	0.345	0.0015	0.0008
200	93.73	0.37	0.0015	0.0008	90.78	0.475	0.0015	0.0008
300	93.76	0.52	0.0015	0.0009	91.00	0.594	0.0015	0.0009
400	93.82	0.55	0.0015	0.0009	91.09	0.720	0.0015	0.0009

the accuracy. This shows that more diverse and complex template for rule-set is beneficial to accuracy, but processing them on the CPU is very time-consuming, and this is where having a hardware accelerators can play an important role.

3.4.3 Brill tagging with different number of rules

In the training phase of Brill tagging, we can set a score threshold and the number of rules to be learned. More rules usually lead to higher training/testing accuracy, although too many rules may cause overfitting. In this section, we show that a larger number of rules significantly increases computation time on the CPU and slows down the training and testing speed. However, the AP and FPGA shows a constant or near-constant processing time for testing when the number of rules increase. In this work, we focus on improving the testing time, because the learning phase is executed rarely, and the results are used many times for new texts.

Table 3.4 presents the testing accuracy and testing time (for rule-matching stage) of the Brill tagger when increasing the number of generated rules on Treebank and Brown news corpora. We refer to Brill as Brill++ when using our approach for increasing the number of rules. We choose the unigram tagger as the baseline tagger, and learn 100 to 400 rules from the training folds based on the fnTBL 37 rule templates (based on the results in Section 3.4.2, unigram tagger and fnTBL rule templates perform better). We observe that testing accuracy improves in both corpus when increasing the number of learned rules.

Table 3.4 also shows rule-matching time, i.e., the testing stage, for the Brill++ tagger on the CPU, AP, and FPGA for the Treebank and Brown news corpora. The testing time of the Brill tagger on the CPU is proportional to the number of rules and it increases when generating more rules. However, the computation time on the AP remains constant (0.0015 seconds) as long as the rule-set can fit on the AP board. Moreover, the computation time for the FPGA is even less than the AP, which is about 0.0009 second for both corpora. This is because all the rules configured on the AP and FPGA can be matched against the input stream (the baseline tagged word sequence) in parallel at the rate of 133MB/s for the AP and 250MB/s for the FPGA.

Table 3.5: Testing accuracy (%) and testing time (in seconds) for Brill++ on CPU, AP, and FPGA for the entire Brown corpus while increasing the number of tagging rules.

#Rules	Entire Brown Corpus				
	Test Acc(%)		Test Time (second)		
	Brill++ RU	Brill++ RUB	Brill++		
			CPU	AP	FPGA
100	93.48	94.02	2.40	0.0172	0.0093
200	93.98	94.28	3.82	0.0172	0.0093
300	94.25	94.41	5.44	0.0172	0.0098
400	94.43	94.52	6.90	0.0172	0.0098
500	94.58	94.6	8.40	0.0172	0.0098
1000	94.90	94.8	15.7	0.0172	0.0098
2000	95.17	94.91	30.05	0.0172	0.0157
3000	95.25	94.94	40.02	0.0172	NA
4000	**95.29**	94.96	44.59	0.0172	NA

We perform a set of similar experiments on the entire Brown corpus, which has 500 documents and 1.16 million words. Table 3.5 shows the testing accuracy of Brill++ when the number of rules increases from 100 to 4,000. We run the experiments for Brill++ using both unigram tagger and bigram tagger as the back-off tagger, with baseline accuracy of 92.60% and 92.69% respectively. Accuracy is improved up to 95.21% when increasing the number of learning rule from the training folds. In Section 3.4.2, we observed that bigram tagger performs better as the baseline tagger for Brill on the entire Brown corpus. However, when increasing the number of rules, we see that the unigram tagger starts to perform better. This implies that more tagging rules work best with a simpler baseline tagger. Therefore, we use unigram tagger as a reliable baseline tagger for rule-based taggers, independent of the corpus size.

Table 3.5 also presents the rule-matching time for the Brill++ tagger on the CPU, AP, and FPGA for the full Brown corpus. The length of input string generated from testing folds of the Brown corpus is about 2.3MB and the AP frequency is 133MB, so the AP testing time is only 17ms (calculated as 2.3MB / 133MB/s). The FPGA rule-matching kernel is around 2× faster than the AP, and this is because the rule processing frequency on the FPGA is higher than the AP (about 250MB/s). However, the testing time of the Brill++ on the CPU consumes up to 44.59 seconds.

As a result, if we only compare the matching part, that is, deducting the baseline tagging time from the Brill++ tagging time, the AP and FPGA can achieve up to 2600× and 1914× speedup over the CPU-based implementation respectively.

3.4.4 Performance Discussion and Future Work

Errors in initial stages of an NLP pipeline have negative effects on the overall accuracy. Therefore, the main focus of many state-of-the-art POS taggers is to improve the accuracy. However, the runtime of POS taggers

Table 3.6: Timing/accuracy trade-off for different methods on Treebank corpus and entire Brown corpus.

Method	Character Embedding	Treebank			Entire Brown		
		Train Time (s)	Test Time (s)	Test Acc	Train Time (s)	Test Time (s)	Test Acc
Brill (CPU)	No	27.21	0.55	93.82	4980	45.43	95.29
Brill++ (AP-FPGA*)	No	27.21	**0.091 (0.090*)**	93.82	4980	**0.837 (0.835*)**	95.29
TnT (CPU)	No	0.46	157	89.95	3.74	15736	94.05
Stanford Tagger (CPU)	No	NA	3.39	91.30	NA	117.58	62.86
Perceptron (CPU)	No	17.01	0.82	95.89	941	48.61	96.24
LSTM (GPU)	No	210.64	1.25	89.3	1832	184.29	91.7
LSTM-ChE (GPU)	Yes	223	2.78	**96.15**	2676	212.06	**96.67**

is very important for time-sensitive tasks (e.g., online machine translation). Therefore, in this section, we discuss the trade-off between accuracy and time for different methods.

Table 3.6 shows training time, testing time, and testing accuracy of rules-based, statistical-based, and neural network (NN)-based approaches on Treebank and entire Brown corpora. Testing times for Brill and Brill++ include both baseline tagger and rule-matching stages. For the Brill++ (AP-FPGA*), numbers on the parenthesis with asterisks represent testing time for the FPGA. The TnT (Trigrams'n'Tags) tagger [60] is a statistical POS tagger based on Markov models. The Stanford POS tagger [68] is also a statistical POS tagger based on a maximum-entropy model. The Stanford tagger is pre-trained on the TreeBank corpus, so we do not report the training time for that. Moreover, because the Stanford tagger is trained on Treebank, its accuracy on Brown corpus is low. The Perceptron tagger (or averaged Perceptron tagger) is a one-layer NN-based solution. TnT, Perceptron, and Stanford Taggers are all from the NLTK package[5]. LSTM[6] is a bidirectional long-short term memory tagger using conditional random field (CRF). LSTM is based on word-level features while LSTM-ChE employs character embedding features in addition to word-level features (both run on the K80 GPU). The testing time is measured using mini-batch size of 20.

The TnT tagger has the lowest accuracy and the longest testing time. However, it has the shortest training time. The Perceptron tagger has the highest accuracy among the methods that does not use character-embedding information, for both corpora. However, its testing time on the CPU is up to 58× slower than Brill++ on the AP and the FPGA. Perceptron tagger would have a better performance on GPUs and will be an interesting point of comparison for the future work. Brill++ has the second highest testing accuracy and by far the lowest testing time (on the AP and FPGA) among the taggers that does not use character-embedding features. The training time of the rule-based approach is higher than Perceptron tagger. Although the training is conducted just once and the rules are used multiple times for the unseen textual data, accelerating the training phase of Brill++ using the AP or the FPGA or other hardware accelerators is an interesting area for future work.

[5] http://www.nltk.org/api/nltk.tag.html
[6] https://github.com/guillaumegenthial/sequence_tagging

We ran LSTM and LSTM-ChE for 7 epochs on the Treebank corpus and 4 epochs on the Brown corpus. Results show that by adding character-embedding feature to LSTM-ChE, the accuracy can increase by 6.85%. Clearly, LSTM-ChE achieves the highest accuracy among other methods; however, testing time of Brill++ on the AP/FPGA is still 253%× less than LSTM-ChE on the GPU.

Most state-of-the-art POS taggers that report high accuracy (about 96%- 97%) take advantage of character-level features in addition to word-level features [69, 70, 71]. A recent study on machine-learning-based POS taggers [72] compares the accuracy of three state-of-the-art taggers, MarMot[7] (a generic conditional random field framework), bilstm-aux[8] (bidirectional long-short term memory tagger) and its own CNN-based tagger for three variations of input features: word only, character only, and word-character combination ([72], Table 1). The results show that combining word feature and character feature can increase the accuracy by 2%-3%.

Compared to word-level POS taggers, Brill++ achieves competitive accuracy with a superior short runtime. Based on the character-level POS tagger study, we hope that adding character-level rules will increase the accuracy of rule-based POS taggers by a similar margin, and make Brill++ fully competitive in accuracy to these statistical/ML-based approaches with superior efficiency. The AP and FPGA have plentiful capacity to extend the tagging ruleset with character-level features while maintaining good runtime.

3.5 Conclusions

The main objective of this paper is to motivate re-consideration of rule-based approaches when real-time computation is needed for NLP applications. To this end, we utilize two state-of-the-art accelerators, the Automata Processor and FPGA, and propose an efficient, rule-based POS tagging approach. We observe that increasing the number of rules, especially from more diverse template-sets and in a larger corpus, results in a higher accuracy that nearly matches the accuracy of statistical/ML-based approaches. Increasing the number of rules only adds minor computational overhead on the AP and FPGA, while the processing time of CPU solutions increases linearly with the number of rules. This is because both hardware accelerators can process a large number of rules against the input corpus in parallel, due to their abundant hardware resources that lay out all the rules in space for concurrent processing. The results show orders-of-magnitude speedup over CPU-based solutions, thus providing NLP application designers with a tradeoff between losing a small amount of accuracy (approximately 1%) in exchange for much faster processing.

[7] http://cistern.cis.lmu.de/marmot/
[8] https://github.com/bplank/bilstm-aux

Chapter 4

Frequent Subtree Mining on the Automata Processor: Challenges and Opportunities

Frequent subtree mining (we name it FTM for not to confuse it with frequent structure mining) refers to finding all the patterns in a given forest (database of trees) whose support is more than a given threshold value, called the minimum support. A subtree pattern is called frequent if the number of trees in the dataset that have at least one subtree isomorphic to the given pattern is more than the minimum support. Frequent subtrees have proven to be extremely important and informative in many real world applications such as XML data, parse-trees in natural language processing, bioinformatics, and patient treatment awareness. For instance, in natural language processing (NLP), frequent subtrees mined from the parse tree databases can be used to increase the accuracy of NLP tasks, such as sentiment analysis and question classification problems [73]. However, finding all frequent subtrees becomes infeasible for a large and dense tree database, due to the combinatorial explosion of the subtree candidates.

Any mining process, including subtree mining, has two steps, candidate generation and candidate enumeration. The first one generates candidate subtrees, which are evaluated for their frequency in the latter stage. The main challenges in FTM are efficiently traversing the search space and performing subtree isomorphism. A number of research studies have attempted to improve the performance of the task by proposing different data structures and counting strategies. These are either based on breadth-first search (BFS) or depth-first search (DFS). BFS is a level-wise iterative search method and usually uses a horizontal

tree representation. BFS suffers from a long processing time because it requires passing through the entire dataset in each iteration. However, DFS usually projects the database into a vertical tree representation for fast support counting, but encounters memory capacity challenges and costly I/O processing because the set of candidates and their embedding list tend to overflow memory [74][75].

Researchers are increasingly exploiting accelerators as performance growth in conventional processors is slowing down. The Micron Automata Processor (AP) is a non-von Neumann, native-hardware implementation of non-deterministic finite automata (NFA). The high bit-level parallelism of the memory-based architecture makes it capable of performing high-speed search and analysis on complex data structures. Recent studies on frequent itemset mining [8], sequential pattern mining [7], disjunctive rule mining [9], and entity resolution [16] have proved that the AP is a promising target accelerator in data-mining and data-matching applications, and these studies have achieved orders of magnitude speed-up over conventional processors. However, the main difficulty in exploiting the AP for FTM is that the AP was intended to support regular languages, whereas tree structures will typically need to be represented by a context-free grammar. By relaxing some of the tree structure constraints, the AP can be effectively utilized to prune the search space of FTM.

In this work, we first study difficulties of directly implementing the FTM problem on the AP platform. Then, we propose a multi-stage pruning framework to greatly reduce the search space of embedded FTM on the AP. This provides a scalable solution in terms of both memory and execution time on large databases and lower support thresholds. Frequent subtree candidates can be the potential features in classification tasks, and the surviving candidates with lower frequency are especially beneficial to boost classification accuracy of rare classes, because these frequent subtree patterns can represent unique and discriminative features of classes with fewer members. In order to maintain both ancestor-descendant relationship and sibling properties on the tree structure, and at the same time provide a feasible computation to exploit the AP, four complementary string representations of the tree structure and their mapping to the automaton representation are proposed. Pruning kernels presented in this work result in a set of potentially frequent candidates (close to the final set of frequent patterns) which may contain a small number of false positives (however, recall is 100%). For the applications that demand an exact solution, we adapt TreeMinerD [74], a quick DFS approach that detects the distinct occurrences of a pattern, to prune the AP results and provide an exact solution. Finally, we develop a BFS-based solution for embedded FTM on GPU in order to compare the AP solution with an additional accelerator architecture.

4.1 Frequent Subtree Mining

4.1.1 Problem Statement

A *tree* is an acyclic connected graph, and a *forest* is an acyclic graph of multiple trees. A *rooted tree* is a tree with one distinct node called *root*. A tree can be defined as an *ordered* or *unordered* tree. In the *ordered tree*, the children of each node are ordered from left to right according to some common property, so we can enumerate them from left to right as the first child, second child, and so on. If the order does not matter, the tree is called *unordered*. And finally, a *labeled tree* is a tree where each node has an associated label.

Tree Mining Problem: We define D to be a dataset of trees and a transitive subtree relation $S \preceq T$ for some of the trees (T) in D. Define $t_1, t_2, ..., t_n$ to be the nodes in T and $s_1, s_2, ..., s_m$ be the nodes in S. Then, S is a subtree of T if there are matching labels of the nodes $t_{i_1}, t_{i_2}, ..., t_{i_m}$ such that (1) $label(s_k) = label(t_{i_k})$ for all $k = 1, 2, ..., m$; and (2) for every branch (s_j, s_k) in S, t_{i_j} should be an ancestor of t_{i_k} in T. The latter condition preserves the structure of S in T. This definition of subtree refers to an *embedded subtree*. By restricting the ancestor-descendant relationship to parent-child relationships in T for the second condition, a new kind of subtree, called *induced subtree*, can be defined. Fig. 4.1 shows an example on different types of subtrees on T_0. There are several other subtree types such as maximal subtree, closed subtree, and partial subtree, which put restrictions on the induced and embedded subtrees and are not considered in this work.

The relative minimum support number (*Rminsup*), defined as the ratio of minimum support number to the total number of transactions (input trees), is used in this paper. We define the size of a tree as the number of nodes in it. Moreover, we represent a candidate of size k with *k-candidate* and a frequent candidate of size k with *k-frequent-candidate* throughout the paper. Many applications are only interested in counting the number of database trees that contain at least one match of a subtree, which is called counting distinct occurrences. On the other hand, weighted counting refers to enumeration of all possible occurrences over all possible trees in the database. In this work, we focus on mining distinct occurrences of embedded subtrees from rooted, ordered, and labeled trees as those types of datasets dominate in data mining applications [74]. Embedded subtree mining has a larger search space and higher mining complexity than induced subtrees [76] [74], and CPU solutions have difficulties dealing with them. The proposed pruning method is not limited to binary trees and can be adopted by unordered embedded and ordered/unordered induced subtree mining with minimal changes.

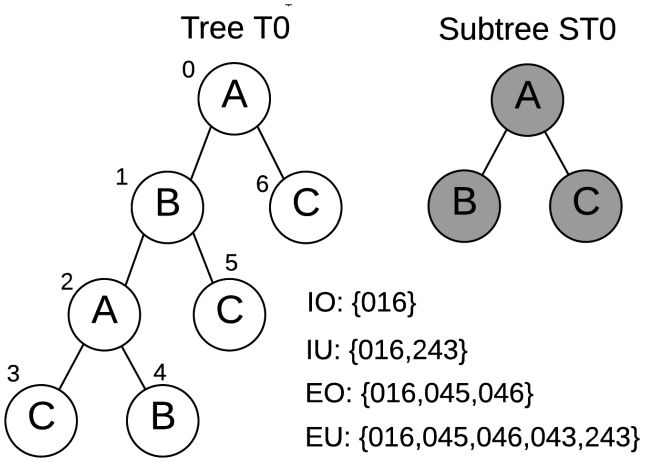

Figure 4.1: An example of subtrees (I = Induced, E = Embedded, O = Ordered, U = Unordered)

4.1.2 Candidate Generation

Our candidate-generation step is based on an equivalence-class, right-most extension approach adopted from [77]. In this approach, the *(k+1)-candidates* are generated from the known *k-frequent-candidates* within an equivalence-class (having the same string prefix). Two frequent patterns can be merged based on the position of the last extended node. In this approach, all the candidates are generated once (avoiding redundancy) and all are the valid candidates. We do not describe the candidate-generation method in detail, as the main focus of the paper is accelerating the candidate enumeration step, which is the bottleneck of the algorithm. Details of the candidate-generation and proof of correctness can be found in [77].

4.2 State-of-the-Art Subtree Mining Algorithms on CPUs and GPUs

A considerable amount of research has been devoted to frequent subtree mining, due to its significance in different domains such as bioinformatics, web mining, and natural language processing. Frequent subtree mining techniques can be classified based on the subtree and the ordering types to induced ordered, induced unordered, embedded ordered, and embedded unordered subtrees. The way the candidate patterns are generated, the data structure representation in the memory, and the candidate subtree enumeration approach can significantly affect the efficiency of the algorithm.

Several algorithms have been proposed to mine labeled, embedded, and ordered subtrees. TreeMiner [77] is the first algorithm for mining embedded ordered subtrees, suggested by Zaki, which is based on DFS search,

and it introduces the concept of vertical tree representation, a space-efficient string encoding of the tree. New candidates are generated by adding one node to the right-most path of the tree (right-most extension approach). TreeMiner uses an *extension and join* approach for the candidate-generation and enumeration and stores the matches in a vertical representation, which can be very large and consume a lot of memory, especially when the number of overlapping matches is high. The same author proposes TreeMinerD [74], an algorithm which enumerates only distinct occurrences of a pattern, and can be beneficial for some applications. However, when the average number of embeddings of a subtree in a tree is low, TreeMiner and TreeMinerD have almost the same performance. XSpanner [78] is another solution for mining embedded, ordered subtrees and adopts the *FP-Growth* concept and its enumeration model generates valid candidates, and counts the distinct trees. An expensive pseudo-projection phase results in poor cache performance in XSpanner. The idea of pseudo-projection techniques is that, instead of physically constructing a copy of the subtree, they reuse the trees in the original tree database. MB3-Miner [79], yet another solution, applies a *Tree Guided Model* to efficiently generate the candidates. However, Tatikonda in [75] shows that the MB3-Miner solution suffers from high memory usage.

The TRIPS and TIDES [80] solutions are proposed to mine embedded and induced subtrees that can be ordered or unordered. They are based on sequential encoding strategies that provide fast generation of a complete and non-redundant set of candidate subtree patterns. They use an embedded list for the candidate-generation and a hash table for the support counting step. Even though their approaches are cache-conscious due to the simple array-based data structure, they still suffer from high memory consumption with larger datasets and with smaller support thresholds. The same authors proposed an architecture-aware FTM algorithm [75] targeting multi-core systems. Several optimizations are adopted to decrease memory access latency and bandwidth pressure, and a parallel *pattern-growth* approach in the context of the TRIPS [80] algorithm on multicore systems is proposed. They show a nice scale-up with the number of cores, however, their solution crashes far sooner than a single-core implementation. This work is the only parallel solution for FTM, to the best of our knowledge.

PATTERN-MATCHER [77] instead employs a breadth-first iterative search to find frequent subtrees. It employs an equivalence-class notion for candidate-generation and counting, and a prefix-tree data structure for indexing the candidates. It prunes the candidates of size $(k+1)$ using the frequent candidates of size k. We consider this algorithm as a baseline to compare with TreeMiner. Furthermore, a level-wise push-down automata-based approach for the candidate enumeration step of FSM is proposed in [31]. A deterministic finite automata is generated for each candidate and the experiments are run on a Pentium 4 CPU. However, the authors do not compare the performance of their method with the *FP-growth* algorithms, which are known to be faster than the level-wise solutions.

Chopper [78] proposes a two-stage solution to find frequent subtrees. In the first stage, the database is converted to a preorder traversal label sequence representation and then, frequent sequences are found using sequential pattern mining, which acts as the pruning stage. Then, frequent subtrees are found by removing the false positives. XSpanner outperforms Chopper [78] and Tree Miner [77] outperforms Xspanner, so we do not consider them in the performance comparison. Furthermore, there is no parallel implementation of frequent subtree mining problem on GPU or FPGA architectures.

4.3 Frequent Subtree Mining on the Automata Processor: Challenges

The AP mainly supports regular languages, however, boolean and counter elements provide stack functionality with a very limited element size, stack size, and population on the AP. We have designed a balanced parenthesis checking structure for the induced FTM problem, which implements a simplified stack structure using counter and STE elements on the AP. The parenthesis checking structures keep track of the branch position in the subtree and require to be repeated for each node in the subtree, and this repetition consumes a large portion of the available STEs and counters. Furthermore, the depth of the parenthesis checking structure depends on the maximum depth of the trees in the database, which makes it impossible to have a database-independent solution. The actual stack functionality is also designed using boolean and STE elements for embedded tree mining problem. For a stack of size 3 and symbol-set of size 4, the stack design needs more than 20 booleans and 48 STEs. Subsequently, it is practically impossible to extend the design for larger symbol set and deeper stack.

Therefore, subtree inclusion checking cannot be accomplished using deterministic finite state machines. The tree structure is more complex than a sequence and cannot be described with regular languages [31]. It implies that instead of a finite state machine, a pushdown automaton (PDA) is needed in order to count the length of a possible branch when searching for a subtree in the input tree. A PDA is a finite automaton with access to a potentially unlimited amount of stack, which is more capable than finite state machines.

We concluded that the AP is an excellent accelerator to *prune* the search space of the candidates in FTM, when relaxing some of the tree constraints in order to make the simpler representations of a tree. In the following section, we propose a set of pruning kernels implemented on the AP to shrink the subtree candidate-set size, which provides a scalable solution to the larger databases and lower support thresholds.

4.4 Frequent Subtree Mining on the Automata Processor: Opportunities

The AP architecture excels at computing regular expressions and as we have already discussed, it is not practical to directly and accurately implement the enumeration step of FTM on the AP. In addition, the existing techniques for itemsets and sequential patterns mining cannot be naturally and accurately extended to the FTM problem. Nevertheless, the AP huge parallelism can be exploited to prune the large search space of the candidate enumeration by simplifying the tree structure to some elementary representations such as FIM and SPM, where the AP can understand and directly implement them.

In this section, we propose four kernels, *(1) subset pruning, (2) intersection pruning, (3) downward pruning, and (4) connectivity pruning.* The first two kernels are independent from the input transaction, while the last two create a new presentation of the trees in the database and use them as the input stream to match against the candidates. The proposed kernels are complementary to each other to avoid overlapping pruning and applied to the candidates in sequence to accommodate more candidates in the early stage.

4.4.1 Preliminaries

Frequent Itemset Mining (FIM): The FIM problem was initially studied to find regularities in the shopping behavior of customers of supermarkets and has since been applied to very broad application domains. We define $I = i_1, i_2, ..., i_m$ as a set of interesting items. Let $T = t_1, t_2, ..., t_n$ be a database of transactions, each transaction t_j is a subset of I. Define $x_i = \{i_{s1}, i_{s2}, ..., i_{sl}\}$ be a set of items in I, called an itemset. The itemset with k items is called k-itemset. A transaction t_p is said to cover the itemset x_q *iff* $x_q \subseteq t_p$. The support of x_q, $Sup(x_q)$, is the number of transactions that cover it. An itemset is frequent *iff* its support is greater than a given threshold value called minimum support, *minsup*. The goal of FIM is to find all itemsets with support greater than *minsup*. Wang et al. [8] proposed a novel automaton template (an automata structure that can be configured with different symbol-set) for matching and counting stage of FIM on the AP that can handle variable-size itemsets (ME-NFA-VSI) and avoid routing reconfiguration. The whole design makes full usage of the massive parallelism of the AP. By using this template structure, one AP board can match and count 18,432 itemsets in parallel with sizes from 2 to 40 for 8-bit encoding and 2 to 24 for 16-bit encoding (for symbol alphabets > 256). Note that the processing rate is 133 MB/s regardless of encoding.

Sequential Pattern Mining (SPM): We define $I = i_1, i_2, ..., i_m$ as a set of items, where i_k is usually represented by an integer, call item ID. Let $s =< t_1 t_2 ... t_n >$ denotes a sequential pattern, where t_k is a transaction and also can be called as an itemset. We define an element of a sequence by $t_j = \{x_1, x_2, ..., x_m\}$

where $x_k \in I$. We assume that the order within a transaction (itemset) does not matter, so the items within one transaction can be *lexicographically ordered* in preprocessing stage. A sequence with a size k is called a k-sequence. Sequence $s_1 =< t_1 t_2 ... t_m >$ called to be a subsequence of $s_2 =< r_1 r_2 ... r_j >$, if there are integers $1 \preceq k_1 \prec k_2 \prec .. \prec k_{m-1} \prec k_m \preceq j$ such that $t_1 \subseteq r_{k1}, t_2 \subseteq r_{k2}, ..., t_m \subseteq r_{km}$. Such a sequence s_j is called a sequential pattern. A sequence is known as frequent *iff* its support is greater than a given threshold value called minimum support, *minsup*. The goal of SPM is to find out all the sequential patterns with support greater than *minsup*. In [7] Wang et al. derive a compact automaton design for matching and counting of SPM on the AP. A key insight that enables the use of automata for SPM is that hierarchical patterns of sequences can be flattened into strings by using delimiters and place-holders. Again, a template is proposed to accommodate variable-structured sequences. This allows a single, compact template to match any candidate sequence of a given length, so this template can be replicated to make full use of the capacity and massive parallelism of the AP. One AP board can match and count 6,144 sequence patterns in parallel with sizes from 2 to 20 for 8-bit and 16-bit encoding. The detail of the design capacity can be found in [7]. Table 4.1 and 4.2 represent capacity information of the SPM macros for different sequence sizes and support threshold.

Table 4.1: Number of macros that fit into one block with 8-bit encoding

	k <= 10	10 < k <= 20	20 < k <= 40
sup < 4096	4	2	1
sup >= 4096	2	2	1

Table 4.2: Number of macros that fit into one block with 16-bit encoding

	k <= 5	5 < k <= 10	10 < k <= 20
sup < 4096	4	2	1
sup >= 4096	2	2	1

4.4.2 Pruning Kernels

We propose four pruning kernels in this section. Each kernel maps to FIM or SPM definition, and we use the automata structures proposed for FIM [8] and SPM [7] problems, from our previous works, to implement the kernels on the AP and calculate the AP board utilization.

Subset Pruning:

According to the downward-closure principle, all sub-patterns of a frequent pattern must themselves be frequent. This means that, when generating a *(k+1)-candidate*, all of its *k-candidates* should be frequent as well. BFS-based FTM approaches can greatly benefit from this property in order to reduce the search space,

whereas DFS implementations do not have all the *k-frequent-candidates* when looking at a *(k+1)-candidate*. The subset pruning kernel checks the downward closure property for all the candidates of size three and more. This property can be directly mapped to FIM, where each generated *(k+1)-candidate* represents a candidate itemset and the items in the itemset are the set of *k-candidates*. In Fig. 4.2, C_{5i} (a *5-candidate*) is generated from F_{4i}, which is a *4-frequent-candidate*, by extending the edge AE. In subset pruning, we should check C_{4j}, C_{4k}, and C_{4l}, the other subsets of C_{5i}, to verify they are frequent as well. The itemset candidate corresponding to C_{5i} is $C_{5i} = \{C_{4j}, C_{4k}, C_{4l}\}$ and the input dataset has only one transaction, which consists of all the frequent candidates of size 4, e.g., $\{F_{40}, F_{41}, ..., F_{4m}\}$, where m is the number of *4-frequent-candidates*. Therefore, the set of all C_5 creates the candidate itemsets for FIM. A subtree candidate will survive at this stage if it occurs in the input transaction (*Rminsup* is 100% here).

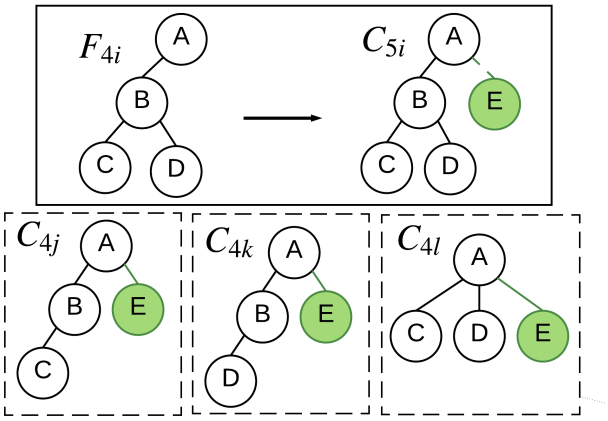

Figure 4.2: An example of subset pruning

The CPU implementation adds each individual frequent subtree into a hash table. Thus, each subtree check takes O(1) time, and since there can be k subtrees of length *k-1* and n candidates, it takes O(nk) time to perform the pruning check for the patterns in each iteration. In the AP implementation, many candidate itemsets are configured on the AP and checked against the input transaction in parallel. The time complexity of the AP solution is O(m), where m is the number of frequent candidates of the previous level. Because the support threshold here is 100%, we can remove the counter element of the FIM AP design [8], which is the main constraint of the AP board utilization. When the number of generated candidates is relatively small, the CPU solution beats the AP, because of the AP configuration overhead. However, when the number of candidates starts to grow, the AP implementation provides a faster solution. This kernel is very light-weight and does not require a pass of input trees (the input for this kernel is the set of *frequent-candidates* of the previous iteration), however, it prunes a large number of candidates in the early stage.

Table 4.3: An example on intersection pruning

$\boldsymbol{T_1}$	$\boldsymbol{F_{4i}}$	$\boldsymbol{F_{4j}}$	$\boldsymbol{F_{4k}}$	$\boldsymbol{F_{4l}}$	F_{4x}
$\boldsymbol{T_2}$	$\boldsymbol{F_{4i}}$	$\boldsymbol{F_{4k}}$	F_{4x}	F_{4x}	F_{4x}
$\boldsymbol{T_3}$	$\boldsymbol{F_{4k}}$	$\boldsymbol{F_{4l}}$	F_{4x}	F_{4x}	F_{4x}
$\boldsymbol{T_4}$	$\boldsymbol{F_{4i}}$	$\boldsymbol{F_{4j}}$	$\boldsymbol{F_{4l}}$	F_{4x}	F_{4x}

Intersection Pruning:

In order to pass this pruning stage, (1) all the subsets of a *(k+1)-candidate*, which are the members of a *k-frequent-candidates*, should happen in the same input tree, and (2) the number of joint occurrences must be more than the minimum support threshold. Let's assume C_{5i} from Fig. 4.2 has passed the subset pruning stage and all its subset has been frequent. Also, assume there is a database of four trees $\{T_1, T_2, T3, T4\}$, where F_{4i} occurs in $\{T_1, T_2, T_4\}$, F_{4j} occurs in $\{T_1, T_4\}$, F_{4k} occurs in $\{T_1, T_2, T_3\}$, and F_{4l} occurs in $\{T_1, T_4\}$. As we see, the set of $\{F_{4i}, F_{4j}, F_{4k}, F_{4l}\}$ (which are the subset of C_{5i}) jointly happens in only T_1. As a result, if the *Rminsup* is less than 25%, F_{5i} will pass the second stage, otherwise, it will be pruned.

Intersection pruning can directly map to FIM, where itemsets are the set of *(k+1)-candidates* and items in the itemsets are the set of *k-frequent-candidates* for each candidate. The number of input transactions is equal to the number of trees in the database and the size of each transaction is the number of frequent candidates contained in the transaction, which creates the AP input stream. If all the frequent candidates fit into the AP boards, one pass of the input stream checks the frequency of intersection pruning for all the candidates at the same time; otherwise, the automaton macros will be loaded with a new set of candidates, which requires another pass of the input stream. The CPU implementation uses a 1D array for each frequent candidate to list the set of trees in which it occurs. The size of the array is equal to the number of trees in the database.

Downward Pruning:

To further prune the search space, the downward pruning kernel simplifies tree representation to a sequence of root-to-child paths in order to check the ancestor-descendant relationships of a subtree in an input tree. Clearly, the original tree cannot be constructed using these paths, but it has some unique properties which help to identify a set of frequent subtrees and reduces computational complexity.

Downward string representation (DSR): It starts from the root of the tree and traverses all the paths from the root to the terminal children. The delimiter $'$,$'$ separates different paths and the delimiter $'$#$'$ represents the end of the downward representation string of an input tree. When mining ordered subtrees, it is important to traverse from the left-most path to the right-most path in order to preserve the order. For

example in Fig. 4.3, the vertical representation of subtree ST_2 is $AC, AB\#$. When delimiter $'\#'$, encoded at the end of subtree downward representation, matches to the input stream, the associated counter counts up by one and then, matching for the next tree starts from the root of the subtree.

Downward Pruning on the AP: For all the surviving *(k+1)-candidates* from the previous stage, the DSR will be created. These candidates can be interpreted as the candidate sequences in SPM, where the nodes in a path represent an itemset and paths create the itemsets. The DSR for the input tree is considered as the input stream for this kernel. We adopt the SPM-AP implementation in [7] for this stage.

DSR creates a sequential pattern of the tree structure, which preserves ancestor-descendant relationships and ignores the sibling information. Fig. 4.3 shows the DSR of an input tree and three example subtrees. According to the SPM definition, both $DSR\text{-}ST_1$ and $DSR\text{-}ST_3$ are the subsequences of $DSR\text{-}T_0$, so they survive the downward pruning and will be checked further at the next pruning kernel. ST_3 is not a true subtree of T_0 and connectivity kernel, a complementary pruning strategy, will prune it in the next stage. $DSR\text{-}ST_2$ is not a subsequence of $DSR\text{-}T_0$ and ST_2 can be safely pruned from the set of candidate subtrees.

Downward pruning ensures that, for all the subtrees candidates with degree no more than one (we call them line-shaped candidates), the final decision regarding their frequency will be made at this stage and no false positive candidate will survive from this kernel. This is particularly true because line-shaped candidates are equivalent to an itemset in SPM, where no branching information is required. The quality of downward pruning directly depends on the topology of the input trees. Deeper trees will benefit more from the downward pruning.

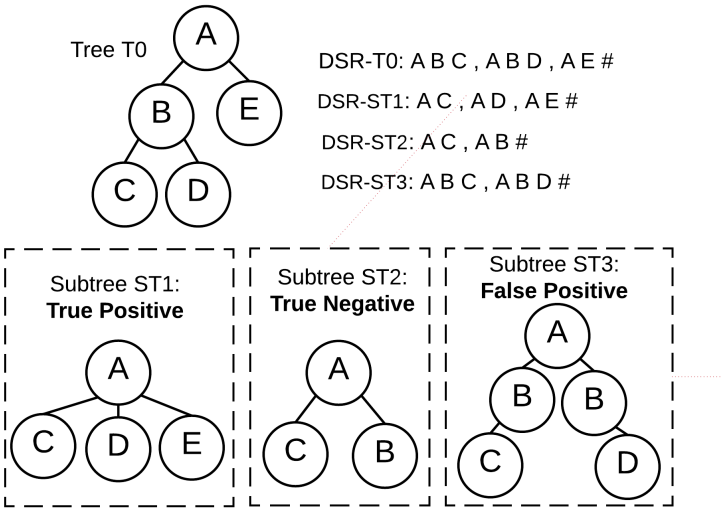

Figure 4.3: An example of downward pruning

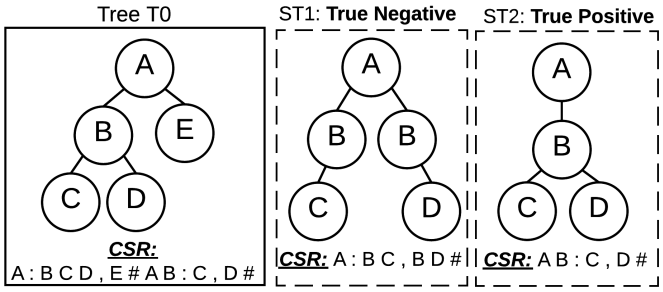

Figure 4.4: An example of connectivity pruning

Connectivity Pruning:

Connectivity pruning addresses situations when the downward string representation generates two itemsets out of one node, which allows some false positives to survive downward pruning. Connectivity pruning finds a mapping of the subtree *root-path* to the input tree and then looks for the child sequences of the last node in the root-path from left to right in the tree. The root-path of a subtree is the path from the root to the first node with degree more than one. For example, the root-path of ST_2 in Fig. 4.4 is AB.

Connectivity string representation (CSR): CSR of a subtree consists of the root-path followed by the delimiter ':', and then, the pre-order representation of the children from the left-most path to the right-most path separated by the delimiter ','. For example in Fig. 4.4, the CSR of ST_1 is $A : BC, BD\#$, where the root-path is A and the pre-order representations of its children are BC and BD, which are separated by the ','. In order to detect the subtree in an input tree, the CSR of the input tree should be extended by all the paths from the root to all the node with degree more that two. Take the input tree T_0 as an instance, where first, A is considered as the root-path and is followed by the left-side children (BCD) and the right-side child (E), and second, AB is considered as the root path followed by the B's children. Delimiter '#' separates root path sets in the trees and subtree inclusion checking starts from the subtree root after '#' appears in the input stream.

Connectivity Pruning on the AP: This kernel can directly map to the SPM, where the root path and children are the itemsets and the nodes are the items. In SPM, the order between the itemsets matters while the order between the items does not matter. However, having a pre-defined order of the items helps simplify the automata structure [7]. In sequences generated by CSR, both items and itemsets have a pre-defined ordering, which means that the CSR can be easily map to the SPM automata structures. Connectivity pruning does not cause any false negatives, because it just relaxes necessary tree structure properties in order to check subtree inclusion.

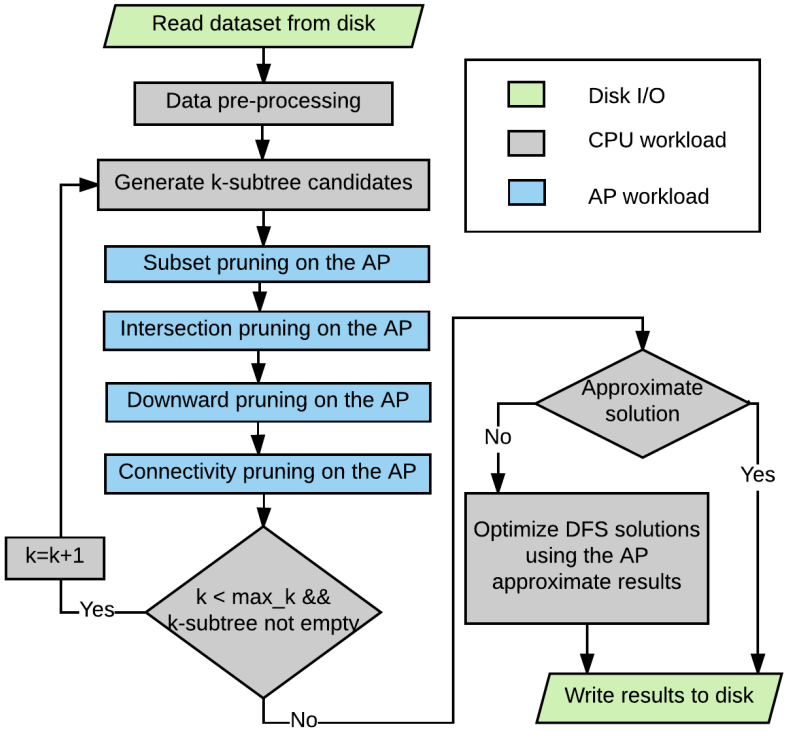

Figure 4.5: The workflow of the AP-accelerated FTM

4.4.3 Pruning Corollaries

3-candidates can only have two different topologies; (1) a root and two children connect to the root (triangle-shaped), and (2) a root with one child and one grandchild (line-shaped). As discussed before, lined-shape patterns will be properly pruned in the downward stage. Connectivity pruning also perfectly trims triangle-shape ones. This is because the root path has just one node, which is the root itself and the left and right child are the only node and do not have hierarchical structure, and they only need to appear (in order) in two different branches of the equivalent tree node to the subtree root. These make all the surviving *3-candidates* in the final set to be true *3-frequent-candidates*. This property is very useful because more precise pruning in early iterations will greatly reduce the chance of getting false positives in the later stages.

4.4.4 Program Infrastructure

Fig. 4.5 shows the complete workflow of the AP-accelerated FTM proposed in this paper. The input database is in horizontal, string-encoded format (the horizontal format is the pre-order traversal of trees, including backtracking information). The following describes the data pre-processing steps:

Computing 1-frequent-candidates (F_1) and 2-frequent-candidates (F_2): To compute F_1, for each item (node label) in the tree, its count in a 1D array will be incremented, where the total time for each tree with size n is $O(n)$. Other database statistics, such as the maximum number of labels and number of trees, is calculated as well. F_2 counting is done using a 2D integer array of size $F_1 \times F_1$ and the total time is $O(n^2)$ per tree.

Recoding the input trees: Depending on the number of frequent items, the item can be encoded by 8-bit or 16-bit symbols. Different encoding schemes lead to different automaton designs and capacity of patterns for each pruning stage.

Making input streams: We create downward and connectivity string representation of the input trees according to the recoded items and keep them in the memory.

After pre-processing and generating tree candidates, the corresponding AP input stream will be generated for each pruning stage. Then, the appropriate pre-compiled template macro of automaton structure for FIM or SPM pattern is selected according to k (size of itemset or sequence candidate) and is configured on the AP board. The candidates are generated on the CPU and are filled into the selected automaton template macro. The input data formulated in pre-processing is then streamed into the AP board for counting. While there are *k-candidates* left to be processed on the AP, the AP computation (symbol replacement and matching) and the AP input generation of the next-level pruning kernel can be done in parallel. Fortunately, the latency of symbol replacement could be significantly reduced in future generations of the AP (because symbol replacement is simply a series of DRAM writes), which would improve the overall performance greatly. At the end of connectivity pruning stage, either k has reached the maximum size or *k-frequent-candidates* set is empty, we have the approximate solution, which is a set of potentially frequent candidates. Depending on the final application, the approximate results can either be directly used with no further final pruning or can be considered as the ground candidate set for an exact FTM solution. We will later show how TreeMinerD [74] can be used to provide an exact solution over the AP results.

4.5 FTM GPU Implementation

Despite the DFS strategies, where memory becomes a limiting factor for performance and scalability, especially in large datasets and lower support thresholds, BFS solutions do not require a large memory footprint, as they do not project the dataset into memory. However, they require a pass of the dataset in each iteration. Thus, to implement FTM on the GPU platform, we chose to adopt the BFS-based candidate-generation and enumeration strategy because (1) the solution will not be bound by the finite GPU global memory, and (2) it exposes many ready-to-process candidate subtrees in order to fully utilize the GPU cores and ultimately, reduces the overhead of database scanning.

FTM-GPU: After recoding labels, the whole dataset is transferred to the GPU global memory. Then, the algorithm iterates over three steps: (1) generating *(k + 1)-subtree* candidates from the frequent *k-subtrees* on the CPU, (2) pruning the candidates using subset pruning on the CPU, and (3) identifying the frequent *(k + 1)-subtrees* on the GPU. For the GPU computation, we convert both input trees and subtree candidates to one-dimensional array. In the CUDA kernel function, each thread is responsible for matching one tree in the input dataset to a candidate. We explore two memory region targets for the subtree candidates. In the first approach, we transfer all the candidates to constant memory (in iterations, if candidate array is larger than constant memory size) and all the threads start matching one candidate to their bound trees. Alternatively, we transfer the candidates to global memory and at each iteration, take just one to shared memory for matching. The constant memory implementation provides a faster solution when the trees in the dataset are similar in terms of size and node labels, otherwise, the shared memory approach shows a better result.

To improve the performance of FTM-GPU, we sort the trees in the database according to the tree size. This sorting tries to provide each warp with a batch of trees of roughly the same size. This greatly helps reduce branch divergence and lessen synchronization time. Once the matching and counting phase is done for all the *(k+1)-candidates*, the results are transferred back to the CPU for the next-level candidate-generation. FTM-GPU is capable of counting both distinct occurrences of subtrees and weighted support. *To the best of our knowledge, this is the first implementation of frequent subtree mining on the GPU platform.*

4.6 Experimental Results

The number of patterns that can be placed into the board, and the number of candidates that must be checked in each stage, assuming a 32-chip Micron D480 AP board, determines how many passes through the input are required for each pruning kernel, and the input processing rate is fixed at 133 MB/s, which allows a simple calculation to determine the total time on the AP (see hardware parameters in [28]). All the automata designs are selected from the 16-bit encoding for simplicity, so there is no need for reconfiguration when the number of labels is more than 256. In each step of pruning, an appropriate FIM or SPM corresponding to the candidate size will be selected and configured on the AP.

4.6.1 Comparison with Other Implementations

We compare the performance and accuracy of the proposed AP multistage pruning for FTM (FTM-AP) versus (1) a BFS-based GPU implementation of FTM (FTM-GPU), (2) a multi-core implementation using pthread (TRIPS-12C) [75], (3) a single-core DFS-based implementation capable of weighted counting (TRIPS)

[80], (4) a single-core DFS-based implementation which counts distinct occurrences of a pattern (TreeMinerD) [74], and a single-core level-wise BFS approach (PatternMatcher) [77].

We consider PatternMatcher in our comparison because it is the only method that does not fail on challenging datasets in lower support thresholds. Despite being very slow, it gives us a baseline for calculating both performance and accuracy for the proposed AP solution. Similarly to TreeMinerD, the FTM-AP solution is designed to only enumerate distinct occurrences of a pattern, thus providing a very fast solution in comparison with TreeMiner [74]. Therefore, we do not compare FTM-AP with TreeMiner. Moreover, TRIPS and TIDES [80] claim that they are orders of magnitude better than TreeMiner. In the same paper, they show that XSpanner is worse than TreeMiner, so we do not compare with XSpanner either.

FIM and SPM implementations on the AP are much faster than their GPU solutions, especially for large datasets [8] [7]. In FTM-AP, all the kernels are mapped to either FIM or SPM, and we can conclude that FTM-AP will outperform the GPU implementation of pruning kernels. Moreover, GPU implementations of subsequence inclusion checking in a sequence and subtree inclusion checking in a tree have almost similar complexity and synchronization overhead. Thus, we predict that exact FTM solution on the GPU (FTM-GPU) will outperform the GPU implementation of pruning kernels (inexact-FTM-GPU), because inexact-FTM-GPU requires at least twice as many subsequence inclusion checking operation as FTM-GPU requires subtree inclusion checking.

4.6.2 Platform and Parameters

All of the above implementations are tested using the following hardware:

- CPU: Intel CPU i7-5820K (6 cores, 3.30GHz), memory: 32GB, 2.133 GHz
- GPU: Nvidia Kepler K80C, 560 MHz clock, 4992 CUDA cores, 24GB global memory
- AP: D480 board, 133 MHz, 32 AP chips (simulation)

Furthermore, we test CPU solutions in a large-memory machine with 512GB of memory later in Section 4.6.7. For each benchmark, we compare the performance of the above implementations over a range of relative minimum support (*Rminsup*) values. To observe the behavior of different implementations and finish all our experiments in a reasonable amount of time, we select *Rminsup* numbers that produce computation times up to one day.

4.6.3 Datasets

We evaluate the proposed algorithm on four different datasets, two real-world (CSLOGS [1] and TREEBANK [2]), and two synthetically generated by the tree generation program provided by Zaki[1] (T1M and T2M). Table 5.1 shows the details of the datasets. CSLOGS consists of user browsing subtrees of the CS department web site at the Rensselaer Polytechnic Institute. TREEBANK is widely used in computational linguistics and consists of XML documents. It provides a syntactic structure of the English text and uses part-of-speech tags to represent the hierarchical structure of the sentences. T1M and T2M are generated based on a mother tree with the maximal depth and fan-out of 10. The total number of nodes in T1M and T2M are 1,000,000 and 100,000, respectively. The datasets are then generated by creating subtrees of the mother tree. The synthetic tree generator provides a preorder-like representation, while TRIPS and TRIPS-12C work with the Prüfer sequence and postorder tree representation. Thus, we convert the datasets to their compatible input format offline and do not consider it in the time calculation.

Table 4.4: Datasets

Name	#Trees	Ave_Node	SD_Node	#Items	Size (MB)
T1M	1,000,000	5.5	6.2	500	49.3
T2M	2,000,000	2.95	3.3	100	60.1
CSLOGS	59691	12.94	22.47	13361	6.3
TREEBANK	52581	68.03	32.46	1387266	27.3

Ave_Node = Average number of nodes per tree
SD_Node = Standard deviation of number of nodes per tree
#Items = Label set size

4.6.4 AP-FTM Breakdown and Speedup Analysis

We choose TREEBANK dataset to study the pruning efficiency of each kernel and compare the performance of the CPU implementation and the AP-FTM for each of them. We also compare the scalability and efficiency of the kernel methods with the counting stage of PatternMatcher. TREEBANK is the most challenging dataset, because it consists of very wide and large trees (the largest tree has 684 nodes) and it has a large number of items and relatively high standard deviation of tree size, which makes it difficult for the CPU solutions to process. Excluding PatternMatcher, other solutions either crash or quickly run out of memory when decreasing the support threshold.

We have implemented all four pruning kernels on the CPU (in C++) in order to isolate the performance difference of the AP vs. CPU for the same work and highlight the AP architectural contribution. Fig. 4.6

[1] http://www.cs.rpi.edu/~zaki/software/
[2] http://www.cs.washington.edu/research/xmldatasets/

shows that subset, intersection, downward, and connectivity kernels achieve up to 163×, 19×, 3144×, and 2635× speedups over their counterpart CPU implementations, while they prune at least 80%, 0.5%, 3.5%, and 4.8% of the total generated candidates in TREEBANK, when ranging *Rminsup* from 0.9 to 0.3 (Fig. 4.7). Due to the AP configuration time overhead, *subset_pruning_CPU* is faster than *subset_pruning_AP* at higher support thresholds. However, when *Rminsup* decreases and more candidates are generated, searching the larger dictionary of a frequent subtree on the CPU takes longer, while the AP solution is almost constant. However, the subset kernel has a very small effect in total execution time, yet at the same time, it has a major contribution in pruning the candidates. Although intersection pruning has the lowest pruning contribution among others, it has the highest computation time on the AP, because the necessary input stream for this kernel is very large (due to the repetition of frequent candidates in different input trees). In total, the AP kernels show up to 215× speedup over the CPU pruning kernels, which implies the AP architectural contribution (black bars in Fig. 4.6).

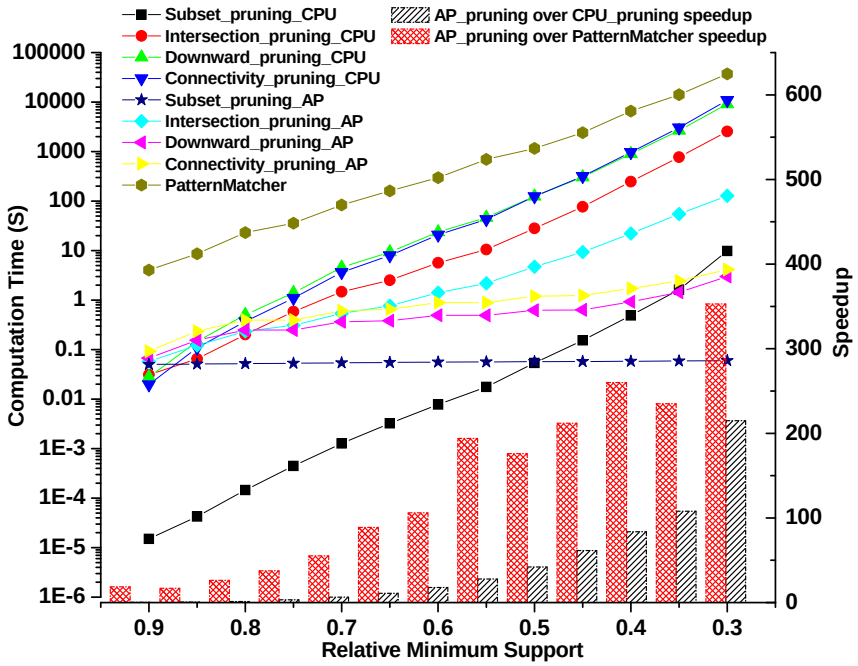

Figure 4.6: Kernels' execution time and speedup

The subset kernel is more effective in lower *Rminsup*, where larger candidates are introduced and survive. This is because larger candidates have more frequent subsets, which increases the probability of pruning false positives. On the other hand, downward and connectivity kernels are more efficient on the smaller candidates because the effect of relaxing tree constraints is less destructive on them. This observation can be clearly seen in Fig. 4.7, where by decreasing the *Rminsup* (which means the population of the larger candidates relative to the smaller ones grows), the contribution of subset pruning increases, whereas the contribution of

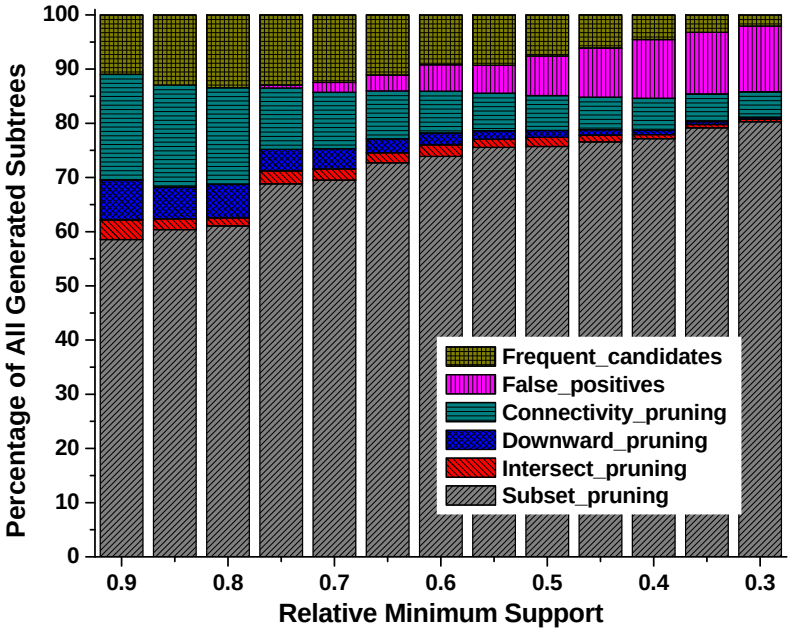

Figure 4.7: Generated subtrees' breakdown

the other three kernels decreases.

We also compare the AP-FTM solution with PatternMatcher in order to show the trade-off between accuracy and execution time, and also study the algorithmic/heuristic contribution of our pruning kernels (the ratio of red to black bars in Fig. 4.6). In total, at least 86% of the generated candidates are pruned using the pruning kernel within less than 105 seconds for the lowest support threshold, where the PatternMatcher takes more than 10 hours to find the exact frequent candidates (about 353× speedup considering pre-processing time). Note that the pruning portions and timing are calculated just for the candidates of size three and more, and we do not consider the number of candidates for *1-frequent-candidates* and *2-frequent-candidates*, as they will be easily detected either on the AP or on the CPU. In order to further study the behavior of the pruning kernels, we test the effects of taking the intersection kernel out of the pruning framework, because as we have already observed, it has the least pruning efficiency and largest computational time. The results show that the AP achieves up to 1530× and 2190× speedup over the CPU-based pruning kernels and PatternMatcher, respectively. However, the AP worst case accuracy decreases from 86% to 83%. Therefore, having more sophisticated kernels can improve the accuracy and depend on the target constraints, and the user can make the trade-off between kernel selection and desirable speedup.

4.6.5 AP-FTM vs. Other FTM Algorithms

Fig 4.8 - 4.11 represent performance comparison (left vertical axis) among single-core CPU implementations (TreeMinerD, PatternMatcher, TRIPS), a multi-core (TRIPS-12C) approach, a GPU solution (FTM-GPU), and our proposed method (FTM-AP) for mining distinct occurrences of embedded subtrees in four databases of ordered labeled trees. All methods are end-to-end solutions and apart from FTM-AP, have an accuracy of 100% for the final frequent set. The right vertical axes in the graphs represent the percentage of the false positives in the output of FTM-AP. The main goal of these graphs is to compare the trade-off between the speed and accuracy of the AP solution versus the existing FTM implementations.

Most existing state-of-the-art tools have difficulty with larger inputs and smaller support thresholds and fail due to scalability limits. Common limits include long execution time, insufficient system memory, limitations in internal data-structures, or crashing/reporting an error due to their design not anticipating a large input. TRIPS and TRIPS-12C either crash or get struck for unknown reasons when decreasing the *Rminsup*. For example, in TREEBANK, TRIPS-12C breaks at *Rminsup* = 0.65 and TRIPS stucks at *Rminsup* = 0.5. TRIPS-12C shows an unstable behavior in T1M and T2M, and from our experience, changing the number of running threads and turning hyper-threading off do not make a difference.

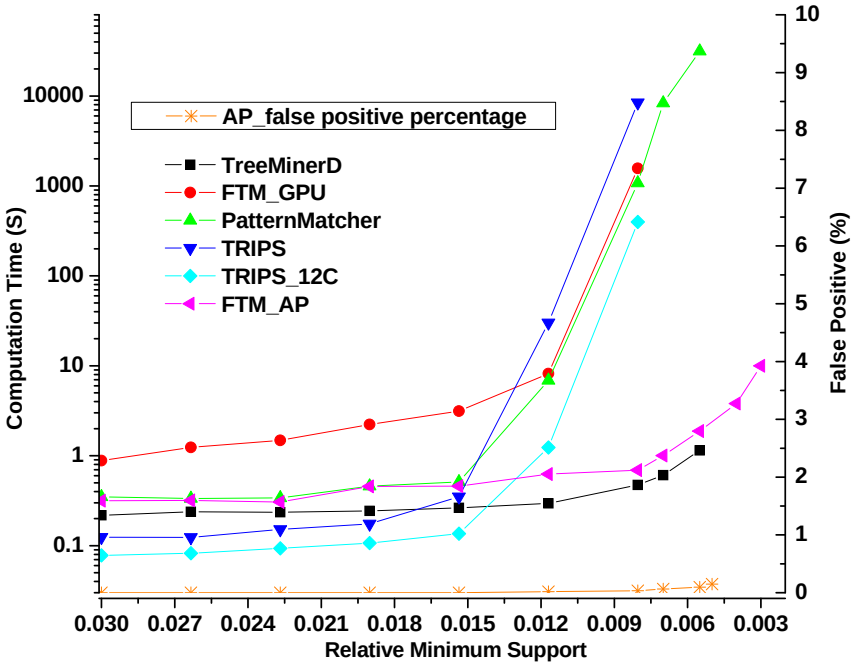

Figure 4.8: Performance comparison on CSLOGS

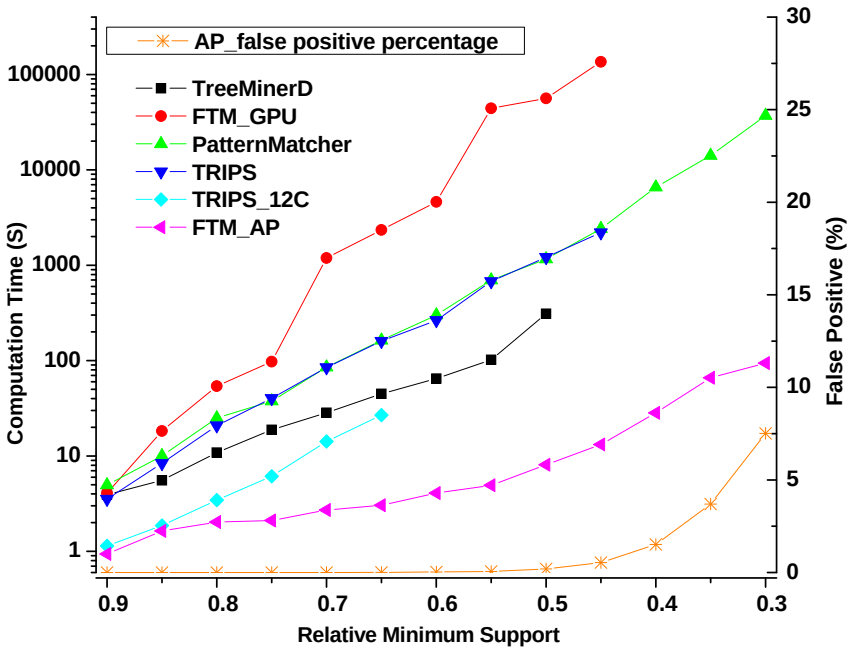

Figure 4.9: Performance comparison on TREEBANK

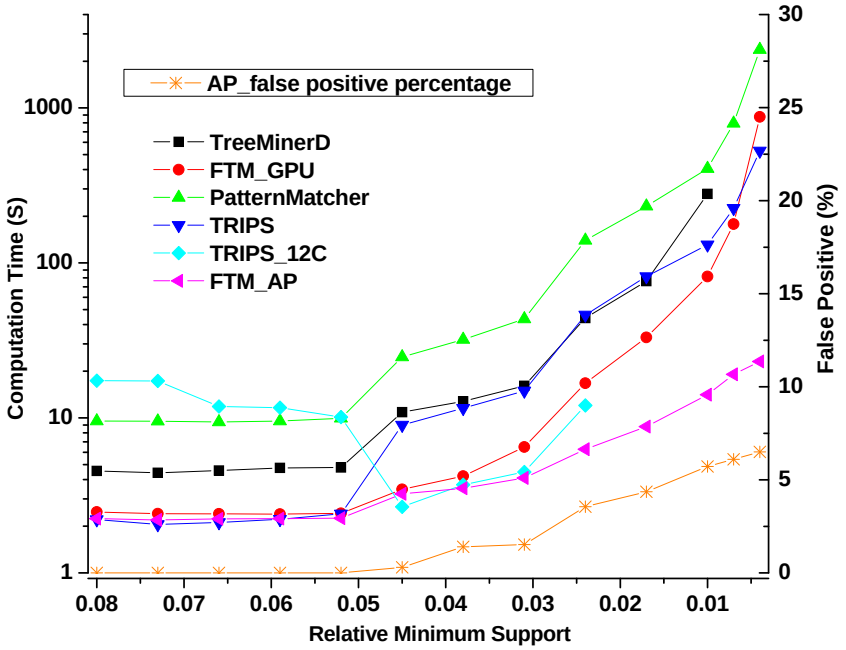

Figure 4.10: Performance comparison on T1M

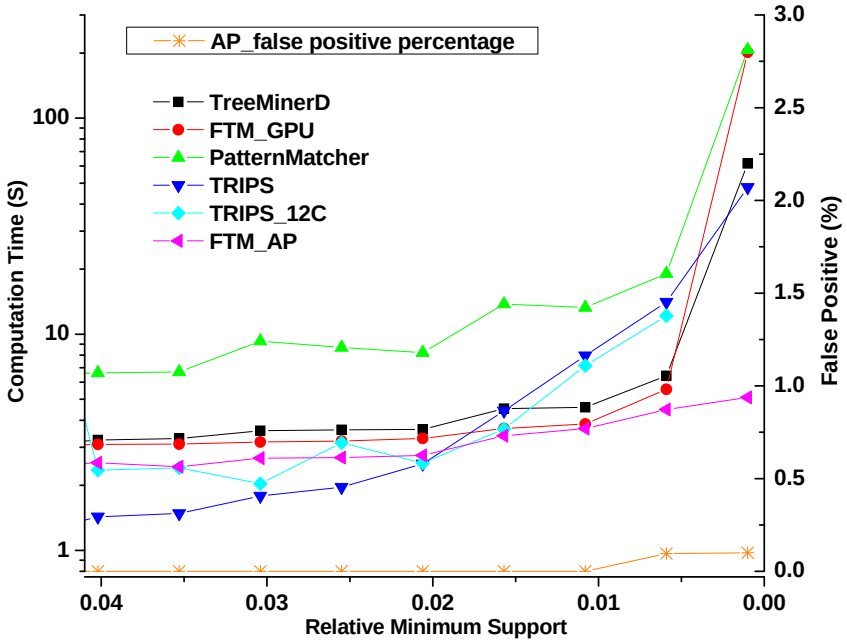

Figure 4.11: Performance comparison on T2M

TreeMinerD is the fastest accurate solution in real-world datasets for smaller thresholds. In CSLOGS (Fig. 4.8), TreeMinerD is even faster than the FTM-AP solution, however, it runs out of memory when $Rminsup$<0.005 (we will discuss the memory usage of TreeMinerD in the next subsection). PatternMatcher requires the least amount of memory among other solutions, but its execution time takes more than 10 hours at $Rminsup$<0.006 and $Rminsup$<0.35 in CSLOGS and TREEBANK, respectively.

Database statistics such as average and standard deviation of the number of nodes per tree, the number of items, and the number of trees directly affect the performance of the FTM-GPU. In Fig. 4.8 and 4.9, FTM-GPU shows almost the worst performance results among others. This is because SD_Node and $\#Items$ in TREEBANK and CSLOGS are relatively high. Higher SD_Node implies uneven distribution of trees with different sizes in the database, and causes the synchronization time between the thread in a warp to increase. Higher $\#Items$ increases the chance of thread divergence in a warp, because the possibility of checking the subtree node against different labels in the input trees of the same warp increases. In CSLOGS at $Rminsup \leq 0.008$ and TREEBANK at $Rminsup \leq 0.45$, the FTM-GPU takes more than 10 hours to run. T1M and T2M in Fig. 4.10 and 4.11 show that FTM-GPU has much better relative performance among accurate solutions as the SD_Node and $\#Items$ are lower (Table 5.1). Overall, the FTM-GPU results show that the GPU platform does not provide a reliable and scalable solution for the FTM.

In Fig. 4.8, for $Rminsup \geq 0.008$, the execution time of TreeMinerD and FTM-AP are almost the same, which is less than a second. In the range of $0.008 < Rminsup \leq 0.006$, PatternMatcher is the only running

accurate solution, which takes 31,456 sec when $Rminsup = 0.006$. Eventually, for $Rminsup \leq 0.006$, FTM-AP continues to be the only reliable running solution, and we are not able to calculate the accuracy of the AP-FTM, as there is no exact solution running in this range. The maximum accuracy reduction for FTM-AP in CSLOGS is 0.09%. For T1M and T2M, AP-FTM beats TRIPS, which is the fastest solution, by factors of 22× and 9.2×, while losing at most 6.5% and 0.1% accuracy, respectively. Overall, FTM-AP, with at most 7.5% false positives, beats PatternMatcher, the feasible and exact CPU solution, by a factor of 394×. The low memory requirement and huge speedup achieved by the AP-FTM makes it a scalable and reliable solution with a final application tolerance of a few percentage points for false positives. Another advantage of the AP is that it gives consistently good performance, while the performance of other techniques varies based on the database characteristics.

4.6.6 Performance Scaling with Data Size and Support Threshold

In this subsection, we study the scaling of performance as a function of input data sizes and minimum support threshold. In order to generate synthetic datasets, we adopt the parameters used to generate T1M (Table 5.1) and increase the number of trees from 10^6 (49MB) to 2×10^6 (99MB), 4×10^6 (196MB), 8×10^6 (386MB), and 16×10^6 (770MB), while preserving the other parameters. Fig. 4.12 represents the performance of TreeMinerD, FTM-GPU, PatternMatcher, TRIPS, and FTM-AP for these five synthetically generated dataset using two relative minimum supports. FTM-AP beats FTM-GPU by a factor of 2.6× at $Rminsup = 0.04$ and 3.9× at $Rminsup = 0.02$, and PatternMatcher by a factor of 3.3× at $Rminsup = 0.04$ and 45.1× at $Rminsup = 0.02$, while losing at most 1% accuracy. TreeMinerD and TRIPS do not provide a scalable solution, because they both run out of memory when increasing the input size and when decreasing the minimum support threshold. The results show that the FTM-AP always has the lowest execution time and its performance advantage grows when $Rminsup$ decreases and input size increases. We are evaluating FTM performance for one node. Sufficiently large CPU/GPU clusters can handle larger FTM problems and run them faster than a single AP, but a cluster of APs would be even faster.

4.6.7 Exact Solution on the FTM-AP

The output of the AP pruning kernels is a set of potentially frequent candidates that preserve a subset of tree topological and label attributes. Depending on the target application, one can directly use the AP output as the set of frequent subtrees, especially in classification tasks, where the applications are able to recover from the introduced false positives. Frequent subtrees are used in many natural language processing (NLP) tasks, because tree structures can capture and represent the complex relations and dependencies of a natural

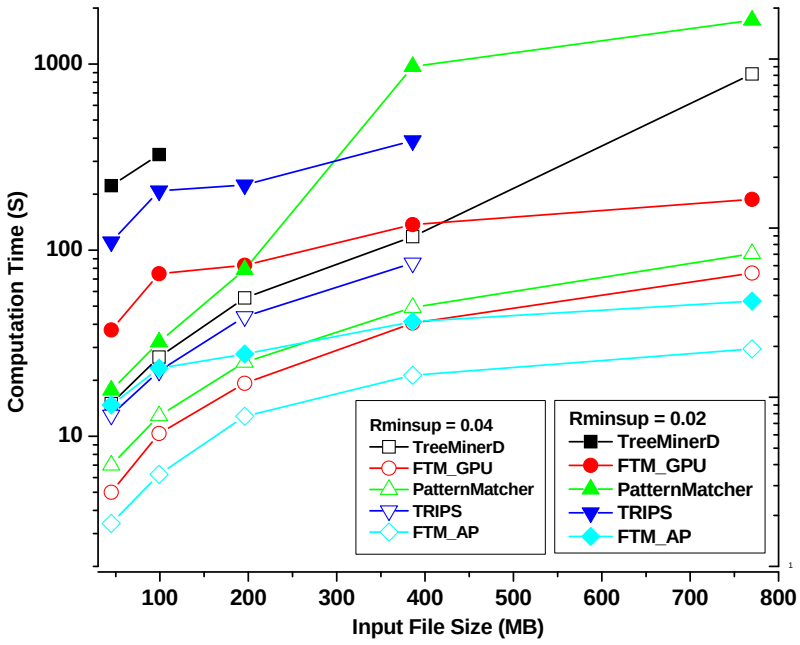

Figure 4.12: Performance scaling with input data-size and Rminsup

language. Agarwal et al. in [73] demonstrated how combining subtree features with sequential patterns and bag-of-words can increase sentiment analysis accuracy. Since the AP results are already examined for sequential properties in downward pruning, and all the false positives in the final candidate set can be translated as sequential pattern features (albeit with low support), it suggests that the AP final results can be directly used for those tasks without potentially affecting the final accuracy, while achieving a huge speedup.

On the other hand, having an exact set of frequent subtrees is a must for some applications. In order to prune the false positives from the AP output, we propose APHybrid-FTM, to combine the AP with TreeMinerD, where the AP solution can help to reduce the memory requirement and increase the speed of the TreeMinerD approach while maintaining its 100% accuracy. As mentioned before, TreeMinerD implements a DFS-based algorithm, and a *(k+1)-candidate* is generated by combining two *k-candidates* on the same equivalent-class, under some circumstances [74]. We store the set of potential frequent subtrees in a dictionary ($\sqcup$) and check whether the *(k+1)-candidates*, generated by the TreeMinerD candidate-generation step, exist in $\sqcup$. If it is a hit, TreeMinerD continues the matching and counting stage (as discussed in Section 4.4.3, the set of *3-candidates* and line-shaped candidates are 100% accurate and do not need to be checked for frequency), otherwise, the candidate is infrequent, which avoids the unnecessary enumeration step, and the algorithm continues to generate the next candidate. The AP-FTM framework greatly helps TreeMinerD to (1) reduce its execution time, and (2) alleviate its memory footprint because many infrequent candidates in TreeMinerD will be pruned early in the candidate-generation step and their occurrences (embedding information in the

database) do not need to be stored in memory.

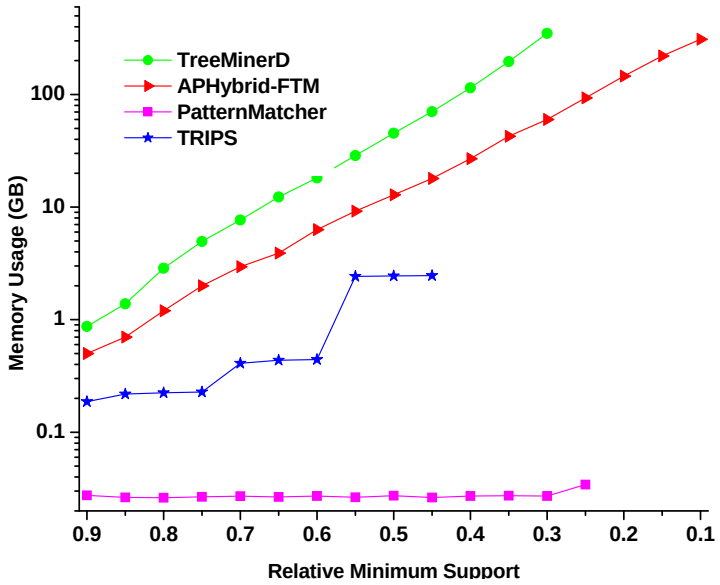

Figure 4.13: Exact solutions memory usage - TREEBANK

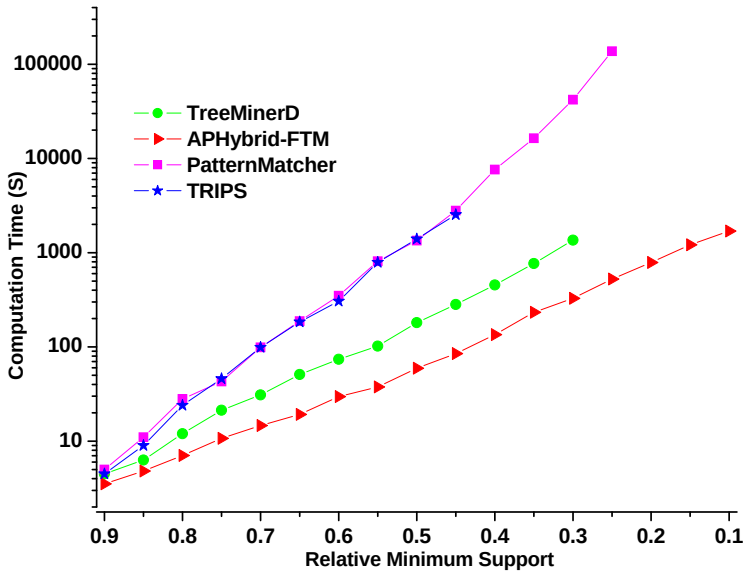

Figure 4.14: Exact solution execution time - TREEBANK

In order to analyze APHybrid-FTM performance and study the effect of memory usage on the FTM scalability, we run TreeMinerD, APHybrid-FTM, PatternMatcher, and TRIPS on a node[3] with 512GB memory for TREEBANK. Fig. 4.13 represents the required maximum memory size and Fig. 4.14 shows the execution time of these methods on a log scale. The speed and memory trade-off among TreeMinerD, PatternMatcher,

[3]Intel(R) Xeon(R) CPU E5-2670 (24 cores, 2.30GHz), memory: 512GB, 2.133 GHz

and TRIPS on the CPU can be easily observed by looking at these two graphs, where TreeMinerD is the fastest tool among these three and demands the largest memory footprint, whereas PatternMatcher, with the lowest performance, requires the smallest memory capacity (less than 100MB). However, APHybrid-FTM is the best. It alleviates memory usage of TreeMinerD up to 5.8× and reduces its execution time up to 4.14 at $Rminsup = 0.3$. For $Rminsup \leq 0.3$, TreeMinerD runs out of memory, while APHybrid-FTM continues. Furthermore, APHybrid-FTM performs 262× better than PatternMatcher at $Rminsup = 0.25$ (it takes more than a day for $Rminsup < 0.25$) and 30× better than TRIPS at $Rminsup = 0.45$ (lowest working threshold for TRIPS).

In summary, APHybrid-FTM provides the fastest exact solution (Fig. 4.14), which in turn extends the scalability of TreeMinerD and its advantages increase at lower support thresholds and larger databases. Overall, the proposed pruning approach can be adopted as a general strategy to accelerate complex and/or memory-intensive pattern-mining problems.

4.7 Conclusions and Future Work

We develop FTM-AP, a multi-stage pruning strategy on the AP, to reduce the candidate search space of the frequent subtree mining problem in a very short amount of time, providing a fast and scalable solution at the cost of a small reduction in accuracy. FTM-AP achieves up to 394× speedup with at most 7.5% false positives over PatternMatcher, a feasible and exact CPU solution. For problems requiring an exact solution, we use the output of FTM-AP as the candidate screen for TreeMinerD, a DFS-based exact solution, in order to remove the false-positive candidates, limit the memory requirements, and achieve up to 262× speedup. The benefits the AP provides for FTM increase with larger datasets and lower support thresholds. The pruning framework on the heterogeneous architecture (CPU and the AP) can also potentially be adopted to extend the scalability of the other subtree types and graph mining problems, an interesting direction for future work.

Additional performance improvements could be achieved with hardware support to minimize symbol replacement latency and maximize capacity of resources on the AP, as well as better support for push-down automata capabilities. The proposed pruning kernels are capable of running in a pipeline system, assuming four AP boards are available, and this also allows scaling of larger problems to cluster- or datacenter-scale resources, another interesting direction for future work.

Chapter 5

A Scalable In-SRAM Architecture for Pushdown Automata

Processing of tree-structured or recursively-nested data is intrinsic to many computational applications. Data serialization formats such as XML and JSON are inherently nested (with opening and closing tags or braces, respectively), and structures in programming languages, such as arithmetic expressions, form trees of operations. Further, the grammatical structure of English text is tree-like in nature [30]. Reconstructing and validating tree-like data is often referred to as *parsing*.

Studies on data processing and analytics in industry demonstrate both increased rates of data collection and also increased demand for real-time analyses [81, 82]. Therefore, scalable and high-performance techniques for parsing and processing data are needed to keep up with industrial demand. Unfortunately, parsing is an extremely challenging task to accelerate and falls within the "thirteenth dwarf" in the Berkeley parallel computation taxonomy [83]. Software parsing solutions often exhibit irregular data access patterns and branch mispredictions, resulting in poor performance. Custom accelerators exist for particular parsing applications (e.g., for parsing XML [84]), but do not generalize to multiple important problems.

We observe that *deterministic pushdown automata* (DPDA) provide a general-purpose computational model for processing tree-structured data. Pushdown automata extend basic finite automata with a stack. State transitions are determined by both the next input symbol and also the top of stack value. Determinism precludes stack divergence (i.e., simultaneous transitions never result in different stack values) and admits efficient hardware implementation. While somewhat restrictive, we demonstrate that unlike NFAs and DFAs, DPDAs are powerful enough to mine for frequent subtrees within a dataset.

In this work, we present ASPEN, the Accelerated in-SRAM Pushdown ENgine, a realization of deterministic

pushdown automata in Last Level Cache (LLC). Our design is based on the insight that much of the DPDA processing can be architected as LLC SRAM array lookups without involving the CPU. By performing DPDA computation in-cache, ASPEN avoids conventional CPU overheads such as random memory accesses and branch mispredictions.

ASPEN supports processing of hundreds of different DPDAs in parallel as any number of LLC SRAM arrays can be re-purposed for DPDA processing. This feature is critical for applications such as frequent subtree mining which require parsing several trees in parallel.

ASPEN is inspired from memory-centric architectures, such as Micron's D480 Automata Processor (AP) [85] and Subramaniyan et al.'s Cache Automaton (CA) [86] which leverage the massive bit-level parallelism of memory arrays (DRAM and L3 cache, respectively) to perform tens of thousands of input comparisons and state transitions in parallel. While the one-cycle-per-byte processing performance of the AP and CA is enticing, the finite automata processing model of computation is too restrictive to directly support parsing of tree-like data. Thus we develop ASPEN, a memory-centric acceleration framework for DPDA processing.

5.1 Homogeneous Deterministic Pushdown Automata

For hardware efficiency, we extend the definition of homogeneous finite automata to DPDA. In a homogeneous DPDA (hDPDA), all transitions to a state occur on the same input character, stack comparison, and stack operation. Concretely, for any $q, q', p, p' \in Q$, $\sigma, \sigma' \in \Sigma$, $\gamma, \gamma' \in \Gamma$, and op, op' that are operations on the stack, if $\delta(q, \sigma, \gamma) = (p, op)$ and $\delta(q', \sigma', \gamma') = (p', op')$, then

$$p = p' \Rightarrow \sigma = \sigma' \wedge \gamma = \gamma' \wedge op = op'.$$

This restriction on the transitions function does not limit computational power, but may increase the number of states needed to represent a particular computation.

Claim 1. *Given any DPDA $A = (Q, \Sigma, \Gamma, \delta, S, F)$, the number of states in an equivalent hDPDA is bounded by $O(|\Sigma||Q|^2)$.*

Proof. We consider the worst case: A is fully-connected with $|\Sigma| \cdot |Q|$ incident edges to each state and each of these incoming edges performs a different set of input/stack matches and stack operations. Therefore, we must duplicate each node $|\Sigma|(|Q| - 1)$ times to ensure the homogeneity property. For any node $q \in Q$, we add $|\Sigma| \cdot |Q|$ copies of q to the equivalent hDPDA, one node for each of the different input/stack operations on incident edges. Therefore, there are at most $|\Sigma| \cdot |Q| \cdot |Q| = |\Sigma||Q|^2$ vertices in the equivalent hDPDA. □

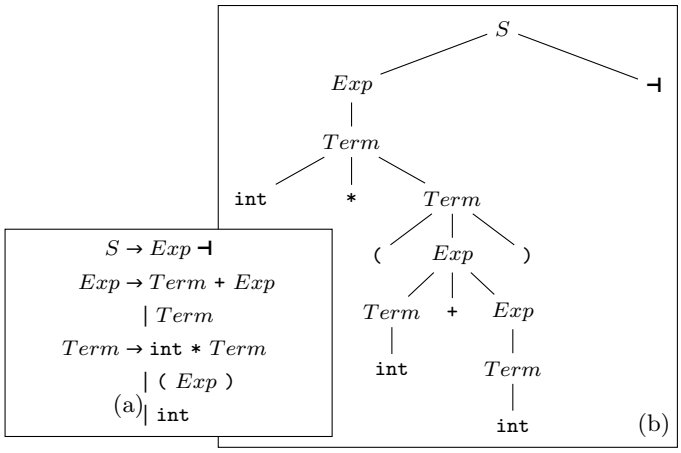

Figure 5.1: An example CFG (a) and parse tree (b). The grammar represents a subset of arithmetic expressions. We use ⊣ to signify the endmarker for a given token stream, which is needed for transformation to a DPDA. The parse tree given in (b) is for the expression $3 * (4 + 5)$. Note that integer numbers are transformed to `int` tokens prior to deriving the parse tree.

In practice, DPDAs tend not to be fully-connected and have a fixed alphabet, resulting in less than quadratic growth. Even in the worst case, hDPDAs do not significantly increase the number of states (cf. the exponential NFA to DFA transformation).

5.2 Tree Mining to Deterministic Pushdown Automata

In this section, we describe context-free grammars and our method to compile such grammars to pushdown automata.

5.2.1 Context-Free Grammars

While DPDAs provide a functional definition of computation, it can often be helpful to use a higher-level representation that generates the underlying machine. Just as regular expressions can be used to generate finite automata, *context-free grammars* (CFGs) can be used to generate pushdown automata. We briefly review relevant properties of these grammars (the interested reader is referred to references such as [39, 87, 88, 89] for additional details).

CFGs allow for the definition of recursive, tree-like structures using a collection of *substitution rules* or *productions*. A production defines how a symbol in the input may be legally rewritten as another sequence of symbols (i.e., the right-hand side of a production may be *substituted* for the symbol given in the left-hand side). Symbols that appear on the left-hand side of productions are referred to as *non-terminals* while symbols that do not are referred to as *terminals*. The *language* of a CFG is the set of all strings produced by

Input character	Transition function
$\tau_0 \in \lambda$	$(q_{00}, \tau_0, *) \rightarrow (q_{11}, (\tau_0, 0)\, *)$ $(q_{00}, \{\lambda \backslash \tau_0\}, *) \rightarrow (q_{00}, \{\lambda \backslash \tau_0\}\, *)$ $(q_{00}, -, *) \rightarrow (q_{00}, \varepsilon)$
$\tau_i \in \lambda$	$(q_{ij}, \tau_i, *) \rightarrow (q_{i+1j+1}, (\tau_i, i)\, *)$ $(q_{ij}, \{\lambda \backslash \tau_i\}, *) \rightarrow (q_{ij}, \{\lambda \backslash \tau_i\}\, *)$ $(q_{ij}, -, (\tau_p, p)) \rightarrow (q_{pj-1}, \varepsilon)$ where $q_{pj-1} = max_{k<j}(q_{kj-1})$ $(q_{ij}, -, \lambda \backslash (\tau_p, p)) \rightarrow (q_{ij}, \varepsilon)$
$\tau_i = \{-\}$	$(q_{ij}, -, *) \rightarrow (q_{i+1j+1}, \varepsilon)$ $(q_{ij}, \{\lambda \backslash -\}, *) \rightarrow (q_{ij}, \{\lambda \backslash -\}\varepsilon)$

Figure 5.2: Rules for creating a pushdown automaton for a candidate tree

recursively applying the productions to a starting symbol until only terminal symbols remain. The sequential application of these productions to an input produces a *derivation* or *parse tree*, where all internal nodes are non-terminals and all leaf nodes are terminals.

An example CFG for a subset of arithmetic operations is given in Figure 5.1 (a). This particular grammar demonstrates recursive nesting (balanced parentheses), operator precedence (multiplication is more tightly bound than addition), and associativity (multiplication and addition are left-associative in this grammar). Figure 5.1 (b) depicts the parse tree given by the grammar for the equation $3 * (4 + 5)$.

5.2.2 Trees to DPDAs

Figure 5.2 describe the rules that are used for generating a pushdown automaton for detecting a subtree in an input tree and is taken from [31]. The notations for the rules are described as follows:

- λ: the set of labels for labeling the trees.
- $\sum = \lambda \cup \{-\}$: the alphabet for the automaton
- $\Gamma = \lambda \cup Z_0 \cup \langle \lambda, i \rangle$: stack symbols, where $\langle \lambda, i \rangle$ denotes a structure containing a symbol and a number of a state
- $\tau = \{\tau_0, \tau_1, ..., \tau_{k-1}\}$: the string encoding of the candidate tree for which the automaton is created, where τ_i is the i^{th} character in τ.
- $Q = \{q_{0j_0}, q_{1j_1}, ..., q_{kj_k}\}$: The states of the automaton were j_i denotes the level of the node in the tree for which the given state was created.
- $*$: Any symbol on the top of the stack.

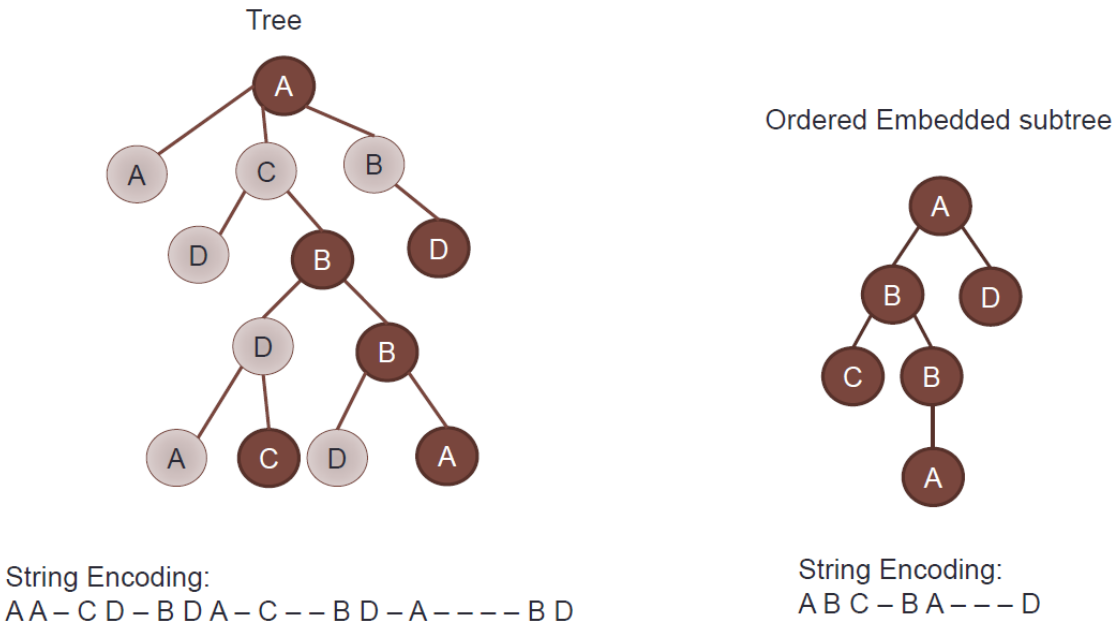

Figure 5.3: An example of tree and subtree. Using DPDA, we check of subtree exist in the tree.

Figure 5.3 represent a tree in the dataset and an candidate subtree. The task is to check if subtree exist in the tree. The trees are represented with strings as follows. The tree is traversed in preorder manner starting from the root, and the label X of the current node is added to the end of string (τ). Whenever the is a backtrack from a child to its parrent, a – symbol is added. When the last label is reached, the algorithm terminates and it does not traverse back to the root.

Figure 5.4 shows the pushdown automaton generated for the candidate subtree in Figure 5.3 using the rules shown in Figure 5.2.

5.3 Architectural Design

In this section, we describe the ASPEN architecture that augments LLC slices with support for DPDA processing. We also discuss the design of a DPDA processing pipeline based on ASPEN and the tradeoffs involved.

5.3.1 Cache Slice Design

The proposed ASPEN architecture augments the last level cache slices of a general purpose processor to support in-situ DPDA processing. Figure 5.5 (a) shows an 8-core enterprise Xeon-E5 processor with LLC slices connected using a ring interconnect (not shown in figure). Typically, the Intel Xeon family includes 8-16 such slices [90, 91, 92]. Each last-level cache slice macro is 2.5 MB and consists of a centralized cache control

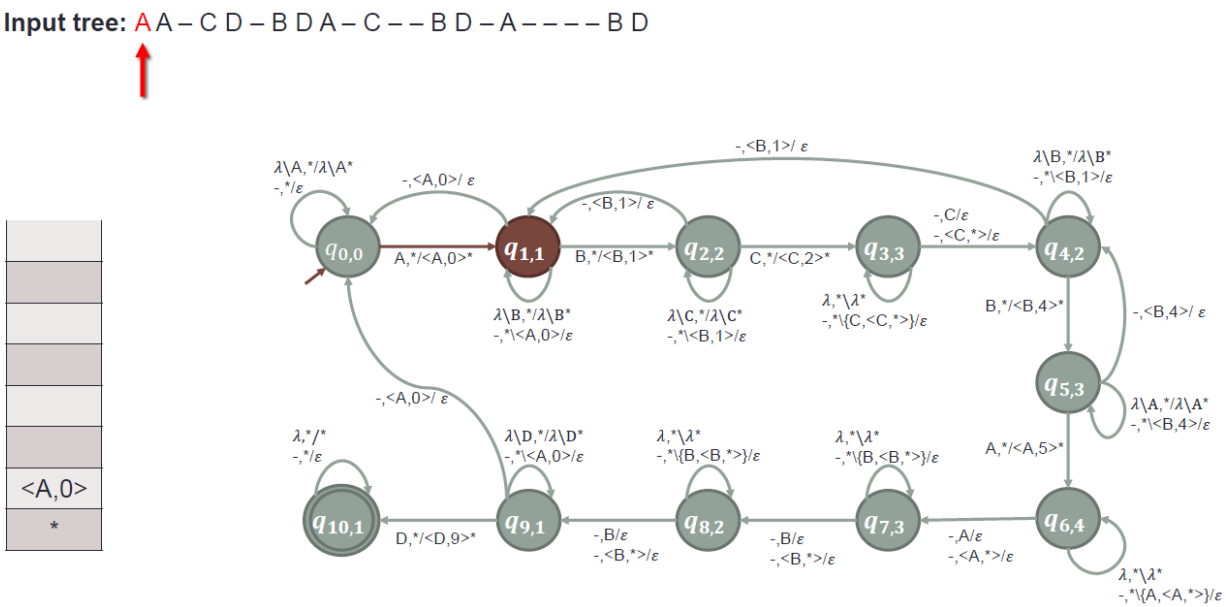

Figure 5.4: The DPDA generated for candidate tree in Figure 5.3

box (CBOX). A slice is organized into 20 ways, with each way further organized as five 32 kB *banks*, four of which constitute data arrays, while the fifth one is used to store the tag, valid and LRU state (Figure 5.5 (b)). All the ways of the cache are interconnected using a hierarchical bus supporting a bandwidth of 32 bytes per cycle. Internally, each bank consists of four 8 kB SRAM arrays (256×256).

A bank can accommodate up to 256 states and a DPDA can span several banks. We repurpose two of the four arrays in each bank to perform the different stages of DPDA processing. The remaining two arrays (addressed by the `PA[16]` bit) can be used to store regular cache data. State-transitions are encoded in a hierarchical memory-based interconnect, consisting of local and global crossbar switches (L-switch, G-switch). A 256-bit register is used to track the active states in each cycle (Active State Vector in Figure 5.5 (c)). We provision input buffers in the C-BOX to broadcast input symbols or tokens to different banks. Output buffers are also provided to track the report events generated every processing cycle.

5.3.2 Operation

This subsection provides the details of DPDA processing. Recall that, in a DPDA, only a single state is active in every processing cycle, and initially, only the start state is active. Each input symbol from the DPDA input buffer is processed in five phases. In the *input match* and *stack match* phases, we identify the active DPDA state which has the same label as that of the input symbol and the top of stack (TOS) symbol respectively. In the *stack action lookup* phase, the stack action defined for that state is determined (i.e., push symbol or number of symbols to pop from the stack). The stack is updated in the following phase (*stack*

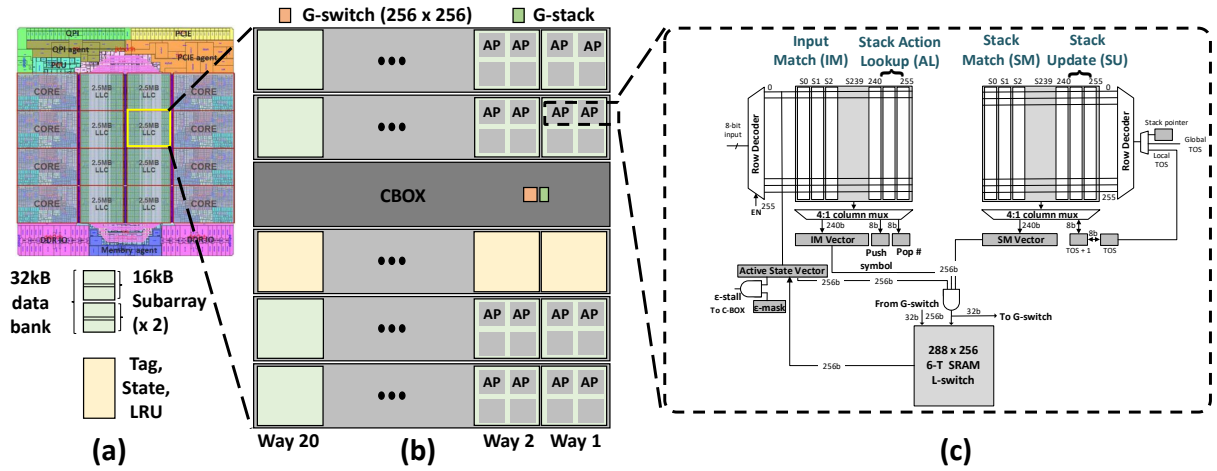

Figure 5.5: The figure shows (a) 8-core Xeon processor, (b) one 2.5MB Last-Level Cache (LLC) slice and (c) Internal organization of one 32kB bank with two 8kB SRAM arrays repurposed for DPDA processing.

update). Finally, in the *state-transition phase*, a hierarchical transition interconnect matrix determines the next active state.

Cycles in which states with an ε-transition are active require special handling. These states do not consume an input symbol but perform a stack action in that cycle (i.e., push or pop). A 256-bit ε-mask register tracks the ε-states in each bank. A logical AND of the ε-mask register and Active State Vector is used to determine if an ε-state is active in the next processing cycle. If an ε-state is active, a 1-bit ε-stall signal is sent to the C-BOX to stall the input for the next processing cycle.

While a single stack action per cycle is sufficient to support DPDA functionality, reducing stalls to the input stream can significantly improve performance. The *multipop* optimization, discussed in Section **??**, reduces stalls due to ε-transitions and is supported in hardware by manipulating the stack pointer and encoding the number of popped symbols in the stack action lookup phase. We now proceed to discuss the different stages involved in DPDA processing.

(1) Input-Match (IM): We adapt the state-match design of memory-centric automata processing models [85, 86] for the input-match phase. Each state is mapped to a column of an SRAM array as shown in Figure 5.5 (c). A state is given a 256-bit input symbol label which is the one-hot encoding of the ASCII symbol that it matches against. The homogeneous representation of DPDA states ensures that each state matches a single input symbol and each state can be represented using a single SRAM column. The input symbol is broadcast as the row address to the SRAM arrays using 8-bits of global wires. By reading out the contents of the row into the *Input Match Vector*, the set of states with the same label as the input symbol can be determined in parallel.

(2) Stack-Match (SM): In contrast to NFAs, where all active states that match the input symbol are candidates for state-transition, DPDA states have valid transitions defined only for those states that match

both the input symbol and the symbol on the top of the stack (8-bit TOS in Figure 5.5). We re-purpose an SRAM array in each bank to determine the set of DPDA states that match the top of stack (TOS) symbol. Similar to Input-Match, we provision 8 bits of global wires to broadcast the TOS symbol as the row address to SRAM arrays. By reading out the contents of the row into the *TOS Match Vector* and performing a logical AND with the *Input Match Vector* and the *Active State Vector*, the candidate states for state-transition are determined. We refer to these candidate states simply as *active states*.

We leverage sense-amplifier cycling techniques [86] to accelerate the IM and SM stages.

(3) Stack Action Lookup (AL): Each DPDA state is also associated with a corresponding stack action. The supported stack actions are `push`, `pop` and `multipop`. The stack action is encoded with 16 bits. Each `push` action uses 8 bits to indicate the symbol to be pushed onto the stack. The remaining 8 bits are used by the `pop` action to indicate the number of symbols to be popped from the stack (> 1 for `multipop`).

The stack action corresponding to each state is packed along with the IM SRAM array in each bank. However, in the AL stage, we lookup this SRAM array using the 256-bit result vector obtained after logical AND in the previous step (see Figure 5.5). This removes the decoding overhead from the array access time. We reserve 16 bits of global wires to communicate the stack action results from each bank to the stack control logic in the C-BOX.

(4) State Transition (ST): The state-transition phase determines the set of states to be activated in the next cycle. We observe that the state transition function can be compactly encoded using a hierarchy of local and global memory-based crossbar switches. The state transition interconnect is designed to be flexible and scales to several thousand states. The L-switches provide dense connectivity between states mapped to the same bank while the G-switch provides sparse connectivity between states mapped to multiple banks. A graph partitioning based algorithm [93] is used to satisfy the local and global connectivity constraints while maximizing space utilization.

The crossbar switches consisting of N input and output ports and $N{\times}N$ cross-points are implemented using regular 6-T SRAM arrays (e.g., L-switch in Figure 5.5 (c)). The 6-T bitcell holds the state of each cross-point. A flip-flop or register can also be used for this purpose but these are typically implemented using 24 transistors making them area inefficient. A '1' is stored in bitcell (i, j) if there is a valid transition defined from state i to state j. All the cross-points are programmed once during initialization and used for processing several MBs to GBs of input symbols. The set of *active* states from the previous phase serve as inputs to the crossbar switch. For DPDAs, only a single state can be active every cycle and we can use 6-T SRAM arrays for state transition, since only a single row is activated.

(5) Stack Update (SU): To allow for parallel processing of small DPDAs, (e.g., in subtree mining), we provide a local stack in each bank. We repurpose 8 columns of the SM array to accommodate the local stack.

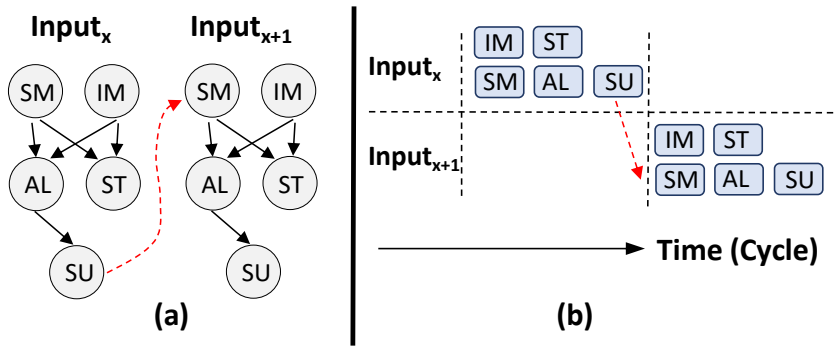

Figure 5.6: DPDA processing on ASPEN. (a) Dependency graph between stages. (b) Serial processing of input symbols.

Larger DPDAs (e.g., in XML parsing) make use of a global stack to keep track of parsing state. The global stack is implemented in the C-BOX using a 256×8 register file and is shared by all the DPDAs mapped to two adjacent ways. Providing a stack depth of 256 is sufficient for our parsing applications (see Section 5.5). Note that only one sort of stack (local or global) is enabled at configuration time based on the DPDA size. The stack pointer is stored in an 8-bit register and is used to address the stack. We also store the symbols at stack positions TOS and TOS+1 in separate 8-bit registers. This optimization saves a write and read access to the larger stack register file and ensures early availability of the top-of-stack symbol for the next processing cycle. The `push` operation writes the stack symbol to TOS+1. A lazy mechanism is used to update the stack with the contents of TOS. Similarly, the `pop` operation copies TOS to TOS+1, while lazily reading the stack register file to update TOS.

5.3.3 Critical Path

ASPEN's performance depends on two critical factors: (1) the time taken to process each symbol in the input stream (i.e., clock period) and (2) the time spent stalling due to ε-transitions. The multipop optimization reduces stalls due to ε-transitions. We now consider the clock period.

In a naïve approach, each input symbol would be processed sequentially in five phases, leading to a significant increase in the clock period. However, not all phases are dependent on each other and need to be performed sequentially. Figure 5.6 (a) shows the dependency graph for the DPDA processing stages. The intra-symbol dependencies are shown in black, while the inter-symbol dependencies are marked in red. Using the dependency graph, each of the five stages can be scheduled as shown in Figure 5.6 (b), where the propagation through the interconnect (wire and switches) for state-transition is overlapped with stack action lookup and stack update. Since the top of stack cannot be determined until the stack has been updated based on the previous input symbol, DPDA processing is serial. We contrast this to NFA processing, which has two independent stages (input-match and state-transition) which can be overlapped to design a

two-stage pipeline [86]. We find that the critical path delay (clock period) of ASPEN is the time spent for input/stack-match and the time taken for stack action lookup and update. The time spent in state-transition is fully overlapped with stack related operations. Section 5.4.3 discusses the pipeline stage delays and operating frequency.

5.3.4 System Integration

ASPEN shares the last level cache with other CPU processes. By restricting DPDA computation to only 8 ways of an LLC slice, we allow for regular operation in other ways. Furthermore, the cache ways dedicated to ASPEN may be used as regular cache ways for non-parsing workloads. Cache access latency is unaffected since DPDA-related routing logic uses additional wires in the global metal layers.

DPDAs are (1) placed and routed for ASPENs hardware resources, and (2) stored as a bitmap containing states and stack actions. At runtime, the driver loads these binaries into cache arrays and memory mapped switches using standard load instructions and Intel Cache Allocation Technology [94]. The input/output buffers for ASPEN are also memory-mapped to facilitate input streaming and output reporting, and ISA extensions are used to start/stop DPDA functions. We disable LLC slice hashing at configuration time. The configuration overheads are small, especially when processing MBs or GBs of input, but are included in our reported results. To support automata-based applications that require counting, we provision four 16-bit counters per way of the LLC.

5.4 Experimental Methodology

5.4.1 Experimental Setup

All CPU-based evaluations use a 2.6 GHz dual-socket Intel Xeon E5-2697-v3 with 28 cores in total, GPU-based evaluations use NVIDIAs's TITAN Xp. We used PAPI [95] and Intel's RAPL tool [96] to obtain performance and power measurements and NVIDIA's *nvprof* utility [97] to profile the GPU. We utilize the METIS graph partitioning framework [93] to map DPDA states to cache arrays.

We compare ASPEN against TreeMatcher [77], a single-threaded CPU implementation, and GPUTreeMiner [98], a GPU implementation. Both employ a breadth-first iterative search to find frequent subtrees. We evaluate using three different datasets, one real-world (TREEBANK[1]), and two synthetically generated by the tree generation program provided by Zaki[2] (T1M and T2M). Table 5.1 shows the details of the datasets. TREEBANK is widely used in computational linguistics and consists of XML documents. It provides a

[1] http:// www.cs.washington.edu/research/xmldatasets/
[2] http://www.cs.rpi.edu/~zaki/software/

syntactic structure for English text and uses part-of-speech tags to represent the hierarchical structure of the sentences. T1M and T2M are generated based on a mother tree with a maximal depth and fan-out of 10. The total number of nodes in T1M and T2M are 1,000,000 and 100,000, respectively. The datasets are then generated by creating subtrees of the mother tree. First, the database is converted to a preorder traversal labelled sequence representation. Then, for each subtree node, depending on its label and position, a set of predefined rules determines the corresponding DPDA. Detailed information on these rules can be found in Iváncsy and Vajk [31]. The total number of subtrees summed over all the iterations of the frequent subtree mining problem is given in the *#Subtrees* column.

5.4.2 Database

We evaluate the proposed architecture on three different datasets, one real-world (TREEBANK [3]), and two synthetically generated by the tree generation program provided by Zaki[4] (T1M and T2M). Table 5.1 shows the details of the datasets. TREEBANK is widely used in computational linguistics and consists of XML documents. It provides a syntactic structure of the English text and uses part-of-speech tags to represent the hierarchical structure of the sentences. T1M and T2M are generated based on a mother tree with the maximal depth and fan-out of 10. The total number of nodes in T1M and T2M are 1,000,000 and 100,000, respectively. The datasets are then generated by creating subtrees of the mother tree. *#Subtree* shows the total number of subtrees in all the iterations of frequent subtree mining problem.

Table 5.1: Datasets

Dataset	#Trees	Ave_Node	#Items	Max_Depth	#Subtrees
T1M	1M	5.5	500	13	9825
T2M	2M	2.95	100	13	3711
TREEBANK	52581	68.03	1387266	38	5280

Ave_Node = Average number of nodes per tree

#Items = Frequent label set size

Max_Depth = Maximum tree depth in the dataset

5.4.3 ASPEN parameters

Each 256×256 6-T SRAM array in the Xeon LLC can operate at 4 GHz [91, 92]. In the absence of publicly-available data on array area and energy, we use the standard foundry memory compiler at 0.9 V

[3] http://www.cs.washington.edu/research/xmldatasets/

[4] http://www.cs.rpi.edu/~zaki/software/

Table 5.2: Stage Delays and Operating Frequencies

Design	IM/SM	ST	AL	SU	Max Freq.	Freq Oper.
ASPEN	438 ps	573 ps	349 ps	349 ps	880 MHz	850 MHz
CA	250 ps	250 ps	-	-	4 GHz	3.4 GHz

in the 28nm technology node to estimate the power and area of a 256×256 6-T SRAM array. The energy to read out all 256 bits was calculated as 22 pJ. Since ASPEN is based on a Xeon-E5 processor modeled at 22nm, we scale down the energy per access to 13.6 pJ. The area of each array and 6-T crossbar switch were estimated to be 0.015 mm^2 and 0.017 mm^2 respectively. Each LLC slice contains 32 L-switches and 4 G-switches to support DPDA computation in up to 8 ways. These switches can leverage standard 6-T SRAM push-rules to achieve a compact layout and have low area overhead (~6.4% of LLC slice area). Being 6-T SRAM based, these switches can also be used to store regular data when not performing DPDA computation. Similar to the Cache Automaton [86], we use global wires to broadcast input/stack symbols and propagate state transition signals. These global wires with repeaters have a 66ps/mm delay and an energy consumption of 0.07pJ/mm/bit.

Table 5.2 shows the stage delays for DPDA processing on ASPEN. The IM/TM phases leverage sense-amplifier cycling [86] and take 438 ps. The ST stage requires 573 ps, composed of 198 ps wire delay and 375 ps due to local and global switch traversal. AL and SU each take 349 ps, composed of 99 ps wire delay and 250 ps for array access.

5.5 Evaluation

To evaluate the benefits of DPDA hardware acceleration for the subtree inclusion kernel, we consider the frequent subtree mining (FTM) problem, where the major computation is subtree inclusion checking. FTM is composed of two steps. In the first step, the subtree candidates of size $k+1$ (*(k+1)-candidates*) are generated from the frequent candidates of size k (*k-frequent-candidates*), where k is the number of nodes in a subtree. Candidate-generation details and a proof of correctness are provided by Zaki [77]. In the second stage, for each candidate subtree, we count the number of occurrences (inclusions) of that subtree in the dataset. If the count exceeds a specified support threshold, we report the candidate as frequent and use it as a seed in the next generation step.

Table 5.3 lists the architectural parameters for the FTM application on different datasets. This application is compatible with the hardware restrictions, including maximum stack depth and supported alphabet size. In contrast to XML parsing, there are no ε-transitions in the subtree inclusion DPDAs, which means that runtime is linear in the length of the input data. The homogeneous DPDAs designed for FTM have an average node fan-out of 2.2 (maximum of 4).

Table 5.3: Architectural Parameters for Subtree Inclusion

Dataset	Automata Alphabets	Stack Alphabets	Stack-Size
T1M	16	17	29
T2M	38	39	49
TREEBANK	100	101	110

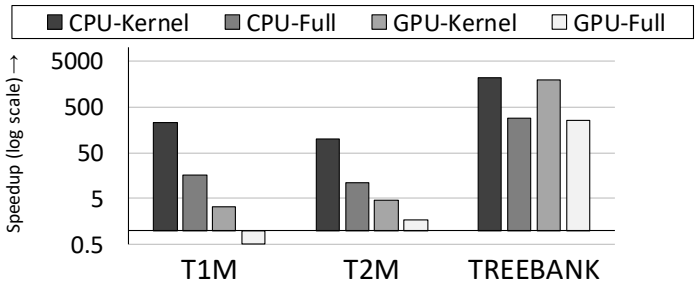

Figure 5.7: Speedup of ASPEN over CPU and GPU.

Figure 5.7 shows the kernel and total speedup of ASPEN over CPU and GPU baselines. For ASPEN, we include timing for pre-processing, intermediate processing (between iterations) on the CPU, loading time (transferring data from DRAM to LLC), and reporting time (moving report vectors back to DRAM), in addition to the kernel time.

ASPEN shows 67.2× and 6× end-to-end performance improvement over CPU and GPU (Figure 5.7). TREEBANK consists of larger trees with higher average node out-degree, which makes its processing difficult on the CPU and GPU. In particular, TREEBANK has an uneven distribution of trees with different sizes in the database, which causes the synchronization overhead between the threads in a warp to increase. In addition, larger trees also increase the thread divergence in a warp, because the possibility of checking a subtree node against different labels in the input tree of the same warp increases. Therefore, GPUs are not an attractive solution for larger trees. On the other hand, GPUs show 2× speedup over ASPEN on T1M. This is because the T1M dataset consists of small and evenly sized trees. Unlike CPUs or GPUs, the complexity of subtree inclusion checking in ASPEN is independent of the input dataset.

Figure 5.8 shows the total energy for ASPEN, CPU, and GPU implementations. The trends in energy are similar to that of performance. The unevenly-sized large trees in TREEBANK increase the runtime of CPU and GPU, leading to an increase in total energy. On average, ASPEN achieves 3070× and 6279× improvements in total energy when compared to CPU- and GPU-based implementations, respectively.

5.6 Conclusion

We present ASPEN, a general-purpose, scalable, and reconfigurable memory-centric architecture that supports rich push-down automata processing for tree-like data. We design a custom datapath that performs state

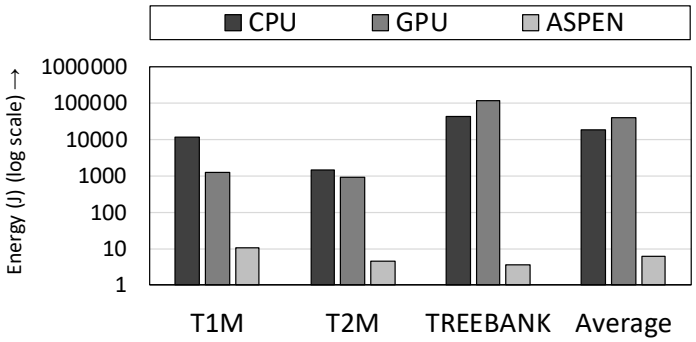

Figure 5.8: Total Energy of ASPEN compared to CPU and GPU.

matching, stack update, and transition routing using memory arrays capable of the state matching, stack manipulation, and transition routing operations of pushdown automata, all efficiently stored and computed in memory arrays. We also develop a compiler for transforming large classes of existing grammars to pushdown automata executable on ASPEN.

Our evaluation against state-of-the-art CPU and GPU tools shows that our approach is highly performant (up to 37.2× faster for subtree inclusion), and energy efficient (3070× lower for subtree inclusion). By providing hardware support for DPDA, ASPEN brings the efficiency of recent automata acceleration approaches to a new class of applications.

Chapter 6

A Scalable and Efficient in Memory Accelerator for Automata Processing

The interconnect design of existing automata processing accelerators are either incapable of efficient place-and-route of a highly-connected automaton or over-provision hardware resources for interconnect, at the expense of resources for state-matching. However, real-world benchmarks are quite large in terms of number of states, too big to fit in a single hardware unit, and thus usually need multiple rounds of reconfiguration and re-processing of the data. This incurs significant performance penalties and make makes state-matching resources a scarce resource.

The AP re-purposes DRAM arrays for the state-matching and proposes a hierarchical FPGA-style programmable interconnect design. Our study on a diverse set of 19 automata benchmarks reveals that congestion in the AP routing matrix cripples efficient state utilization, especially for difficult-to-route automata. This means that only 13% of the state matching resources in a block are utilized in Levenstein automata, and the remaining 87% cannot be used because there are not enough routing resources left. Moreover, although density of DRAM memory is high, one AP chip can only store 1.5MB of data (i.e., state matching rules), whereas a conventional DRAM of equal area can store 25MB of data [99, 29]. This implies that a majority of the chip area is likely spent for the interconnect and hiding DRAM latency.

Recently, Subramaniyan et al. [29] proposed an in-SRAM automata processing accelerator, Cache Automata (CA), by re-purposing last-level cache for the state-matching and using 8T SRAM cells for the interconnect. To address routing congestion in the AP, CA proposes to use a full-crossbar (FCB) topology for the interconnect to support full connectivity in an automaton, meaning there can be an edge between every two states. This implies a full-crossbar of size 256 needs 256^2 switches. This means that more than 50% of

the hardware resources in CA are spent for interconnect! However, our study of 19 automata benchmarks reveals that on average, only 0.53% (maximum 1.15%) of the switches are utilized. Therefore, full crossbars are extremely inefficient and costly for the automata processing applications. This expensive interconnect has an opportunity cost in terms of using that area for state matching.

To address the interconnect inefficiencies in the existing in-memory automata processing architectures, this work presents a *reduced-crossbar (RCB)* design, a low-overhead and yet flexible interconnect architecture that efficiently implements state-transition. RCB design is inspired by intrinsic properties of real-world automata connectivity patterns. RCB requires at least 7× fewer switches compared to the FCB design used in CA. This in turn reduces the wire length, which results in shorter latency and lower power consumption. In addition, the area efficiency of RCB provides an opportunity to design a denser state matching resources, which can accommodate more states and results in fewer rounds of re-configuration and re-processing of data.

Across 19 application from [2, 100], 17 of them can entirely map to RCB design and no FCB is required. To provide a general interconnect solution for every possible connectivity topology, we design a reconfigurable memory array for state-matching, in which blocks can be re-purposed as an FCB to provide full connectivity when needed (at the expense of some state capacity). In addition, to support an automaton with larger number of states, we design global switches that provide inter-block connectivity between RCBs and FCBs blocks.

To efficiently allow many-to-many transitions in an automaton, the underlying memory technology for eAP should be able to support logical OR functionality within memory rows in a subarray. This requires memory cells (a) to provide non-destructive read, and (b) to drive output to a "stable" state (logical OR in this case) when multiple bitcells drive a common bitline. 8T SRAM cells [35] and gain-cell embedded DRAM (GC-eDRAM) [36, 37] are examples of feasible memory technologies to implement eAP. Note that conventional DRAM and Reduced-Latency DRAM (RLDRAM) [101] cannot be used for this purpose. They have destructive reads and the value of the simultaneously-activated rows cannot be recovered in the write-back phase.

CA design uses 8T SRAM cells. In this work, we evaluate eAP on both 8T and 2T1D (2 transistor 1 diode) memory cells. The 2T1D cell is a GC-eDRAM designed and fabricated by [38]. 2T1D uses substantially fewer transistors than an 8T SRAM cell and thus incurs lower area overhead, which results in higher state density and therefore better throughput (due to the reduced rounds of reconfiguration and re-streaming of input). The scalability of gain-cells has been extensively studied in FinFET technology [102, 103, 104], which show gain cells have the potenetial to scale to smaller technology nodes in FinFETs.

Interestingly, the wire-OR capability of 8T or 2T1D memory arrays can also be utilized for in-situ computation of other important kernels in neural networks and graph processing. For example, very recent

studies explore the potential of processing binary neural networks computations using 8T SRAM cells and its alternatives [105, 106].

6.1 The Importance of Capacity

In CA, the authors use the ANMLZoo benchmarks to calculate cache utilization and report 1.2MB of cache usage on average. The automata provided in ANMLZoo benchmark suite [2] represent just a small portion of the actual application (normalized to fill one AP chip). However, real applications are much larger, with many independent automata comprising the various patterns that make up the full application, which requires orders of magnitude more states than reported in Table 1 in [2].

We illustrate this issue using sequential pattern mining (SPM) [7] benchmark, used in ANMLZoo. SPM is an iterative algorithm where in each iteration of the algorithm, a set of sequence candidates (automata) are checked against the input stream. A relatively small but realistic dataset in SPM requires about 300× more state capacity compared to SPM benchmark in ANMLZoo in order to run the whole application. This means that in order to execute one iteration of the algorithm on a parallel automata accelerator such as one AP chip (48K state capacity), we need to reconfigure the hardware 300 times ($reconfig$), each with a subset of the overall problem, and each time stream the whole input string. This incurs a large overhead from repeatedly re-streaming the input, as well as reconfiguration time. The overall execution time for a large application is shown in Equation 6.1.

$$Total\_time = \#reconfig \times [configOverhead + (\#inputSymbols \times cycleTime)] \quad (6.1)$$

Therefore, providing an accelerator with higher state capacity can significantly improve performance.

6.2 Interconnect Architecture

In this section, we first describe a simple implementation of interconnect using memory subarrays (FCB) and then, we present an efficiently compact and reconfigurable interconnect design (RCB) and its feasibility in hardware. Then, we discuss the potential switch cells (memory cells) that can be used in our interconnect architecture.

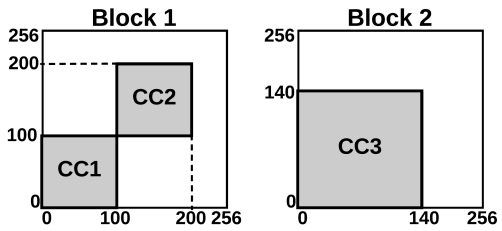

Figure 6.1: Full-crossbar utilization

6.2.1 Reduced Crossbar Interconnect

The interconnect should provide functionality for every STE to wake up all their successors in one step. This process should be done in one cycle, since the triggered successors are needed to process the next symbol in the next cycle. This implies an interconnect that is statically programmed and can ensure that all required paths are routable, non-blocking, and contention-free. More conventional interconnects would require many steps to process all the activations for each symbol. For example, buses can carry many bits simultaneously but cannot support a large number of clients. Ring, mesh, and hypercube are multihop, typically many hops, and contention is a problem.

Full Crossbar Interconnect (FCB) is a straightforward interconnect topology for connecting STEs in an automata, where every state is connected to every other state (including itself) at the cost of $O(N^2)$ (N is the number of states). To model transitions from multiple STEs to one STE, the output should be connected to multiple inputs. This is equal to logical OR of active inputs. Therefore, there is no need for dynamic arbitration. Cache Automata (CA) uses the FCB interconnect topology for both local and global switches.

NFAs for real-world automata applications are typically composed of many independent rules or patterns, which manifest as separate *connected components (CCs)* with no transitions between them. Each connected component has usually a few hundred states. All the connected components can thus be executed in parallel, independently of each other. Therefore, a crossbar switch can be utilized by packing connected components as densely as possible using a greedy approach [29].

However, using an FCB is very inefficient in routing resources. Assume the FCB switch block's size is 256×256. In a greedy approach, CCs are first sorted based on the number of states in each component and then, are assigned to the interconnect resources. Assume there are three CCs of size 100, 100, and 140. Figure 6.1 shows mapping of CCs to the FCB switch blocks. Switches in gray areas are configured for the corresponding CC. White areas (70% of total area), are unused switches. Moreover, within each connected component, transitions are sparse, meaning very few switches in the gray areas are used.

We observed that in our 19 real-world and synthetic benchmarks, in average, fewer than 0.48% (maximum 1.1% in Levenshtein) of switch cells (256^2 cells) are utilized in the FCB interconnect solution. This shows

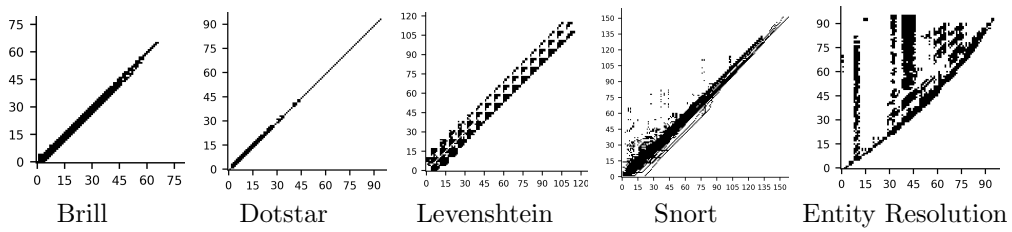

Figure 6.2: Union heatmap of routing switches with BFS labeling

that FCB model is extremely inefficient for automata processing applications and forces larger area overhead, power consumption, and delay in the state-transitions phase.

To motivate our efficient and compact interconnect, we visualize the connectivity matrix for the automaton in each benchmark with an image. We first label each node in an automata with a unique index using breadth first search (BFS) numeric labeling, since BFS assigns adjacent indices to the connected nodes. To draw the image, we model an edge (transition) between two nodes (with indices i and j) in an automaton with a black pixel at coordinate (i,j).

In Figure 6.2, each graph shows the union over all connectivity images for connected components in one benchmark. We chose union to make sure that we have considered every possible transition, even for rare connection patterns. Except Snort and Entity Resolution, the rest of the benchmarks (17 out of 19) represent a nice diagonal connectivity property. The union image and average image for the rest of benchmarks can be found here[1].

This diagonal connectivity pattern motivates a more compact and efficient interconnect, and comes from two properties: first, the power of numeric BFS labeling, which tries to label a child node closely to its parent(s); second, CCs are mostly tree-shape graphs with short cycles and the nodes have a small out-degree. Motivated by these observations, we propose a *reduced* crossbar interconnect (RCB), which has switch patterns similar to what we observed in the union images. RCB will have a smaller area overhead, lower power consumption, and smaller delay compared to FCB. Moreover, it can be applied to CA or AP without reducing their computation power.

Feasibility support for RCB Design: In order to actually save area in RCB design, we need to compact the memory array, still with the same amount of input and output signals similar to an FCB, but with smaller area overhead, since it needs less switches. This might complicate the layout process because wiring congestion may happen while compacting the array. Automated layout generation tools sometimes are not clever enough to provide the best compacting scheme even for regular patterns like RCB. Therefore, we propose a simple scheme to compact a FCB array to a smaller RCB array.

[1] https://github.com/anonymousUser0/isca/tree/master/Heatmap

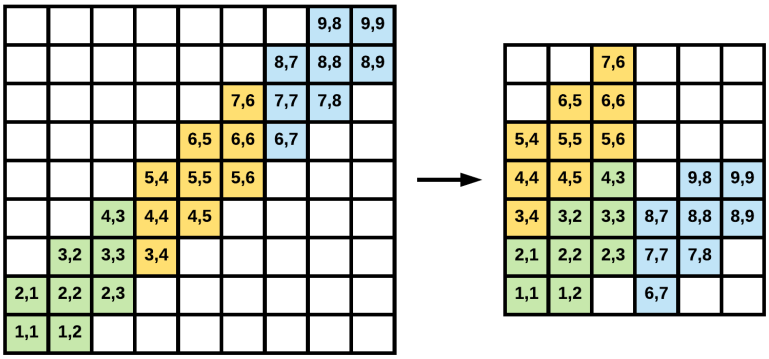

Figure 6.3: FCB to RCB compression

Simply flipping the diagonal-shape interconnect to a horizontal or vertical block forces the wire congestion in one dimension and it does not utilize the other available dimension to contribute in signal routing. However, squeezing the diagonal-shape to a square shape would significantly compact the subarray and at the same time, spread the burden of signal routing in both dimensions.

Figure 6.3 shows a toy example for an FCB subarray of size (9,9) with diagonal width of 3. In each square, the first index shows the row-index and second one shows the column index. For example, a switch in the location (4,3) shows that the input signal comes from a STE labeled 4 (in BFS) and it is connected to a STE labeled 3. The left block shows the initial naive mapping of diagonal memory cells, while all the white regions are the wasted areas (or switches). The right block shows how moving nearby memory cells close to the lower left side can reduce 9×9 array to 7×6. Our placement guarantees that in each row and column, wire counts has increased to a maximum of 3 times compared to the original FCB placement. For example, in row 4 of RCB, there are two input signals (word-line signal), 2 and 9, and in column-3, there are two output signals (bitline signal), 3 and 6.

Our calculation shows that an FCB of size 256×256 and diagonal width 21 can be reduced to a RCB of size 96×96, which results in approximately 7× switch saving. From our experiments, we found that the diagonal width of 21 is a safe margin to accommodate all the transitions (except in Entity Resolution and Snort). It should be noted that in the routing subarrays, there is no need for decoding the input, because the "active state vectors" (or an array of registers) are directly connected to the wordlines. Therefore, RCB does not incur any extra area overhead for extra decoders. Moreover, RCB has smaller bit-lines due to area compression, which potentially leads to a shorter memory access cycle.

6.2.2 Mapping to Memory Technologies

As discussed earlier, to implement the proposed interconnect in memory, the underlying memory technology should be able to support logical-OR functionality among memory rows in a subarray. This requires memory

cells (1) to provide non-destructive read (it means data is maintained after read operations and write-back is not necessary), and (2) to drive output to a "stable" state (logical OR in this case) when multiple bitcells drive a common bitline.

Clearly, conventional DRAM and Reduced-latency DRAM (RLDRAM) [101] cannot be adopted, because they have destructive reads and wired-OR destroys the value stored in every node participating in the OR operation. Furthermore, 6T SRAM is not also able to perform wired-OR, because if two cells with different values drive the same bitline, the resulting value would be unstable or undefined. On the other hand, 8T SRAM cells [35] and gain-cell embedded DRAM (GC-eDRAM) [36, 37] appear to be the most suitable memory technologies to implement eAP.

Gain Cell embedded DRAMs (GC-eDRAMs) are comprised of 2-3 standard logic transistors and optionally an additional MOSCAP or diode [107]. Recent adoption of GC-eDRAMs as on-die caches [37, 36, 108] provide realization for in-eDRAM acceleration of the state-of-the-art applications. Three-transistor (3T) [37] and two-transistor (2T) [109, 38] GC-eDRAMs are particularly beneficial for providing (1) a fast read-cycle time, and (2) non-destructive read, by splitting read and write paths to the cell. The latter property is especially useful for the interconnect design, where wired-OR functionality is needed.

In this paper, we adopt two memory cell technologies as the reconfigurable switches to evaluate our architecture: (1) 8T SRAM cells, as used in CA [29], and (2) the 2T1D (2 transistor 1 diode) GC-eDRAM cell [38]. Compared to 8T SRAM, 2T1D cell uses substantially fewer transistors and has lower leakage current [110, 109, 36]. Both cell types provide the wired-OR. The scalability of gain-cells has been extensively studied in FinFET technology [102, 103, 104], suggesting that gain cells are promising to scale to smaller technology nodes in FinFETs and to maintain an area advantage over 8T.

2T1D Switch Cell: The 2T1D DRAM cell holds the connectivity value in the switch, which is '1' if the switch is connected and '0' if it is disconnected. A connected switch implements an existing transition between two STEs in a state machine. Figure 6.4 shows the details of the 2T1D cell. The cell itself consists of a PMOS (PW) transistor for write operation, an NMOS (NS) for read operation, and an N-Type Gated-Diode (NC) for reducing coupling effect.

The cell has two modes: write mode and route mode. As shown in Figure 6.4, during the write mode, *Write-Worldline* is '1' and the value on the *Write-Bitline* is stored in the node "X". The *Write-Bitline* value controls a switch between STEs to be connected or disconnected. *Write-Bitline* is VDD for the connected switch and GND otherwise. During the route mode, the values that are stored determine whether there is a connection between a source STE (active state) and destination STEs (potential next states).

In the state transition part of Figure 2.1, vertical wires are *Read-Wordlines* and horizontal wires are *Read-Bitlines*. There is one switch in each cross point and the ones with the black dots show that the

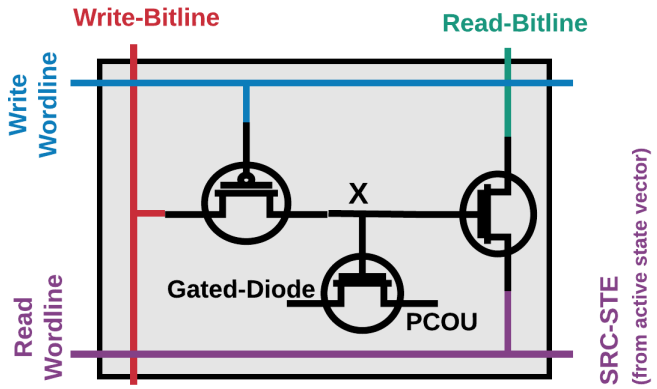

Figure 6.4: 2T1D switch cell

switch is connected. If the switch is connected and the source STE is in an active state, then coresponding *Read-Bitlines* activate the potential next states (in more detail, the *Read-Bitlines* are discharged. Therefore, the sense amplifier connected to the *Read-Bitline* will sense '0' and then is conveted to '1' after a NOT gate).

8T Switch Cell: We adopt the switch cell design from CA [29]. The cell consists of a 6T SRAM cell and two additional transistors, which connect the cell to a bitline. This allows a 6T cell to drive the bitlines only when the cell holds '1' and the input signal (active state vector in this case) is '1'. This implies that 8T cells can support OR functionality.

6.3 Embedded Automata Processor

In this section, we explain the design of eAP architecture for one bank. The bank design of eAP_2T1D and eAP_8T is very similar. The banks are replicated to in order to accommodate large number of automata. The overall capacity of eAP_2T1D and eAP_8T is different and is discussed in Section 6.6.6.

6.3.1 eAP Bank Design

Figure 6.5 shows the general overview of a bank in eAP. Each bank consists of multiple *subarrays* (Figure 6.5 (a,b)), which share a global decoder, a global sense amplifiers, and a set of global bitlines that connect local sense amplifiers to the global sense amplifiers. Each subarray has its own local sense amplifiers and local decoder. Based on subarray-level parallelism (SALP) idea [111], with small changes in the global decoder, we can access to more than one row by reducing the shared resources and enable activation to different subarrays to be done in parallel. Therefore, activation and precharging can be done locally within each subarray. In this paper, we utilize SALP for the state-matching phase in automata processing in order to match an input symbol with multiple automata in parallel.

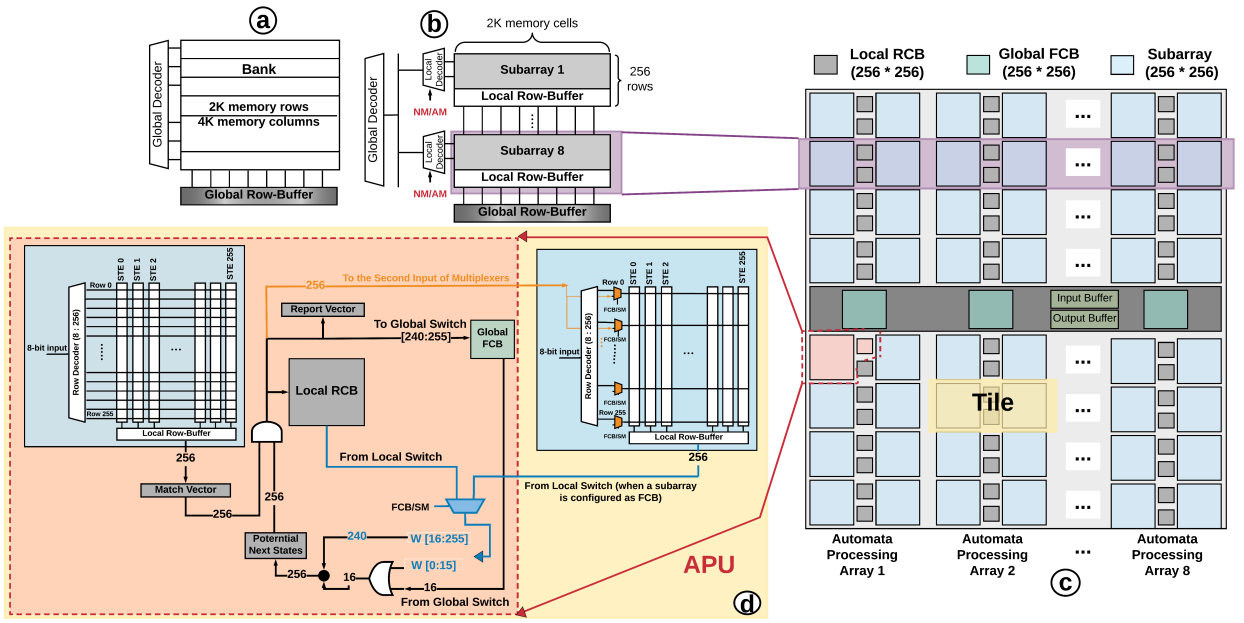

Figure 6.5: (a) Bank abstraction. (b) Physical implementation of a bank. (c) General overview of eAP architecture in one bank. (d) Inside one tile with datapath and communication to local (RCB) and global switches (FCB)

In our design, a memory bank supports two modes; *normal mode (NM)*, ie. for data storage as last level cache, and *automata mode (AM)* (Figure 6.5 (b)). During the NM, the global decoder only selects one of the connected subarrays based on the input address, and then selects one row within the subarray. During the AM, all the local decoders get the same address (input symbol) from the global decoder and activate the same row in each subarray, in parallel, based on the input symbol. The entire row corresponding to that symbol is read to the sense amplifiers, yielding a vector of all the states accepting the input symbol. This arrangement is shown in Figure 6.5 (c) maps to the blue square-blocks.

There is no need for column addressing, because all the local sense amplifiers (*match vectors*) should be read and propagated to the *state transition* stage. Automata mode only requires read operations. Configuration of STEs (memory columns) is done at context-switch time in normal mode using write operations.

In Figure 6.5 (c), each bank has eight columns of automata processing arrays (APA) with maximum capacity of 4096 STEs each. Each APA consists of eight *tile*s and each tile contains two automata processing units (APUs). Each APU hosts a memory subarray of 256×256 for state matching (blue squares) and a RCB subarray (smaller gray square) with aggregate size of 256 nodes as local interconnect. Inside each APA, tiles are connected to work collaboratively through a global switch (FCB of size 256×256) to process larger connected components that do not fit in a single APU. These choice of parameters are based on some prior organizations [28, 29].

The global FCB switch allows 16 states in each APU, called *port nodes* (PNs), to communicate with all PNs of different APUs in the same APA. The global FCB is positioned in the middle of the APA to minimize the longest required wire to/from bottommost and topmost APUs.

For uncommon cases in which a connected component does not fit into an RCB interconnect (such as EntityResoloution, see Fig. 6.2), eAP repurposes state matching subarrays as FCB interconnects. Specifically, it combines the state-matching subarray of one of the APUs in the tile (as a full crossbar interconnect) and the state-matching of the other APU in the same tile. When a subarray needs to be configured as an FCB instead of regular state match operation, the FCB/SM signal (Fig. 6.5.d right blue square) of that tile is set to one. This signal selects the word lines of the target subarray to be driven by the match vector register bits instead of the decoder output (See Fig. 6.5.d). This mode halves state capacity of the contributed tile but provides the ability to accept connected component without any limitation on interconnect shape.

To support this functionality, an array of 2:1 multiplexers needs to be added for one of the subarrays in each tile (FCB/SM mutliplexers in the right blue square of Fig. 6.5.d). This has less than 2.5% area overhead based on industry 28 2:1 mux area numbers[2]. This reconfiguration promotes a tile to embed any connected component (with size less than 256) plus having 16 PNs to communicate with other APUs in the same APA to provide more flexible interconnect topology in a column.

6.3.2 Pipeline Design

To process a single input symbol, two memory accesses are required; one for finding the *match vector* in the state-matching phase and one for finding the *potential next state vector* in the state-transition phase (see Figure 2.1 (b)). The result of state matching of the current symbol is stored in the *Match Vector* registers, which acts as a pipeline registers, and can be overlapped with the state transition routing from the previous input symbol matches.

Cache Automata [29] proposes a three-stage pipeline for automata processing, shown in Figure 6.6 (a) (SM: State-Match, GS: Global-Switch, LS: Local-Switch). However, we have found that this pipeline has a data-hazard issue. To process input symbol i+1, the result of state-match of the current cycle (i+1) and state-transition (including LS and GS) of the previous cycle (i) should be ready at the end of stage-2. However, the LS output is only ready at the end of third stage. To solve this, one pipeline stall is necessary for each input to resolve the hazard, which decreases the throughput by factor of 2. Another solution (to avoid data hazard) is merging GS and LS in one stage, but they need to operate sequentially (see Figure 1(d) in [29]).

[2]This is obtained using a standard cell library provided under NDA, so while we can describe the result, we cannot identify the vendor.

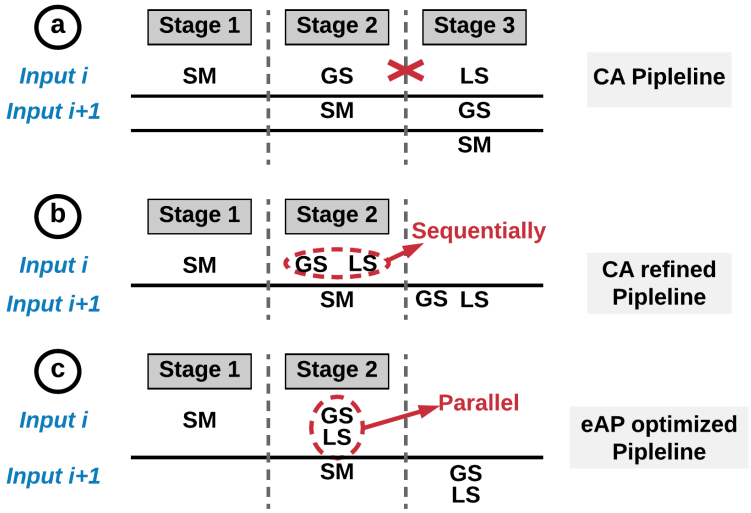

Figure 6.6: CA (a) vs eAP (b) pipeline

This means that stage 1 has one memory access whereas stage 2 has two consecutive memory accesses. Figure 6.6 (b) shows our refined version of CA pipeline design. This has been verified with the authors.

Unlike CA, our proposed pipeline tries to balance the amount of work between the two stages of pipeline, since the final frequency is determined based on the slowest stage. Figure 6.5 (d) represents the interconnect organization. Both global and local switches can operate in parallel in one stage and the result from the global switch is ORed with the corresponding wires from the local switch (Figure 6.6 (c)). Performing an additional 16-bit OR operation costs much less than one memory access. Similar pipeline optimization (parallel GS and LS) can be applied to CA. Performance results for both designs are shown in Section 6.6.3.

6.3.3 Input and Output

eAP has two asynchronous FIFOs to hold the input symbols in the input buffer (IB) and reports in the output buffer (OB). The host CPU communicates with the IB and OB using interrupt triggered memory-mapped IO or DMA while the interrupt service routine (ISR) is responsible to fill in the IB and evict the OB. Assuming 1.5 and 2.5 working frequency for eAP_2T1D and eAP_8T respectively (see Section 6.6.3) and 1 frequency for interrupt, an IB of size 2.5KB can store enough data to feed the eAP until the next IB interrupt. Recently, [112] has characterized the reporting statistics of ANMLZoo's benchmark. The results show that 10 out of 12 benchmarks produces less than 0.5 reports per cycle (in average). This investigation motivates us to use 512 entries for the OB (4 bytes each for report meta-data) to keep a similar interrupt rate as the IB.

After writing the automata configuration bits in the normal operation mode (NM), eAP switches to automata mode (AM) and starts consuming inputs from the IB. Buffers have two output signals (E and F) to show if they are full or empty. E-signal of the IB and F-signals can raise the interrupt signal of the

CPU to service the device as needed. In Automata mode, in each cycle, the symbol at the front of the IB is popped and drives the shared address bus of all banks contributing in eAP symbol matching. Each APU is equipped with a *report vector mask* to identify report states in each cycle by simply performing a bitwise AND operation with the *active vector*. We use the Report Aggregator Division (RAD) mechanism proposed in [112] (which is an improvement over Micron's AP reporting procedure) to fill up the OB with report state IDs and cycle information. RAD adaptively shrinks the report message based on the current number of active states to use the OB space efficiently. When the OB is filled up, an interrupt signal is raised to ask for service from the host CPU and free space for future report events.

6.3.4 System Integration

This section discuss the possible integrations for eAP with 2T1D GC-eDRAM cells (eAP_2T1D) and 8T SRAM cells (eAP_8T). High-bandwidth On-Package Memory (OPM) introduces a new on-package memory layer between off-chip DRAM and on-chip cache in conventional memory hierarchy. Intel has included eDRAM as an OPM in its Haswell, Broadwell and Skylake architectures to fill the the gap between on-chip and off-chip memory bandwidth. For Haswell and Broadwell processors, eDRAM with 1T1C cells was used as L4 cache [113, 114]. For eAP_2T1D, we replace the 1T1C eDRAM cells with 2T1D and then, re-purpose a postion of banks in L4 cache for automata processing.

For eAP_8T, we assume the same integration as Cache AUtomata [29]. Cache Autoamta re-purposes last level cache (L3) slices for automata processing and access the cache *ways* by leveraging Cache Allocation Technology (CAT) [115]. For both eAP designs, in automata mode, the compiler generates a configuration array (the state match and interconnect configuration bits) and writes it in the eAP memory address space to start offloading the input task.

6.4 Compiler

Our compiler has two main tasks. First, it should check if a connected component can fit into a RCB switch template or needs to be mapped to an FCB. Second, it should provide a mapping from each state of the automaton to its hardware representation (STE). To accomplish the decision problem (RCB or FCB), a fixed matrix representation of the RCB interconnects is initially generated (See Figure 6.3), called a diagonal matrix (DM). We assign a '1' in row i and column j, if there is a switch at location i and j in RCB interconnect.

For any given automaton, we first number nodes using BFS traversal, starting from a fake root connected to all nodes that are start nodes in the automata. Then, we calculate the connectivity matrix of given

automaton using BFS assigned numbers. If the calculated matrix is a subset of the DM, then it can be fit into a diagonal switch box (RCB). Otherwise, the given automaton should fit into a FCB.

For diagonal automata, we search through all the previously-assigned RCB interconnect blocks and try to find the one with the least free capacity that can still fit the current automaton being placed. We keep the same BFS order of labels to assign inputs of the assigned interconnect block, but with an offset equal to the last-used input of that interconnect block, instead of 1 for the first automaton (connected component) that was assigned to this interconnect block. If there is no such partially used interconnect with enough spare capacity, we initialize a new RCB interconnect block from the pool of available interconnect blocks.

Our compiler supports a set of optimizationa such as enforcing constraints on fan-in and fan-out, automaton merging, and minimization. The utilization for RCB and FCB blocks is discussed in Section 6.6.1.

Input : list of automatas **L**
Output : mappings **M**[RCB, FCB] from **L** to interconnects
foreach *automata* **A** *in* **L** **do**
  label **A** *using* BFS
  find **C**, connectivity matrix of **A**, *using* BFS labels
  if **C** *is a subset of RCB routing matrix* **then**
    RoutingType ← RCB
  else
    RoutingType ← FCB
  end
  find $\mathbf{M_i}$ in **M**[RoutingType] *with* smallest number of unmarked inputs that can fit **C**
  if $\mathbf{M_i} = \varnothing$ **then**
    initialize $\mathbf{M_i}$, a new RoutingType interconnect
    add $\mathbf{M_i}$ *to* **M**[RoutingType]
  end
  add **C** *to* $\mathbf{M_i}$
  mark used inputs of $\mathbf{M_i}$
end

Algorithm 1: Mapping Algorithm

6.5 Evaluation Methodology

Applications: We evaluate the eAP architecture using ANMLZoo [2] and Regex [100] benchmark suites. ANMLZoo represents a set of diverse applications including machine learning, data mining, and security. We use the standard 10MB inputs stream included in ANMLZoo.

Experimental Setup: We evaluate eAP architecture on memory arrays with 2T1D cells (we call it eAP_2T1D) and 8T cells (we call it eAP_8T). In eAP_8T, both state matching and interconnect memory arrays are based on 8T cells. This is because we sometimes re-purpose state-matching arrays for interconnect and they should be able to provide the required logical OR functionality (6T SRAM cells are unable to provide

OR functionality because multiple cells cannot drive one bitline). We compare eAP_2T1D and eAP_8T with CA, CA_opt, and the AP, all using (or scaled to) 28nm technology. In CA, state-matching is based on 6T SRAM arrays and interconnect is based on 8T SRAM arrays. To calculate area, power, and row cycle time of memory arrays, we use a standard memory compiler. For 2T1D analysis, we rely on the results from the designed and fabricated chip in [38].

We develop an in-house cycle-accurate automata simulator[3] to perform software optimization on the automata, map them to the proposed architecture, and extract per-cycle statistics for the energy estimation.

6.6 Results

In this section, first we present architectural contributions of our interconnect compared to FCB. Then, we evaluate the overall area, performance, and power for eAP with 8T and 2T1D and compare it with CA and AP. Then, we discuss scalability of eAP.

6.6.1 Interconnect Efficiency

In this section, we first compare overall architectural benefits of our proposed interconnect design, RCB, over the CA interconnect architecture, FCB. As we presented earlier in Section 6.2, RCB interconnect is a memory block of 96×96, whereas FCB is a memory block of 256×256, meaning that RCB consumes 7× fewer switches (or memory cells) than FCB, which reduces area overhead for the interconnect. RCB has also faster row cycle time because of shorter wires and consumes less power.

To study the applicability of RCB design in real-world and synthetic automata applications, we calculate the number of required RCB and FCB blocks for each application. The compiler iterates over the connected components (CCs) and checks if they can fit in a RCB switch block. If not, a FCB switch is needed to accommodate connectivity. In Table 6.1, we compare the number of required routing blocks of our interconnect approach, which is a hybrid of RCB and FCB, versus the baseline FCB, which is proposed in CA and assumes full connectivity for all the connected components.

As shown in Table 6.1, all the connected components in 17 out of 19 applications can entirely map to RCB blocks and no FCB block is needed. This means that when using RCB blocks, the total number of switches (memory cells) required for these applcaitons is 7.1× less than when using FCB blocks. This again confirms confirms that the FCB is extremely excessive for automata applications.

In Entity Resolution, there are many long distance loops, and none of the CCs can fit in the RCB switch block (Figure 6.2). In Snort, our interconnect accommodates most of the CCs in RCB blocks (only 19 FCB

[3] https://github.com/anonymousUser0/isca

Table 6.1: Comparison of our interconnect approach (hybrid RCB and FCB) with CA interconnect (FCB only). Our idea requires up to 7.1X fewer switches (memory cells) than CA.

Benchmark	#Transitions	#Connected Components	Baseline	Our Idea		Switch Reduction (times)
			#FCB	#FCB	#RCB	
Brill	62054	1962	168	0	168	7.1
Dotstar	94254	2837	378	0	378	7.1
EntityResolution	219264	1000	500	500	0	1
Fermi	57576	2399	160	0	160	7.1
Hamming	19251	93	47	0	47	7.1
Levenshtein	9096	24	12	0	12	7.1
PowerEN	40271	2857	160	0	160	7.1
Protomata	41635	2340	165	0	165	7.1
RandomForest	33220	1661	139	0	139	7.1
Snort	81380	5025	270	19	252	5
SPM	211050	2687	419	0	419	7.1
BlockRings	44352	192	192	0	192	7.1
Dotstar03	12264	299	49	0	49	7.1
Dotstar06	12939	298	50	0	50	7.1
Dotstar09	12907	297	50	0	50	7.1
Ranges05	12472	299	50	0	50	7.1
Ranges1	12406	297	50	0	50	7.1
ExactMath	12144	297	50	0	50	7.1
Bro217	2130	187	10	0	10	7.1

and 256 RCB), whereas the baseline uses 270 FCBs. Levenshtein is a difficult-to-route automata. The AP compiler can fit this benchmark in an AP chip with 48K states. However, the total number of states in Levenshtein is 2784. This implies that much of the STE and interconnect resources of an AP chip are wasted in order to deal with the routing congestion. However, in our interconnect model, we just need 12 RCB switches (9% routing resources of a eAP bank and 0.07% of routing resources on eAP 128 banks) to accommodate all the automata in Levenshtein.

Our compiler provides optimizations such as forcing constraints on the number of fan-in and fan-out of each node. Based on our sensitivity analysis, forcing each automata to have maximum fan-in and fan-out of 5 results in the minimum number of switches. The proposed interconnect optimization is general and can be applied to any memory-based interconnect, such as variations of gain cells or non-volatile memorty (NVM), where memory cells have non-destructive read property and can implement OR functionality for routing.

6.6.2 Overall Area Overhead

In this section, we discuss the area overhead of state matching arrays, interconnect arrays, and total overhead for supporting state capacity equivalent to 32K STEs (one eAP bank). Furthermore, we separate architectural contribution from technology contribution in our analysis.

A subarray size of 512 by 128 with 2T1D cell is fabricated in 65nm with area $0.085mm^2$ [38]. From the die image, we estimate the area for a block of 256 by 256 to be $0.084mm^2$ (60% of which is spent for memory cells and 40% is spent for decoder and sense amplifiers), which is $11mm^2$ to support 32K states. The projected area to support 32K states in 28nm is $2mm^2$. Therefore, the area of a 2T1D memory cell in 28nm is estimated $0.143\mu m^2$ and calculated as:

$$\frac{0.6\times2\times10^6}{32\times1024\times256} = 0.143\mu m^2 \tag{6.2}$$

In [103], Bhoj et. al presented two architectures for 2T1D cells in 30nm FinFET technology. According to their work, the area of a 2T1D memory cell is between $0.137 - 0.163\mu m^2$, which is consistent with our scaling assumptions.

Figure 6.7 shows the area overhead for state matching, interconnect, and total overhead of different architectures, assuming supporting 32K states. Compare to CA, eAP_8T reduces area overhead of interconnect ~4× (resulting from architectural contribution, i.e., RCB design) and eAP_2T1D reduces area overhead of interconnect ~8× (~4× resulting from architectural contribution, i.e., RCB design and ~2× resulting from technology choice).

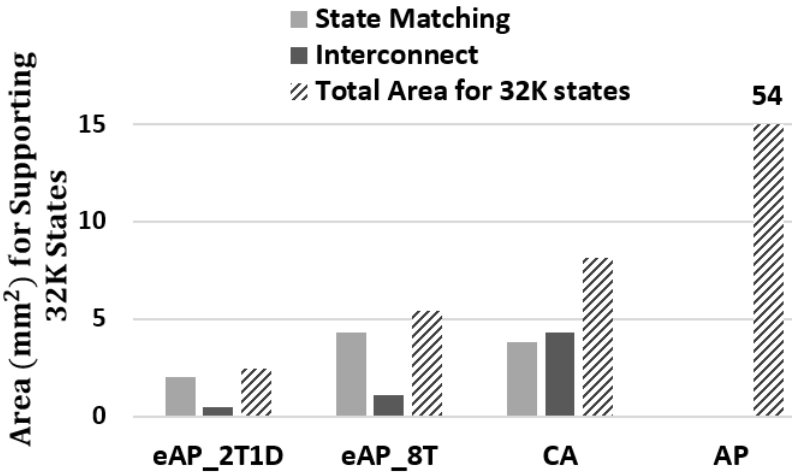

Figure 6.7: Comparing area overhead of eAP, CA, and AP normalized for 32K states all in 28nm. CA interconnect is ~4× higher than eAP_8T (architectural contribution) and ~8× higher than eAP_2T1D (~4× architectural contribution and ~2× technology contribution).

Overall area overhead (both state match and routing) of eAP_2T1D is 2.2×, 2.3×, 22× less compared to eAP_8T, CA, and the AP respectively, all in 28nm technology.

6.6.3 Overall Performance

Zhang et al. [38] report the read-cycle frequency of 6T SRAM array is twice that of a 2T1D gain cell array in 65nm technology. We assume a similar ratio in order to estimate the read-access frequency of a 2T1D array of size 256×256 (for FCB) in 28nm, using the read-access frequency of 6T SRAM array of size 256×256 in 28nm (which is 229 ps and calculated using standard SRAM compiler in nominal voltage 0.8V). In other work on 2T1D, Bhoj et. al [103] presented two architectures for 2T1D cells in 30nm FinFET technology. According to their work, a 2T1D memory array can operate at 2GHz, which is consistent with our assumption. Despite the area reduction in RCB (96×96), we still assume the worst-case delay for RCB to be the same as FCB.

CA proposes a sense-amplifier cycling technique and assumes 4× reduction in the read-access delay. However, sensing is just 25% of the total row-access delay. We re-calculated the delay in local and global switches in CA with best-case assumptions using an SRAM memory compiler. Fixing (1) switch delay calculation and (2) pipeline data-hazard problem in CA reduces the clock frequency from 2.2GHz to 1.43GHz. This correction has been verified with the authors.

Table 6.2: Pipeline stages delay. All designs are in 28nm.

Design	State-Match	L-Switch	G-Switch	Freq. Max	Freq. Operated
eAP_2T1D	500 ps	599 ps	599 ps	1.66 GHz	1.5 GHz
eAP_8T	349 ps	349 ps	349 ps	2.8 GHz	2.5 GHz
CA	438 ps	349 ps	349 ps	1.43 GHz	1.3 GHz
CA_opt	438 ps	349 ps	349 ps	2.2 GHz	2 GHz

Based on the SPICE simulation in CA, the wire delay is calculated as $66ps/mm$. Considering a cache slice of $3.19mm \times 3mm$, the switch delay is estimated as 99ps, assuming 1.5mm wire length. We assume the same wire delay for FCB and RCB in eAP_2T1D and eAP_8T (this is the worse-case assumption for RCB as it requires shorter wires).

Table 6.2 shows the delay for pipeline stages in CA and eAP in 28nm. As discussed in Section 6.3.2, in the new, CA-refined pipeline, L-Switch and G-Switch should be done sequentially in one stage, which means the pipeline delay is 698ps (349ps+349ps). In eAP optimized pipeline, L-switch (RCB) and G-switch (FCB) can be done in parallel. Therefore, pipeline delays for eAP_2T1D and eAP_8T are 599ps and 349ps respectively. Similar optimization proposed for eAP can be applied to CA (we call it CA_opt) which improves CA frequency from 1.43GHz to 2.2GHz. Therefore, the architectural contribution of our optimized pipeline improves the clock frequency of eAP (both 2T1D and 8T) ~2× and CA ~1.5×.

Like commodity DRAM, 2T1D cells also require periodic refreshes to retain stored bit values. The refresh operation is a sequence of dummy reads and write-backs to the memory rows. eAP_2T1D refresh time is 0.01%, which is calculated by dividing the time required for refreshing 256 rows (meaning 256 reads and 256

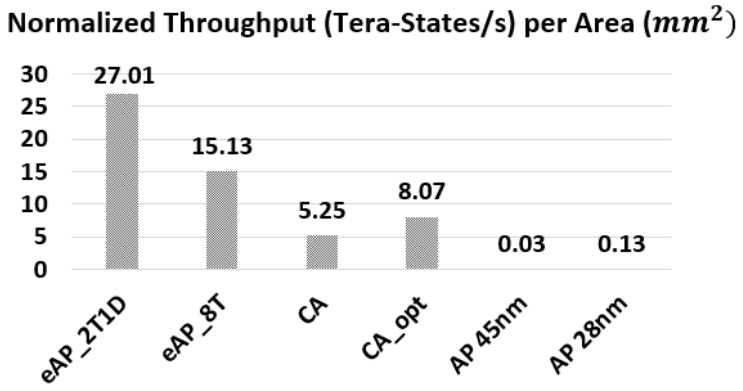

Figure 6.8: Comparison of throughput normalized per area. eAP_2T1D performs best due to its interconnect and technoloy benefits.

writes) by the retention time. Refresh is performed among all the subarrays in parallel and blocks the normal read/write operations.

6.6.4 Throughput per Unit Area

In the AP, CA, and eAP, each input symbol can be processed in one cycle. Therefore, they have a deterministic throughput of one input symbol per cycle, which is independent of input benchmarks. Another important metric in addition to frequency is state-matching capacity; if the capacity is not enough to accommodate all the automata in one iteration, several passes of the input stream, each with some reconfiguration overhead, are needed.

Figure 6.8 represents the throughput of different architecture normalized to area. The throughput here is defined as the number of states that can run in parallel multiplied by clock frequency (Tera-states per second). The AP is based on 45nm technology and operates at 133 MHz frequency, while CA and eAP are based on 28nm. To compare the different architectures in the same semiconductor technology node, we also show technology projection of the AP on 28nm.

Overall, eAP_2T1D achieves 1.7×, 5.1×, 3.3×, and 210× better throughput per unit area over eAP_8T, CA, CA_opt and the AP respectively, all in 28nm technology. As expected, eAP_8T has 1.8× better throughput per area over CA_opt. CA design uses 6T arrays of size 256×256 for state matching and 8T arrays of 280×256 for interconnect, and thus, the interconnect overhead is more than 50%. eAP_8T adopts 8T arrays for both state matching (of size 256×256) and interconnect of size (96×96), thus as previously mentioned, the interconnect overhead is ~4× less than state matching resources.

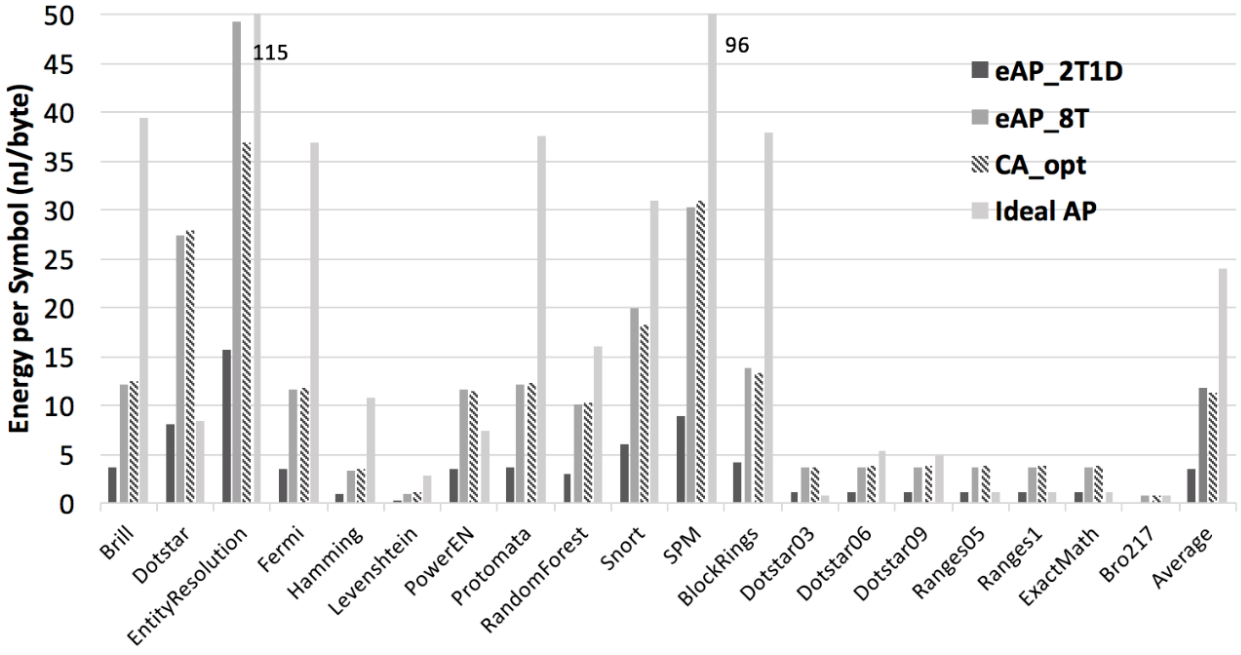

Figure 6.9: Overall energy consumption of eAP_2T1D compared to eAP_8T, CA_opt, and ideal AP.

6.6.5 Energy/Power Consumption

This section discusses the energy and power consumption of eAP_2T1D and eAP_8T and compares to CA_opt and the AP. To calculate energy consumption, we need to know (1) the number of active partitions for state-matching and switch blocks, and (2) the number of transitions between local switches to consider for the energy consumed driving wires.

Note that it is not possible to power-gate state-matching memory arrays on a cycle-by-cycle basis. In order to power-gate these subarrays, it is necessary to know the potential next states beforehand. However, in the pipeline, the state matching results and next potential state are calculated simultaneously, which prevents the power-gating. (One can still power-gate an array that is unoccupied.) This observation is not considered in CA. We update the energy/power results in CA paper [29] based on this observation.For the AP, we adopt the *ideal AP* model presented in CA. All the statistics per cycle are extracted from our compiler.

Static power consumption of eAP_2T1D system consists of two main components: (1) the leakage current of the cell itself and (2) the refresh power to keep the data alive. The refresh power of 2T-based gain cells is the dominant portion of static power [38]. Moreover, the static power of 2T1D memory array is 20% of static power in 6T SRAM array. We use the same ratio to calculate static power for eAP_2T1D. We estimate the dynamic energy consumption for RCB and FCB 8T blocks using a standard memory compiler.

Figure 6.9 shows the energy per input symbol for eAP_2T1D, eAP_8T, and CA_opt on 28nm, and ideal AP model. We can observe that benchmarks with larger number of states, such as Entity Resolution, Dotstar,

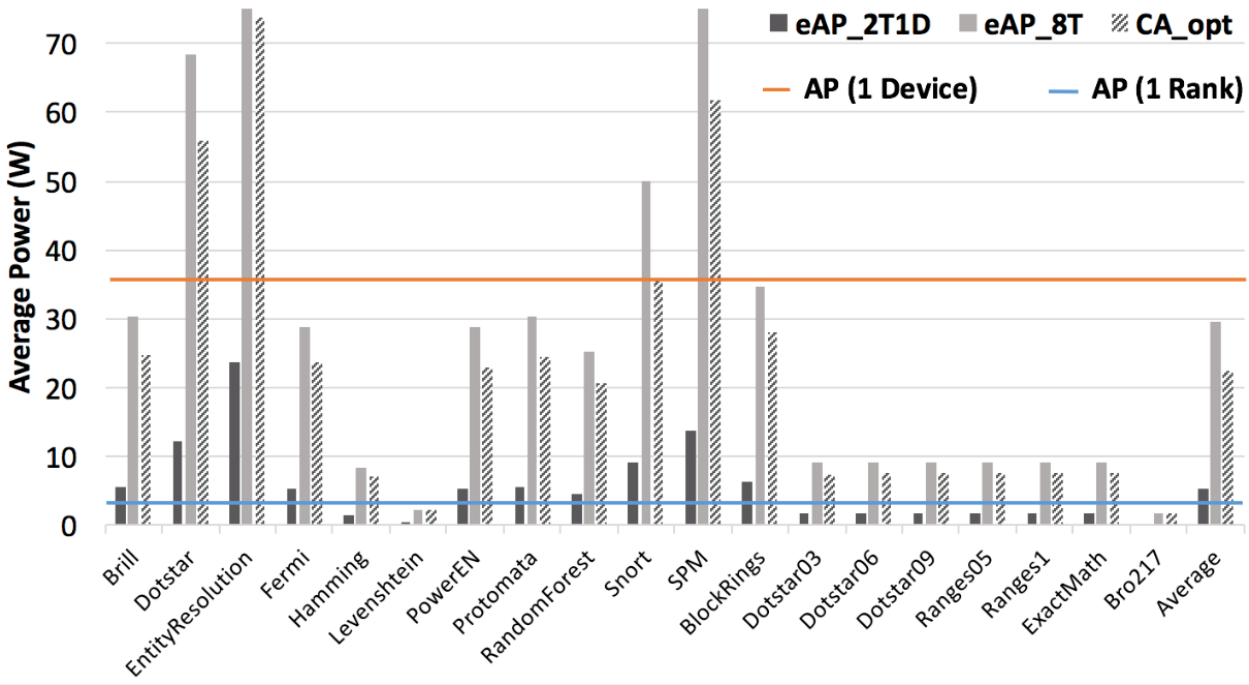

Figure 6.10: Overall power consumption of eAP_2T1D compared to eAP_8T, CA_opt, and the AP (reported by Micron).

Snort, and SPM, consume higher energy. This is because these benchmarks have utilized more state matching and switch arrays to accommodate larger number of states. Furthermore, Entity Resolution cannot utilize lower-energy RCB resources for the local interconnect (as shown is Table 6.1) and needs to use FCB, which results in higher energy consumption. Overall, the energy consumption of eAP_2T1D is about 3× less than eAP_8T and CA_opt. Energy efficiency of eAP_2T1D comes from its density and a compact RCB design, which results in consuming lower dynamic energy due to shorter wires and smaller number of switches.

Figure 6.10 shows the average power consumption across benchmarks. The power consumption of eAP_2T1D is 5.4× and 4.1× less compared to eAP_8T and CA_opt respectively. As expected, the power of the eAP_2T1D is the highest, because it has fastest clock speed.

6.6.6 Performance Scaling with Application Size

In Section 6.6.4, we discussed the throughput per unit area. In this section, in order to show the effect of larger benchmarks on performance, we increase the number of automata in the ANMLZoo benchmark up to 1024× and study two power-constrained and non-power-constrainted scenarios. In non-power-constrainted scenario,

In this section, we study the scalability of different designs in two we assume CA, CA_opt, and eAP_8T can utilize the 40MB L3 cache [29], which is equal to accommodating 1280K STEs. The Intel 4^{th}-generation

Core processor (Haswell and Broadwell) has a 0.5Gb/1Gb embedded memory die connected to CPU as L4 cache [113, 116]. For eAP_2T1D, we assume 1Gb of embedded memory with 128 banks. Therefore, eAP_2T1D can support up to 4096K STEs

Table 6.3 summarizes the key properties of eAP_2T1D relative to eAP_8T, CA, CA_opt, and AP. In short, eAP_2T1D has a density advantage compared to other designs. When the area allocation for automata processing is small enough that the total power is not a limiting factor, the density advantage will apply. At some point, enough area is allocated that power becomes a limiting factor. Then eAP only has a capacity advantage.

Table 6.3: Summary of different memory-based automata architectures (for 32K states, including interconnect blocks)

	eAP_2T1D	eAP_8T	CA	CA_opt	AP
Freq. (GHz)	1.5	2.5	1.3	2	0.133
Power (W)	4.15	29.69	22.57	14.69	2.6
Area (mm^2)	2.47	5.41	8.12	8.12	140

However, some benchmarks in ANMLZoo represent just a portion of actual applications (normalized to fill one AP chip). While one bank is enough for regex-based applications such as Snort, Brill, and Dotstar, which will not require much power, the density advantage will pertain; but other applications require orders of magnitude more states. This will then require multiple passes over the input, with each pass implementing a portion of the overall automata set. In such cases, reconfiguration overheads will apply, and as mentioned, this is more costly for CA.

Figure 6.11 shows the performance of CA, CA_opt, eAP_2T1D, and eAP_8T averaged on ANMLZoo, normalized to the AP performance, with and without power constraints.

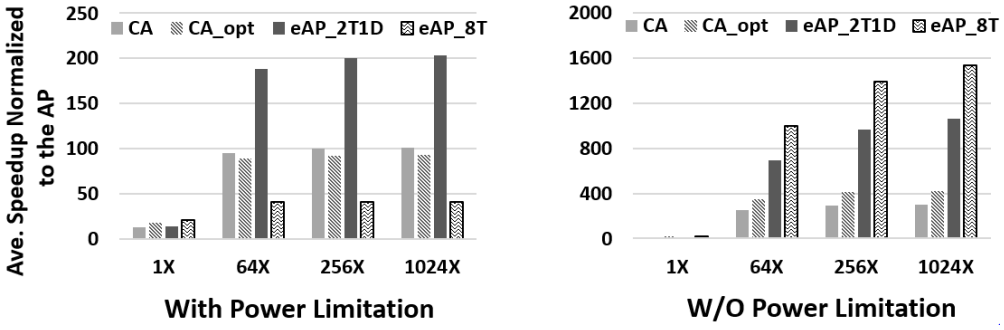

Figure 6.11: Performance scaling with benchmark size

In the non-power-constrained scenario, we assume CA and CA_opt can utilize the whole 40MB L3 cache in their design [29], which is equal to accommodating 1280K STEs. In eAP designs, we assume the whole 128 banks are utilized, which can support up to 4096K STEs (See Section 4). In this scenario, the relationship

among the designs follows Table 6.3, except for the additional factor of reconfiguration overhead, so the speedup of eAP_8T is 5×, 3.6×, and 1.4× over CA, CA_opt, eAP_2T1D respectively. This is because eAP_8T has the highest clock speed.

In the power-constrained scenario, we assume the maximum power of 75W for all the designs. This in turn reduces the allowable number of active processing blocks. eAP_8T has 1.6×, 1.1×, and 1.4× better performance over CA, CA_opt, eAP_2T1D on the original-size benchmarks in ANMLZoo (1X), because eAP_8T has higher frequency than others. However, when increasing the benchmark size, hardware reconfiguration and multi-processing of the input stream become a limiting factor for CA and CA_opt (due to less capacity and lower frequency) and eAP_8T (due to high power consumption) - see Equation 6.1.

eAP_2T1D shows up to 2×, 2.1×, and 4.9× better performance over CA, CA_opt, eAP_8T when increasing the benchmark-size up to 1024×. This is because eAP_2T1D has higher density and lower power consumption. The performance benefits of eAP increase when processing larger automata benchmarks. Furthermore, the advantages of eAP_2T1D over CA, CA_opt, and eAP_8T increase when increasing the input size.

6.7 Conclusions

In this paper, we propose eAP, a high-speed, dense, and low-power reconfigurable architecture for automata processing. We exploit inherent bit-level parallelism in memory to support multiple concurrent transitions in NFA and utilize subarray-level parallelism in memory to process thousands of automata in parallel. Motivated by connectivity patterns in the real-world automata benchmarks, we propose a *reduced* crossbar interconnect for state transitions, which compacts the switch patterns in a full crossbar interconnect and provides a 7× reduction in the number of switches. This in turn reduces power consumption and delay due to shorter wires. Overall, eAP presents 5.1× and 207× better throughput normalized to area compared to the previously designed in-memory automata accelerators, Cache Automata (CA) and the Automata Processor (AP) respectively. Benefits of eAP are even higher for applications that require multiple passes.

Chapter 7

FlexAmata: A Flexible Automata Processing Engine

Because regular expressions have most commonly been used for text, packet, and other byte-oriented processing, existing automata accelerators are designed based on a fixed 8-bit (ASCII) symbol processing scheme, similar to software solutions [32, 33]. This means that the automata structure is based on 8-bit symbols and an 8-bit input is processed in each cycle. However, we observed that the fixed-size symbol processing could be a source of area and throughput inefficiencies and it also limits the general adoption of applications for automata processing.

In memory-based solutions, symbols are encoded in memory columns, such that each symbol activates a different row of memory. This tends to reinforce designs based on 8-bit symbols, because 256 (2^8) is a fairly conventional subarray height. However, this can be extremely inefficient, especially when the application alphabet (symbol-set) size is very small and the number of rows in a subarray is more than required. For example, in genomics, the alphabet is A, T, C, and G, and a 2-bit automata organization with only 2^2 rows is enough to perform the string matching. This is 64× smaller than what existing spatial architectures provide!

On the other hand, the 8-bit symbol processing architectures can limit the generality of the architecture for applications that have more than 256 symbols. For example, in sequential pattern mining [7], the input database can be quite large, such as the product database from Amazon, and the number of unique items (or symbols) can be very large (on the order of 2^{20} or more). In formal verification problems, the symbols map to the events, and thus, the automata symbol-set size can be extremely large [117, 19]. However, due to delay, power issues, and signal integrity, it is impractical to change the hardware to support 2^{20} rows in each memory subarray. Moreover, simply daisy-chaining multiple states to support larger alphabets result in false

report generation.

One more problem with the existing 8-bit approach for spatial accelerators is that, if the memory subarray size of the underlying memory technology changes, then there is a need to make sure that the application symbol-set size is still compatible to the memory architecture. For example, Cache Automaton [29] re-purposes caches in conventional processors for automata processing. If the number of rows in the subarrays of a cache structure changes, then the automata structure and input bitwidth consumption need to be changed for correct functionality and full hardware utilization.

One advantage of using custom in-memory solutions or FPGAs as automata accelerators is that the processing bitwidth can be customized for the application need. In this paper, we aim to answer the following questions. What are the necessary hardware/software modifications to efficiently support very large or very small alphabets on spatial architectures? How can an application make better use of existing hardware for automata acceleration? What is the best bitwidth size for automata processing on spatial platforms? How to design next-generation automata accelerators with higher throughput? To the best of our knowledge, this is the first work that explores these research questions.

To answer these questions, we propose FlexAmata, a software solution that transforms an automaton structure with arbitrary symbol alphabets to support different bitwidth processing. FlexAmata accepts an $m-bit$ processing automaton as the input and then, (1) generates the binary automaton and applies several minimization techniques, and (2) converts the binary automaton to an $n-bit$ processing unit, where n can be larger or smaller than m, depending on the target architecture. Thanks to the fine-grain, bit-level optimizations in FlexAmata, state and transition overhead of a transformed automaton is reasonably low.

FlexAmata offers arbitrary bitwidth processing, thus improving efficiency for small alphabets, enabling hardware acceleration for large alphabets that were nearly impossible to process efficiently up till now, and maintains application compatibility with the future automata hardware accelerators. Furthermore, FlexAmata can improve efficiency and provide large-symbol-set compatibility even for conventional in-memory solutions such as Cache Automata. By analyzing FlexAmata on several automata applications from the ANMLZoo [2], AutomataZoo [118], and Regex [100] benchmark suites, we propose in-memory and FPGA solutions for the most efficient bitwidth processing.

Figure 7.1 shows throughput per unit area of our two solutions across different bitwidths. In summary, we find that 4-bit processing has 2.3× higher throughput per unit area than 8-bit processing in in-memory architectures, and 16-bit FPGA solution has 2.5× higher throughput per unit area than 8-bit FPGA solution. Moreover, in-memory architectures perform at least two orders of magnitudes better than FPGAs. The area efficiency introduced by FlexAmata provides an opportunity to design denser state-matching resources, which can accommodate more states and results in fewer spatial resources.

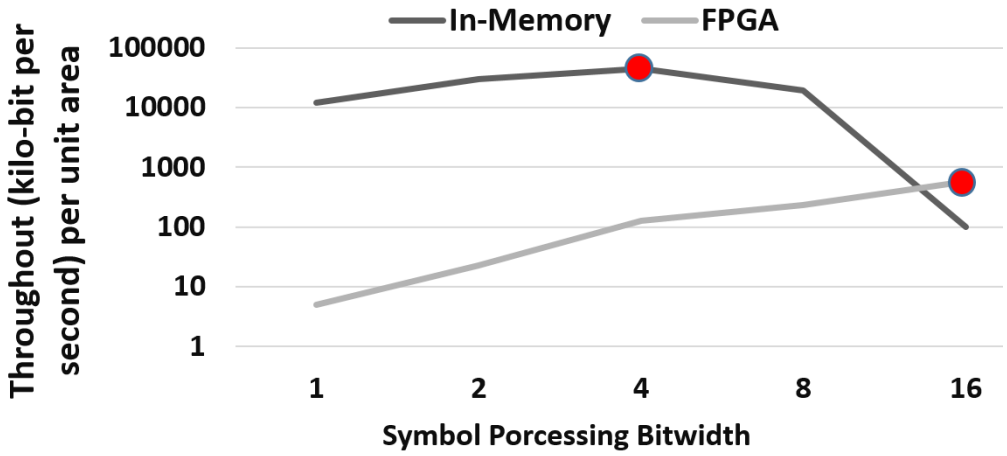

Figure 7.1: Comparing in-memory and FPGA solutions in different bitwidths.

This paper makes the following technical **contributions**:

We present FlexAmata, a compiler solution to provide *application compatibility* with existing and future memory-centric automata processing architectures, and *hardware compatibility* for existing application by transforming automata structure. The former allows execution efficiency and feasibility of applications with very small or large alphabets on spatial accelerators.

We propose area efficient and high-throughput in-memory architectures and FPGA automata processing engine and explore various bitwidth automata processing using FlexAmata. Our exploration introduces hints on how to design an automaton for more efficient hardware mapping and insights for next-generation automata processing accelerators.

We present an open-source toolkit for automata simulation, minimization, transformation, performance modeling on memory-centric architectures, and performance evaluation on FPGAs.

7.1 FlexAmata

FlexAmata transforms an *m-bit* (m is 8 for ASCII) automaton A to an *n-bit* automaton B, where n can be larger or smaller than m. This transformation is done in two steps; (1) converting A to a bit-level representation (A_b), and (2) generating automaton B by transforming A_b to process *n-bit* in each cycle. To generate the *n-bit* automaton, we find all the unique paths of size n in A_b and represent them as a single edge (or equivalently transition rule) in B. The algorithm performs bit-level minimization on the automata and merges the states and transitions in binary path when applicable. Finally, automaton B is converted to its homogeneous representation to properly be configured on an in-memory or FPGA accelerator.

In Figure 7.2, we explain how an 8-bit automaton is transformed into 3-bit and 4-bit automata. In the notation STE_x^y, x is state index and y is the bitwidth size. The original homogeneous automaton (a) has two states and accepts language $(A|B)C^+$. Using FlexAmata, we generate binary automata (b) and minimize the states when possible. For example, the first 6 bits of symbol A and B can be merged. Then, 3-bit (c) and 4-bit (d) are generated from the bit-automaton.

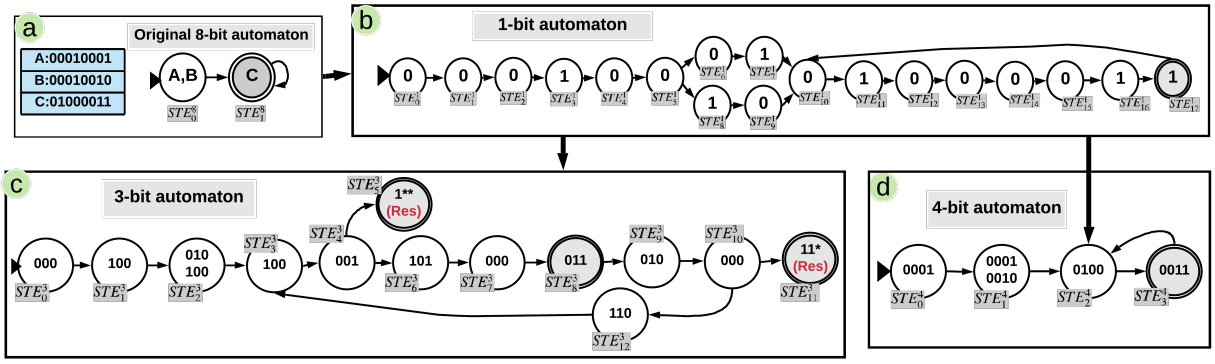

Figure 7.2: An 8-bit automaton (a) is converted to the minimized *1-bit* automaton (b). The *3-bit* (c) and *4-bit* (d) automata are generated from the *1-bit* automaton.

In the 3-bit automaton, STE_0^3 is an start state and STE_5^3, STE_8^3, and STE_{11}^3 are final states. Each state processes one or more 3-bit symbol. STE_{16}^1 in bit-automata is equivalent to reaching the state STE_4^3 in the 1-bit automaton. STE_{17}^1 is a report state and there is a loop back to the state STE_{10}^1. Assume STE_4^3 is an active state, and the next input character is "1", then it should generate a report. Equivalently in 3-bit, STE_4^3 is an active state, and because the next input is 3-bit, then, "1**" should generate a report. In order to address this unalignment in the bitwidths, we generate residual states (Res) to report when a match happens in the middle of a multi-bit input. STE_5^3 is a residual state that reports when the matching happens in the first bit of the 3-bit input. The residual states can be used to find the exact location where a match is happened.

The generation of the 4-bit automaton is straight-forward and no residual state is needed. This is because 8 (from original 8-bit automaton) is divisible by 4, and this avoids the unalignment in input characters. In order to get the correct functionality, we always make sure the input size is divisible by the bitwidth size by padding input streams.

Non-divisible bitwidths effect: as Figure 7.2 illustrates, the 3-bit automaton has more state and transition overhead than 4-bit. From our experiments, we observed that when re-shaping an n-bit automaton to an m-bit automaton, the overhead would be minimum if either $m \bmod n = 0$ or $n \bmod m = 0$. Figure 7.3 explains the reason with one state in an automaton with four 8-bit symbols. First, 1-bit automaton is generated. Then, 3-bit and 4-bit are generated from 1-bit. In 3-bit, because 8 mod 3 ! = 0, it needs to generate

more states and transitions to consider for combinations of paths when a jump is needed. The overhead of non-divisible bitwidths is more severe for automata with more symbols sets and complex interconnect typologies.

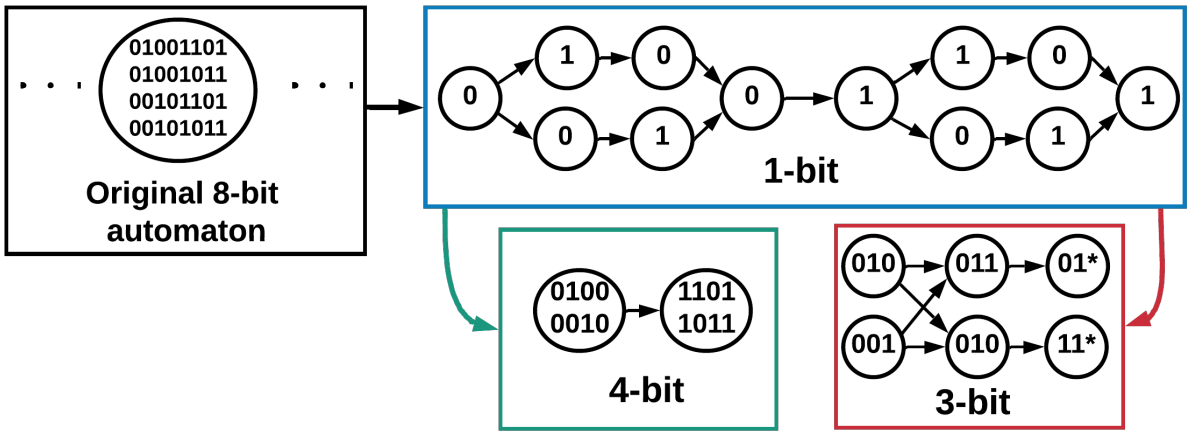

Figure 7.3: State and transition overhead is less in divisible bitwidths (4-bit) than non-divisible bitwidths (3-bit).

Interval effect: We observed that the applications with the states containing symbol ranges have more state and transition overhead when transforming to a different bitwidth. For example, if an STE matches on $[a-f]$, it accepts a range of symbols and can be activated with any of these characters if its parent(s) is in an active state. When constructing the 1-bit automaton, these characters are potentially split into different states to preserve the correctness of matching. This is an extra source of state and transition overhead when transforming the automata to process different bitwidths.

When symbols of a state are any character except one specific character, we call it a negation operation. Assuming an application with 256 symbols, a negation operation can accept 255 characters, and need to be split when transforming automata. Figure 7.4 demonstrates an example of how to avoid negation operation when possible. Both automata accept language $a[\hat{}b]^*b$, meaning that any pattern that contains sequence "ab" with any character between the occurrence of 'a' and 'b' is a match. The left-side design has one active transition in each cycle, but when generating the 1-bit automaton, the state with symbol $[\hat{}b]$ generates up to 255×8 1-bit states. The right-side design has the same functionality as the left-side one, and can have up to two active transitions. However, when generating the 1-bit automaton, the state with the $*$ symbol generates only 8 1-bit states with symbol '*', which means each of 1-bit states can be either '0' or '1'. Therefore, designing an automaton with a '*' symbol rather than a negation symbol (when possible) can utilize the parallel transitions resources for NFA processing and reduce the state and transition overhead significantly.

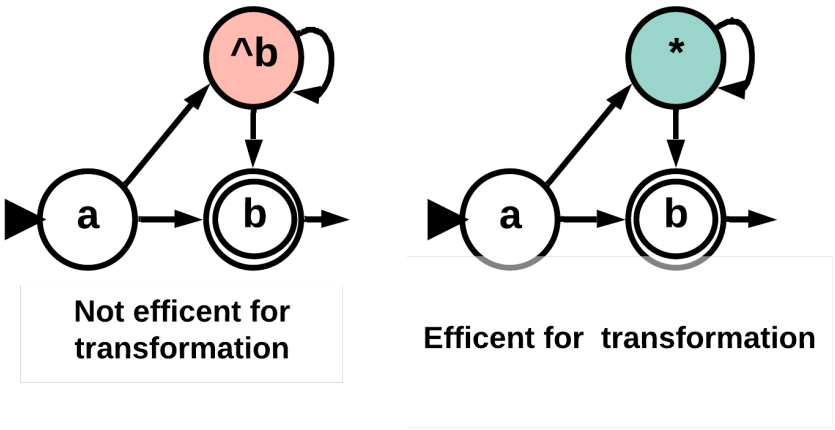

Figure 7.4: Both automata detect language $a[\hat{}b]^*b$. The right one results in a more efficient automata transformation.

We found that 6 out of 20 applications have a large number of negation operations. To reduce the overhead of transformation, we redesign the automata benchmarks by changing the negation symbols with the '*' symbols when possible, verifying that the change does not alter the semantics of the automata. Our experimental results show that this change reduces the transformation overhead significantly (more discussion Section 7.3.1). This observation can help application developer to design regexes or automata more efficiently for this purpose.

7.1.1 Application Implications and Software Optimizations

The existing automata processing accelerators are designed based on 8-bit symbol processing scheme. However, the applications can have a very small or large alphabets (symbol-set). FlexAmata provides software compatibility for the existing automata hardware accelerators, meaning that if the application has small symbol-set, then FlexAmata can generate the 8-bit automata to fully utilize the hardware. On the other hand, if the application has a very large symbol-set, FlexAmata transforms the automata to 8-bit automata, which provides feasibility support for the application.

Utilization for smaller symbol-sets: Figure 7.5 shows an example of how FlexAmata improves utilization and throughput of an application with small symbol-set on an existing 8-bit automata accelerator.

The original automaton has four symbols, A, T, C, and G. Therefore, 2-bit is enough to encode the symbols. Directly processing the automata on an 8-bit in-memory automata accelerator wastes the state-matching resources 64×. This is because in the 8-bit scheme, state symbols are encoded in a 256-bit memory column,

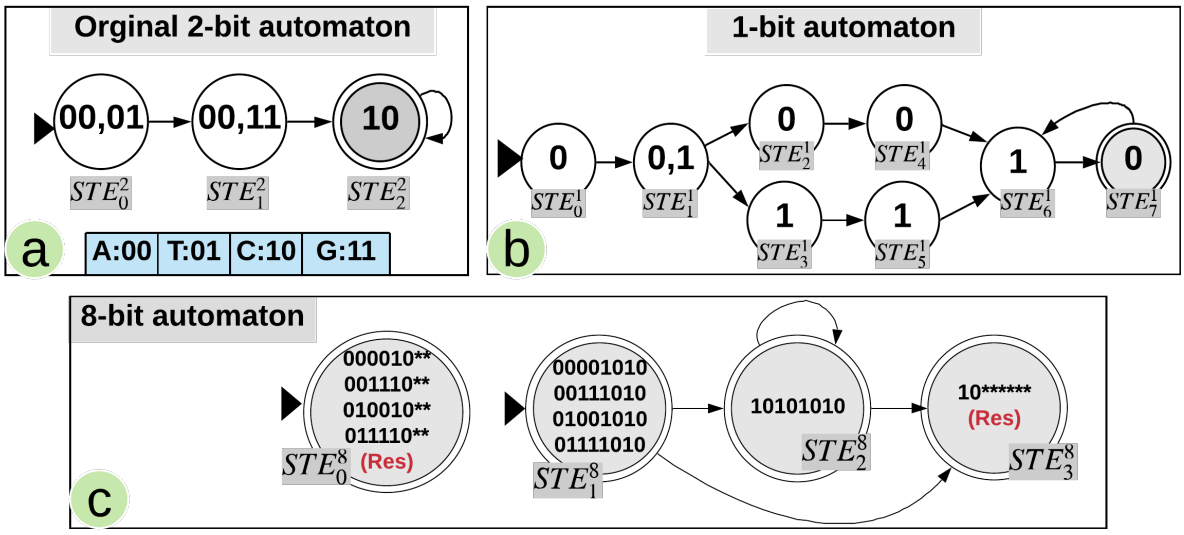

Figure 7.5: (a) Original automata has 4 symbols and can be represented with two bits. (b) FlexAmata generates 1-bit from original automaton. (c) Then 8-bit is generated from 1-bit, and can process 4× more symbols at a minimal state and transition costs compared to the original automaton.

however, the 2-bit application only needs 4-bit memory column. To fully utilize the 256-bit memory column, we generate the 8-bit automaton from the 2-bit original automaton. This is done by generating the 1-bit automaton. In this example, the 8-bit automaton has 25% more state 25% more transitions compared to the original 2-bit automaton. However, it processes 4 2-bit symbols in each cycle. This means that this transformation can increase the throughput of the application 4× with minimal resource overhead, totally based on a software solution (no hardware modification is needed).

Feasibility for larger symbol-sets: As discussed, there are some applications that have very large symbol-sets. For example, in natural language processing, each word can be a symbol. In pattern mining tasks, an entity can be an item in Amazon or name of a person. Clearly, increasing the memory column size is not a feasible and cheap solution. Moreover, simply breaking a state with 16-bit symbols to 2 states with 8-bit symbols (or in other word, daisy-chaining two symbols) results in false report generations when one state accepts more than one symbol. We explain the problem using Figure 7.6. For simplicity, assume the architecture A supports 2-bit symbols, but the application has 16 symbols and requires the support for 4-bit processing.

The left-side automaton (a) accepts $(0011|1100)^+$. To process this automaton in a 2-bit architecture, chaining to two states introduces false positives. (b) breaks the STE_0^4 to two states STE_0^2 and STE_1^2. However, it accepts $(0011|1100|0000|$
$1111)^+$. (c) To preserve the semantic of original automata, FlexAmata splits the states where chaining the

states causes false positive. Several minimizations are applied on the 1-bit automaton to reduce state and transition overhead when splitting the states.

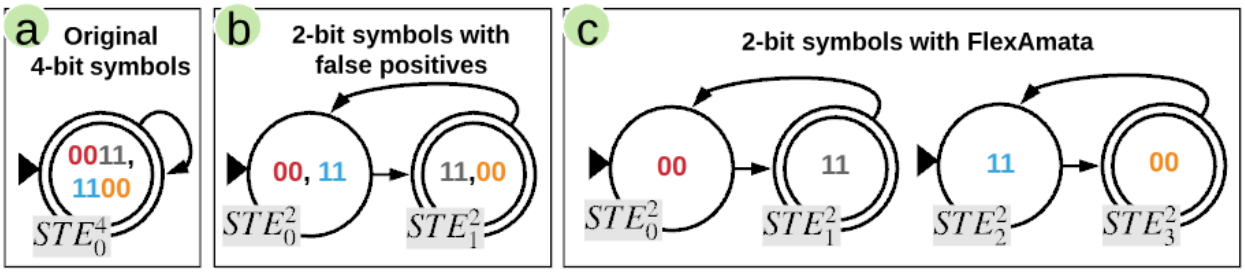

Figure 7.6: The problem with chaining two symbols to support larger symbol-sets.

7.1.2 Hardware Implications

To better understand the effect of different bitwidths on automata and its hardware implications, we ran FlexAmata across a diverse set of 20 applications from [2] and [100]. Figure 7.7 shows the average number of states and transitions in different bitwidths normalized to the number of states and transitions in the original 8-bit automata in log_2 scale. These averages mask some important individual application behaviors that are addressed later in the paper.

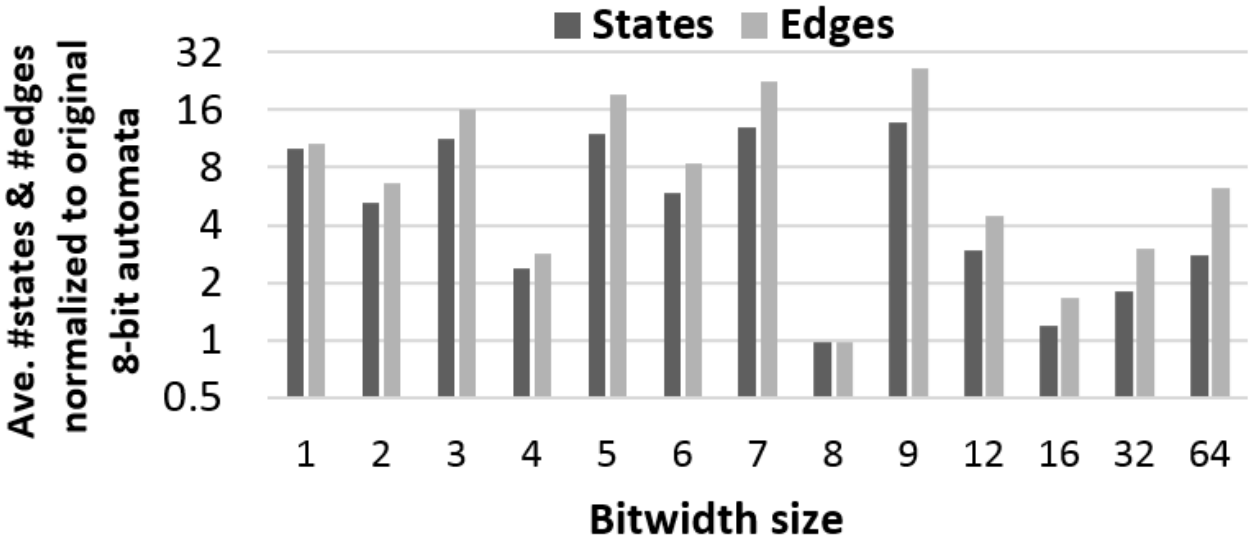

Figure 7.7: Average increase in the state and transition count for different bitwidths normalized to original 8-bit automata.

2-bit and 4-bit designs process one-fourth and half of an 8-bit symbol in each cycle, and incur 5.2× and 2.4× state overhead, and 6.6× and 2.8× transition overhead on average, compared to the 8-bit design,

respectively. However, the 8-bit design requires memory subarrays with 256 rows (see Section **??**), while 2-bit and 4-bit designs only need memory subarrays with 4 and 16 rows, respectively.

16-bit design incurs only 1.2× state overhead and 1.6× transition overhead on average, and propose higher processing rate. However, it is very costly and inefficient to encode a 16-bit symbols to a memory column with 2^{16} rows. On the other hand, 16-bit symbols can be efficiently store in look-up-table resources on FPGAs. The larger bit-width processing (e.g., 32-bit, 64-bit, etc.) causes dramatic increase in the number of symbols (e.g., up to 2^{32} symbols in 32-bit processing). We leave exploration for very large bitwidths to future work.

Based on these observations and different properties of automata in each bitwidth, we investigate different bitwidth automata processing on spatial architectures and find the best bitwidths size for each architecture. We present and evaluate in-memory solutions for 1-bit, 2-bit, 4-bit, and 8-bit automata processing, and we call them reduced bitwidth designs (RBDs). The results show that 2-bit and 4-bit performs better than 8-bit processing. We then present and evaluate a reconfigurable FPGA solution for different bitwidth processing (2, 4, 8, 16). The results show the middle-sized bitwidths (such as 8-bit and 16-bit) work best on FPGAs. Finally, we compare our solutions with other spatial architectures.

Reduced Bitwidth Designs

NFAs for real-world automata applications are typically composed of many independent rules or patterns, which manifest as separate *connected components (CCs)* with no transitions between them. Each connected component has usually a few hundred states. All the connected components can thus be executed in parallel, independently of each other.

Figure 7.8(a) represents the 2-stage pipeline architecture of a 4-bit automata processing unit, which can process a connected component with up to 256 states and any connectivity pattern. In the state matching phase, the 4-bit input is decoded as the input of the SRAM-based memory subarray. The state whose labels matches the input is read to the row-buffer and stored in the match vector.

In the state transition stage, the potential next states (the states that are connected to the current active states), are discovered through the local switches. Finally, bitwise AND operation of the potential next states and match vector recognizes the states that (1) are matched with the current input symbol and (2) their parent(s) were an active state(s) in the previous cycle.

To support connected components of larger size, which is especially needed when processing a different bitwidth automaton, we design a hierarchical interconnect to connect local switches through a global interconnect. Global switches allow 64 states in each automata processing unit to communicate with 64 states in three other automata processing units. Figure 7.8(b) shows an overview of the reduced bitwidth architecture, which allows a 4-bit connected component with up to 1024 states. Both local and global

interconnects are full-crossbar and support full connectivity in an automaton, meaning there can be an edge between every two states. A switch in the crossbar is modeled with an 8T SRAM memory cell, same as Cache Automaton interconnect design [29]. The transition overhead from an automaton transformation translates to a more switch utilization in the full-crossbar design, thus does not incur a resource overhead.

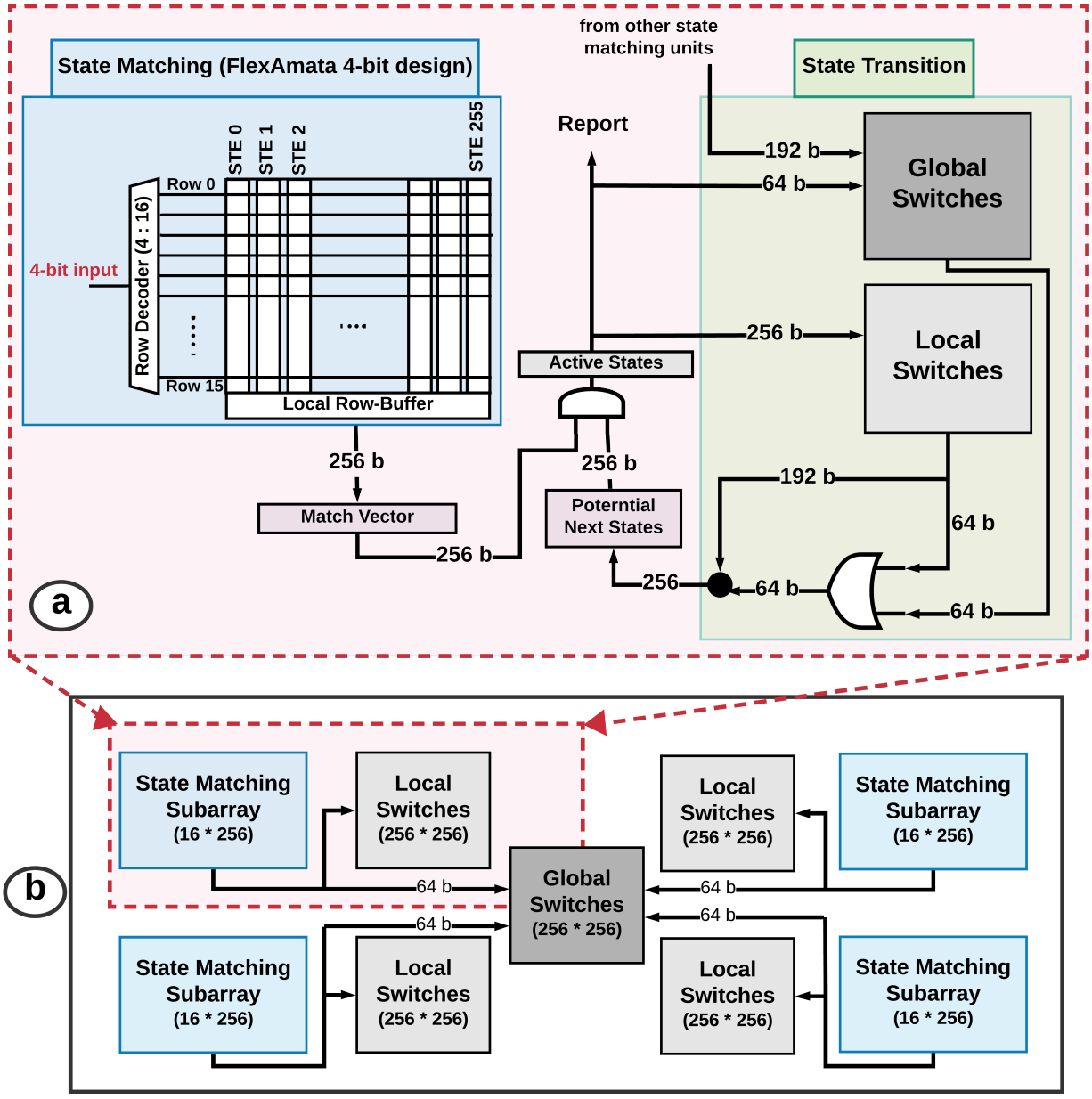

Figure 7.8: (a) A 4-bit automata processing unit, (b) Using 2-level switch structure to support larger automata.

The 1-bit and 2-bit designs would be similar to Figure 7.8, but with state matching subarrays of 2×256 and 4×256, respectively. Compared to the 8-bit design, the subarray size decreases 128×, 64×, and 16× for 1,

2, and 4-bit designs, respectively. Moreover, the memory decoder size, the memory access latency, and energy consumption for accessing state matching subarrays of 1, 2, and 4-bit designs decrease accordingly.

Memory technologies such as reduced latency DRAMs (RLDRAM) [101] have smaller column sizes in each subarray to achieve higher memory access rate. The efficiency of *4-bit* automata processing architecture introduces the potential of alternative use of memory technologies (e.g., RLDRMAs) for automata processing.

FPGA

FlexAmata transforms the automata to the target bitwidth, and then, generates the HDL code for FPGA backends. This section discusses the automata processing engine on the FPGA to highlight the insights of processing variable bitwidths on this target platform.

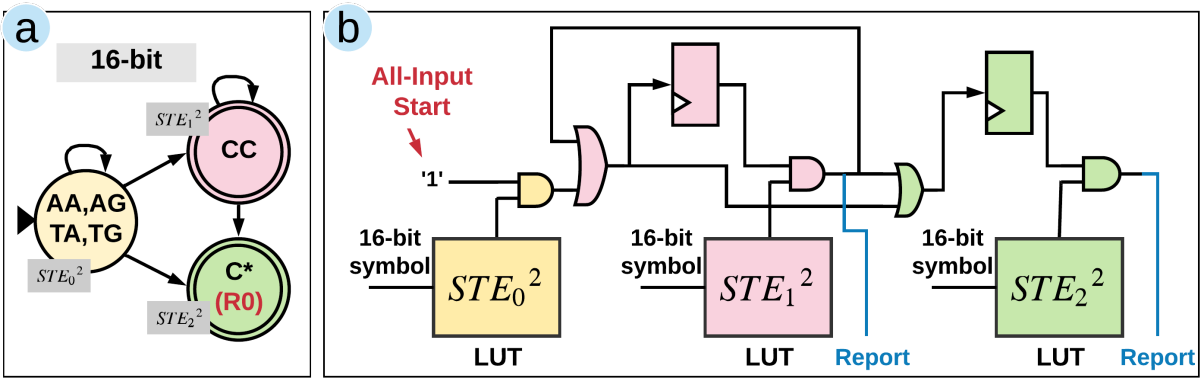

Figure 7.9: Mapping an automaton to FPGA resources.

In Fig. 7.9, a simple homogeneous automaton has been shown which process two symbols (16-bit) in each cycle. States have been color-coded to represent their equivalent units in a circuit. Symbol matching is being done entirely in LUTs based on the 16 bits input symbol. Theoretically, Flip-flops (FFs) are equivalent to potential next state registers in Figure 7.8 and they represent that a state is potential to be active in the next cycle if the next input symbol matches the matching conditions.

The input signals of the FFs come from an OR gate, which is the OR signal of all the states that have incoming transitions to that specific state (parent states). This is compatible with our previous definition, where a state is activated, all of its children are considered as potential active states. The states that have common set parents can share their FFs and save hardware resources. However, in theory, their corresponding states cannot be merged since they are not equivalent states. Same as Figure 7.8, the report signals of the final states are generated from the AND gate of matching signals and potential active states.

We observed that small bitwidths does not utilize FPGA LUT resources well. This is mainly because LUTs can implement up to two functions with one input, and thus, store the 1, 2, and 4-bit symbols inefficiently on

6-bit LUTs. On the other hand, processing more symbols per cycle leads to a more complex matching with many intervals from different states combined to a single state. This situation makes matching using 6-inputs LUTs inefficient in terms of resource usage and clock frequency (longer critical path). We observed that the middle-sized bitwidths can efficiently utilize the resources and achieve decent performance.

7.2 Evaluation Methodology

NFA workloads: We evaluate our proposed claims and architectures using ANMLZoo [2], AutomataZoo [118], and Regex [100] benchmark suites. They represent a set of diverse applications including machine learning, data mining, and network security. We present a summary of the applications in Table 7.1, including the number of states and transitions in each benchmark as well as the average degree (the number of incoming and outgoing transitions) for each state, the total number of loops in each benchmark, and symbol density. Symbol density shows the average number of symbols per state, and is calculated by dividing the total number of symbols over the total number of states in each benchmark. We later show that the higher state density causes higher state and transition overhead when transforming an automaton.

Table 7.1: Benchmark Overview

Benchmark	**#Family**	**#States**	**#Transitions**	**Ave. Node Degree**	**Symbol Density**
Brill [2]	Regex	42658	62054	2.90	52.2
Bro217 [100]	Regex	2312	2130	1.84	1.8
Dotstar03 [100]	Regex	12144	12264	2.01	3.1
Dotstar06 [100]	Regex	12640	12939	2.04	4.8
Dotstar09 [100]	Regex	12431	12907	2.07	6.7
ExactMath [100]	Regex	12439	12144	1.95	1
PowerEN [2]	Regex	40513	40271	1.98	5.8
Protomata [2]	Regex	42009	41635	1.98	116
Ranges05 [100]	Regex	12621	12472	1.97	1.2
Ranges1 [100]	Regex	12464	12406	1.99	1.2
Snort [2]	Regex	100500	81380	1.61	7.5
TCP [100]	Regex	19704	21164	2.14	10.1
Hamming [2]	Mesh	11346	19251	3.39	113
Levenshtein [2]	Mesh	2784	9096	6.53	1
EntityResolution [2]	Widget	95136	219264	4.60	47
Fermi [2]	Widget	40783	57576	2.82	7.1
RandomForest [2]	Widget	33220	33220	2	179
SPM [2]	Widget	69029	211050	6.11	26.5
BlockRings [2]	Synthetic	44352	44352	2	1
CoreRings [2]	Synthetic	48002	48002	2	1

Experimental setup: To calculate area, power, and clock cycle for RBDs, we use CACTI 7.0. We

assumed a 4MB SRAM-based memory with eight banks on 22nm technology and operating temperature 360K.

All FPGA results are obtained on a Xilinx Virtex UltraScale+ XCVU9P with a PCIe Gen3 x16 interface, 75.9 Mb BRAM and 1182k CLB LUTs in 16nm technology. The FPGAs host computer has an 8 cores Intel i7-7820X CPU running at 3.6 GHz and 128 GB memory. Designs are synthesized with the default synthesis option.

Because the AP, RDB designs and our FPGA solution have similar run-time execution models and all are PCI-Express boards, we can disregard data transfer and control overheads to make general capacity and performance comparisons between these platforms.

Comparison metric: To compare spatial automata processing architectures (the AP, CA, RBDs, and FPGAs), we use throughput per unit area. Throughput is defined as the number of bits that can be processed in one second ($frequency \times Bitwidth\_size$). We then calculate throughput per area (total area used for a benchmark) to consider the effect of consumed spatial resources used in each architecture. If the automata (connected components) in a benchmark cannot fit in one hardware unit (HU), we replicate HUs until all the automata are accommodated. The total area is calculated by multiplying the area of one HU and the number of required HUs for each benchmark.

A full LUT utilization on FPGA means that the board cannot be used to run any other application. Therefore, we use the FPGA die area to calculate total area by each benchmark. Hardware replication can avoid the costly reconfiguration of FPGAs. This increases hardware cost, but cloud computing providers, such as Amazon, offer instances with several FPGAs with reasonably cheap price.

ANMLZoo benchmarks are designed to fit into an AP chip (with up to 48K states). However, because of the AP inefficient routing, the ANMLZoo benchmarks cannot utilize all 48K states, and thus, the benchmarks are pretty small and do not indicate a real-size application. We replicate each benchmark 1000 times to create a larger set of automata for each benchmark. This makes sure that all the benchmarks require at least one unit of hardware in our studied architectures.

7.3 Results

In this section, first, we evaluate the effect of negation operation and analyze the state and transition overhead in various bitwidths on ANMLZoo and Regex benchmark suites. Next, we evaluate hardware implications of FlexAmata on our RDB and FPGA designs and compare them with the AP and CA architectures. AutomataZoo provides an open-source tool to generate automata applications with different configurations.

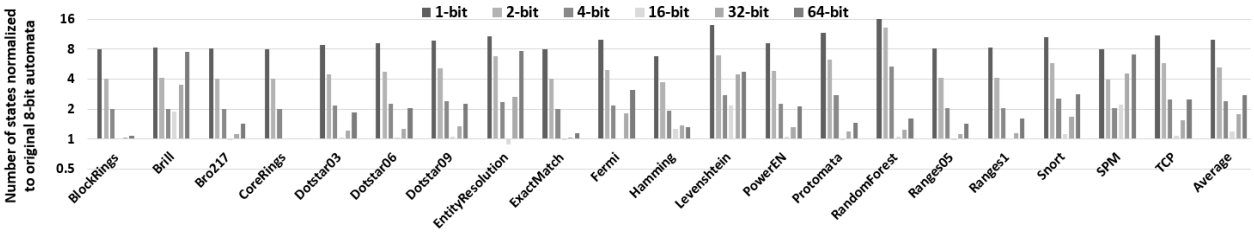

Figure 7.10: State overhead in different bitwidths normalized to original 8-bit automata.

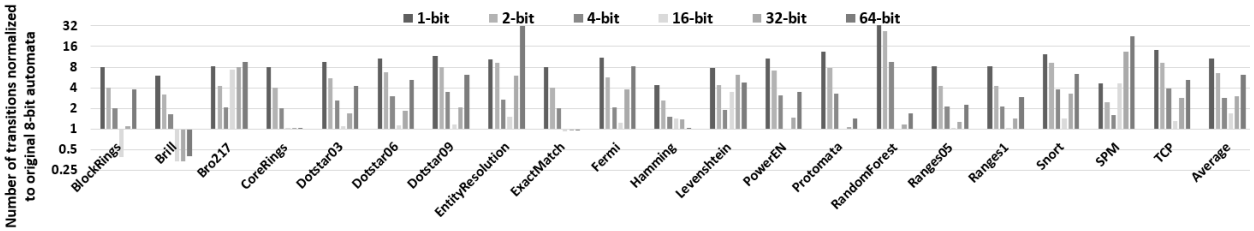

Figure 7.11: Transition overhead (#edges) in different bitwidths normalized to original 8-bit automata.

We then use AutomataZoo to generate automata with very large or small alphabets to evaluate software implications of FlexAmata.

7.3.1 Complexity Analysis of Different Bitwidths

This section discusses the state and transition overhead in different bitwidths and evaluates interval effect explained in Section 7.1.

We observed that benchmarks with higher symbol density (last column in Table 7.1), such as Brill, EntityResolution, Hamming, Protomata, RandomForest, and SPM, have highest state and transition overhead in different bitwidths. Except for RandomForest, the rest of these applications have many states with negation operation ($\hat{}s$, where s is a single symbol). We redesigned the automata for these applications and changed $\hat{}s$ symbols to '*' symbols when possible. This reduces the number of symbols significantly, because in FlexAmata, '*' is translated as one symbol whereas $\hat{}s$ is translated as 255 symbols (assuming 8-bit symbols) - see more details in see Section 7.1, interval effect. Figure 7.12 shows that our interval reduction can decrease the state overhead up to 5.8× (average 3×) and transition overhead up to 13.9× (average 6.8×) in small bitwidths. The rest of the paper is based on the refined version of these applications.

We generate n-bit automata (n=1-16) with FlexAmata for the benchmarks. Figure 7.10 and 7.11 show the number of states and transitions in each bitwidth normalized to the number of states and transition in the original 8-bit automata design. Due to space limitation, we only represent the ones with lowest overhead (1, 2, 4, and 16-bit designs). RandomForest has the highest overhead because its symbol density is relatively high and we could not apply interval reduction technique (negation operation removal) to this benchmark (because

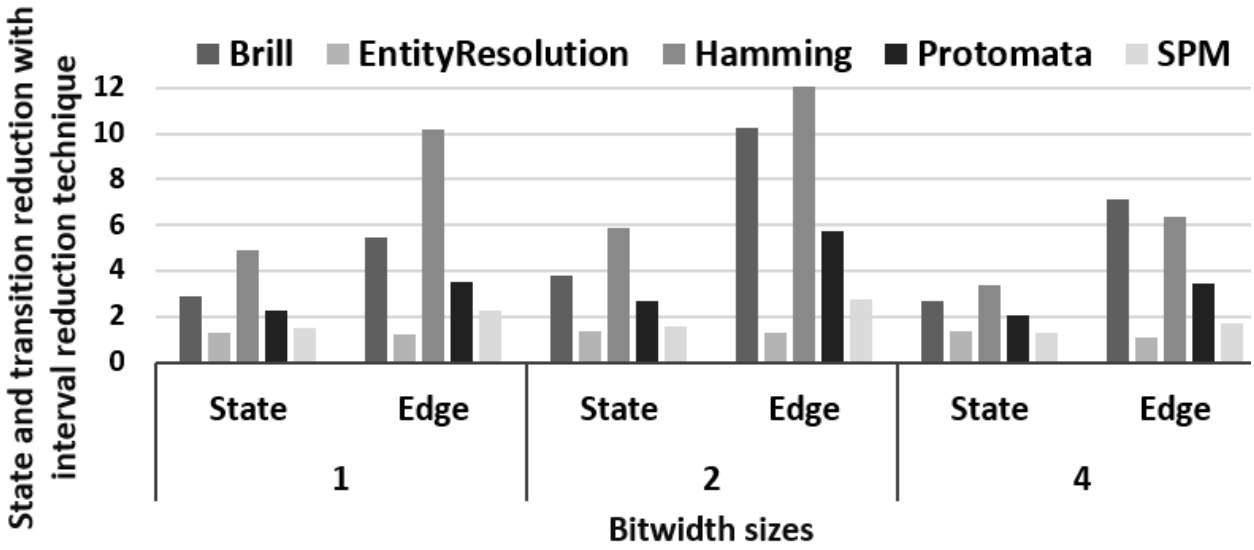

Figure 7.12: Reduction in state and transition overhead of bitwidth transformation in FlexAmata after removing the negation operations relative to the state and transition overhead with the negation operations in the original designs.

it changes the functionality of the application). Interestingly, the number of states in EntityResolution in 16-bit design is less than original 8-bit design. This is because when the original design is converted to bit-automata, FlexAmata applies bit-level minimizations to maximally merge the states. Therefore, the 16-bit designs generated from the optimized bit-automata have fewer states compared to the original 8-bit design. This shows that FlexAmata can be useful to minimize an automaton when its original design is not optimized and have redundancy. Moreover, the applications with high average degree node, such as Levenshtein, have higher state/transition overhead.

On average, 1, 2, 4, and 16, designs have 9.9×, 5.2×, 2.3×, and 1.1× more states and 10.5×, 6.6×, 2.8×, and 1.6× more transitions over the original 8-bit designs. The increase in the number of states translates to higher utilization of memory columns in in-memory designs and LUTs in FPGAs. The increase in the number of transitions translates to more interconnect resources in FPGAs. However, transitions in RBDs are implemented with a memory-based full crossbar interconnect (Figure 7.8), which supports full connectivity. This means that the higher transition count in smaller bitwidths utilizes the existing hardware switches in full crossbar and does not incur extra resource overhead.

Using these analysis, architects can identify the best bitwidth-size for designing an efficient state matching and interconnect architecture. The following subsections outline the bitwidth implications for RBDs.

Table 7.2: Comparison among 1, 2, 4, and 8-bit RBDs and the AP. RBDs are all based on 4MB SRAM-based memory.

Architecture	Symbol size (bit)	Subarray Size	Number of subarrays	Number of States	State matching delay (ns)	Interconnect delay* (ns)	Frequency (GHz)	Area (mm^2)	Total Dynamic R/W energy (nJ) per access	Total leakage power (mW)**
RBD (22nm)	1	R=2, C=256	65,536	16,384K	0.11	2.7	0.368	13.14	0.53	1895
	2	R=4, C=256	32,768	8,192K	0.11	0.368	2.7	10.29	0.45	1969.24
	4	R=16, C=256	8,192	2048K	0.15	0.368	2.7	7.76	0.39	1664
	8	R=256, C=256	512	128K	0.23	0.368	2.7	5.06	0.35	1396
AP (50nm)	8	R=256, C=256	192	48K	7.5	7.5	0.133	144	N/A	N/A

* In RDBs, critical path is state transition stage (interconnect), which is similar for all RDBs. Therefore, they all have a similar clock frequency.

** The details of energy and power are not available for the AP. The estimated TPD is 4W maximum.

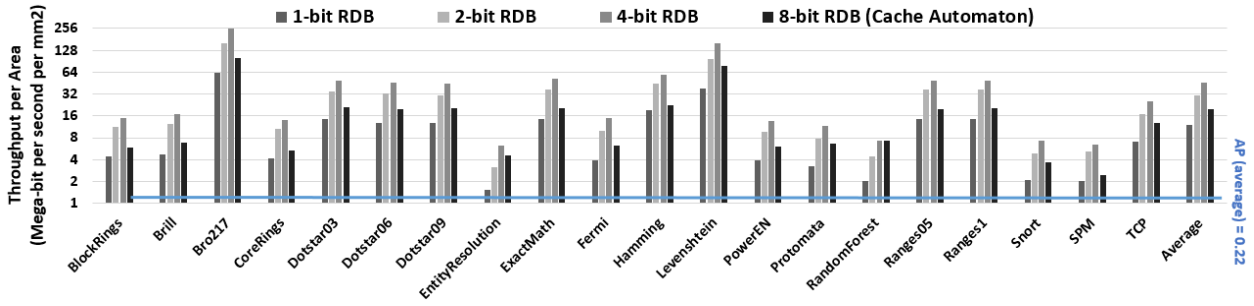

Figure 7.13: Comparing throughput per area (**mega-bit** processing per second per $1mm^2$ area) in RBDs and the AP. RDBs are in 22nm and the AP is scaled to 16nm technologies. On average, 2-bit and 4-bit processing have 1.6× and 2.3× higher throughput per unit area than 8-bit processing, respectively, and more than 100× than the AP.

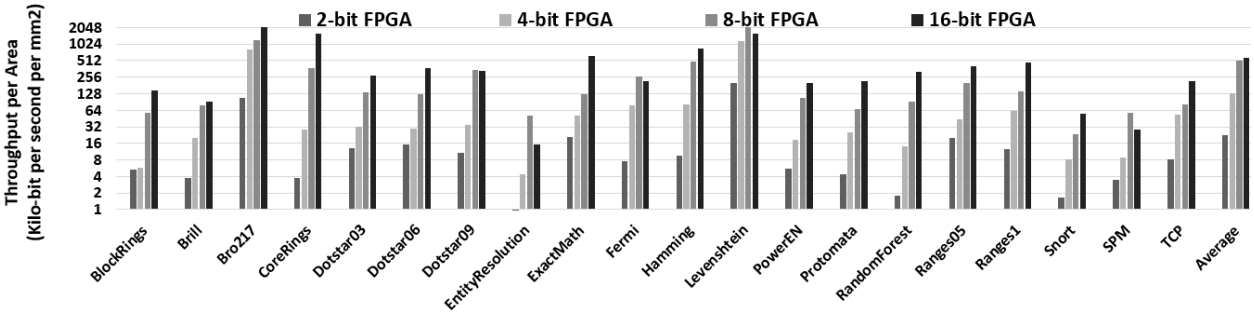

Figure 7.14: Comparing throughput per area (**kilo-bit** processing per second per $1mm^2$ area) in FPGA solutions. On average, 16-bit has 1.3×, 4.4×, and 25× higher throughput per unit area than 2-bit, 4-bit, and 8-bit designs, respectively. Moreover, on average, 16-bit has 2.5× higher throughput per unit area than 8-bit design when ignoring three applications with highest average node degree (SPM, Levenshtein, and EntityResolution. See Table 7.1).

7.3.2 Reduced Bitwidth Designs (RBDs)

This section evaluates reduced bitwidth designs and compares them with the Automata Processor and Cache Automaton models. Table 7.2 presents the architectural parameters for different RDBs (1, 2, 4 and 8-bit) and the AP. The 8-bit design represents Cache Automata model. Generally, in an in-memory automata design, the number of states shows the number of required memory columns and $2^{biwidth-size}$ shows the number of required memory rows.

All RDBs are designed assuming a 4MB SRAM-based memory. The smaller bitwidth designs have smaller subarrays, and thus, they have a higher state density and smaller read and write access time. To calculate clock frequency, we found that the critical path is the state transition stage, where local and global switch arrays are calculating the potential next states in parallel (see Figure 7.8). The global switch stage requires 0.368ns composed of 0.125ns due to wire-delay (SPICE modeling) and 0.243ns due to global switch. The distance between SRAM arrays and global switch arrays is estimated to be maximum 1.9mm assuming maximum state matching dimension of $3.5mm \times 3.75mm$ (for 1-bit design). The pipeline clock frequency is determined by the slowest stage. Thus, the maximum possible frequency is 2.7GHz. We choose to operate at 2.5GHz. The area, total read/write access, and total leakage power of a smaller bitwidth design are higher. This is because more subarrays incur more sense-amplifier and wiring overhead.

Figure 7.13 compares throughput per unit area in 1-bit, 2-bit, 4-bit, and 8-bit RDB designs and the AP across several benchmarks. The applications with more states, such as Entity Resolution and Snort require more hardware resources, and therefore, have lower throughput per unit area. Within each application, the 1-bit design has the lowest throughput per area. This is because the state and transition overhead in 1-bit design will not be amortized by higher state density of 1-bit architecture. On average, 2-bit and 4-bit designs have 1.6× and 2.3× higher throughput per area than original 8-bit design, respectively. This means that to reach the same throughput, an 8-bit design requires 1.6× and 2.3× more hardware units on average than 2-bit and 4-bit designs, respectively.

This is mainly because state density in the 2-bit and 4-bit designs is exponentially (64× and 16×, respectively) higher than 8-bit design (column 5 in table 7.2). This means that more automata can be configured in similar amount of area, which reduces the total hardware resource requirements. The higher state density can pay for 5.2× and 2.3× higher state count and larger total area (column 9 in table 7.2) in the 2-bit and 4-bit designs compared to the 8-bit design. On average, 2-bit and 4-bit designs have 138× and 206× higher throughput per area than the AP.

Using FlexAmata toolchain, we calculated that the number of states in a connected component does not exceed 1024, which is in compliant with our architecture model (Figure 7.8). Moreover, our investigations show that the interconnect of larger connected components can entirely fit into a four 256×256 crossbar switch designs, with allowing up to 64 connections between each with a global switch.

7.3.3 FPGA Results

Performance results for FPGA-based implementations are presented in Table 7.3. A modified REAPR is presented in [1] on thirteen benchmarks from ANMLZoo. We compared our results to this implementation

Table 7.3: Comparing FPGA performance results for different bitwidths and a modified version of REAPR (8-bit) [1] on original size of ANMLZoo [2] benchmarks.

Benchmark																
	2-bit	4-bit	8-bit	16-bit	REAPR	2-bit	4-bit	8-bit	16-bit	REAPR	2-bit	4-bit	8-bit	16-bit	REAPR
Brill	118,220	65,589	39,102	87,191	27,621	147,768	72,044	32,441	44,772	27,782	93	141	165	214	166
PowerEN	130,193	88,252	49,526	53,711	35,359	192,418	89,281	38,398	23,427	31,530	153	174	286	279	163
Protomata	127,237	73,745	47,092	46,706	49,791	194,629	90,628	34,491	19,866	36,285	116	196	167	263	126
Snort	75,754	43,829	22,601	31,610	43,061	345,239	148,443	58079	44,456	28,047	97	117	89	162	98
Hamming	31,170	13,876	7,380	9,302	5,602	139,885	17,884	6,702	3,450	6,637	62	118	187	210	312
Levenshtein	14,286	4,209	2,278	9,877	2,538	17,921	6,090	2,346	3,128	2,242	609	514	719	406	434
Entity Resolution	423,515	178,125	65,020	244,925	50,349	412,980	165,450	53,605	61,890	47,102	85	82	175	97	212
Fermi	113,460	44,729	27,804	38,743	36,314	165,495	71,682	29,555	20,127	32,261	183	376	393	225	116
Random Forest	215,066	89,544	41,907	27,971	50,349	321,262	118,944	30,961	15,571	25,769	80	135	205	233	200
SPM	254,038	14,441	87,435	219,014	64,615	381,749	173,161	57,008	84,244	59,106	185	136	264	168	126
BlockRings	110,782	88,507	41,201	22,496	44,446	177,675	88,875	44,367	22,368	44,185	126	53	123	86	256
Average	146,702	75,893	39,213	71,959	37,276	227,002	94,771	35,268	31,209	30,995	163	186	252	213	201

from the published results and thus, limit Table 7.3 to these thirteen benchmarks. Compared to REAPR (which is an 8-bit design), our 8-bit design has a higher frequency and higher LUTs/FFs usage. Our design uses a staging technique to localize signals and avoid high fanout wires (e.g., input signal). This reduces the critical path, which increases frequency by 25% at the expense of 4% more LUTs and 12% more FFs.

On average, 2-bit and 4-bit processing require 3.7× and 1.9× more LUTs and 6.4× and 2.7× more FFs compared to the original 8-bit design, respectively. This is mainly due to the higher state and edge overhead in the smaller bitwidths (see Figure 7.10 and 7.11). LUTs can have one input and up to two outputs, and thus, they cannot accommodate more states in smaller bitwidth designs. They also have 26% and 35% lower frequency and smaller input processing rate compared to 8-bit design. All these confirm that small bitwidth processing is not suitable for FPGAs.

Compared to 8-bit, 16-bit design has 1.8× more LUTs, 11% fewer FFs and 15% lower frequency. However, the input rate processing of 16-bit design is twice as 8-bit design. This implies that for the applications with real-time processing needs, a 16-bit design with 2× higher throughput can be used. In larger bitwidths (e.g., 16-bit), the number of symbols increases, and thus, more LUTs are required. However, in 16-bit design, there are more states with common parents than 8-bit design, and can share the FFs. Therefore, more LUTs are used than FFs in larger bitwidths. In smaller bitwidths (e.g., 2-bit), more FFs are used than LUTs. This is because there is a higher chance that two states share one LUT when having 2-bit symbols.

Figure 7.14 compares different bitwidths in our FPGA solution. As expected, 2-bit and 4-bit designs have lower throughput per unit area than 8-bit design. Benchmarks with higher average node degree, such as Levenshtein, EntityResolution, and SPM, require relatively more LUTs in 16-bit design than 8-bit design (see Table 7.1 and Table 7.3). This decreases throughput per area in these applications in 16-bit design. On average, the 16-bit design has 2.5× higher throughput per area that 8-bit design (ignoring these three applications). Overall, for regular expressions with relatively lower average node degree than mesh and widgets, 16-bit designs perform best on FPGAs.

Table 7.4: All are 8-bit architectures. Throughput is mega-bit per second per mm^2, and is averaged over 20 benchmarks in Table 7.1. Each benchmark is replicated 1000×.

Architecture (8-bit)	Technology Size (nm)	Ave. Throughput per Area
RDB	22 (original)	19.6
AP	50 (original)	0.007
FPGA	16 (original)	0.5
RDB	16 (projected)	47.1
AP	16 (projected)	0.22

7.3.4 8-bit Processing Across Architectures

This section compares 8-bit designs across spatial architectures, the AP, FPGA, and an in-memory solution (8-bit RDB). On average, 8-bit RDB has 94× and 214× higher throughput per area than FPGA and the AP solutions on the same technology node (16nm). This means that to achieve similar throughput for an application, an FPGA needs on average 94× and the AP needs on average 214× more spatial resources than an in-memory solution (RDB).

RDB efficiency is derived from an efficient and flexible routing architecture, which is a memory-based full-crossbar interconnect that can connect any two states. This results in higher automata density because the state-matching resources are not underutilized due to routing congestions. The automata with more complex routing structures incur routing congestion on the AP and FPGA (FPGAs can handle more complex routing better than the AP), and thus, incur higher area overhead than RDB to accommodate all the automata in a benchmark.

Moreover, our place-and-route algorithm on RDB is 1-2 orders of magnitude faster than the AP compiler, and the AP compiler is 1-2 orders of magnitude faster than the FPGA tools. With a large application and limited number of hardware units, the application might need several rounds of reconfigurations on the hardware. This implies that the AP or FPGA will incur a significant performance penalty for place-and-route when an application do not fit on the available hardware resources.

7.3.5 Hardware Utilization for Smaller Bitwidths

This section evaluates FlexAmata for applications with smaller bitwidths running on existing 8-bit automata accelerators. Many pattern matching applications have smaller alphabets and cannot fully utilize the existing 8-bit accelerators. We use Levenshtein automata[1] in AutomataZoo benchmark suite [118] to evaluate throughput per unit area on 8-bit RDB when using FlexAmata. Levenshtein automata are designed to calculate edit distance between two strings, and are useful for genomics applications.

[1] https://github.com/tjt7a/AutomataZoo/tree/master/Levenshtein

First, we generate Levenshtein automata for different string lengths with 2% edit distance. The strings are randomly generated with A, T, C and G symbols to resemble read-alignment in genome sequencing. Clearly, two-bit is enough to represent the state symbols and input characters. However, the original design assumes 8-bit symbols. Then, using FlexAmata, we (1) reduce symbol-size from 8-bit to 2-bit, (2) generate bit-automata from 2-bit Levenshtein automata, and (3) generate 8-bit automata from bit-automata. The generated 8-bit design can be processed on the existing 8-bit accelerators. The 8-bit automata processes four 2-bit symbols in each cycle, which results in up to 4× increase in throughput with just software modifications.

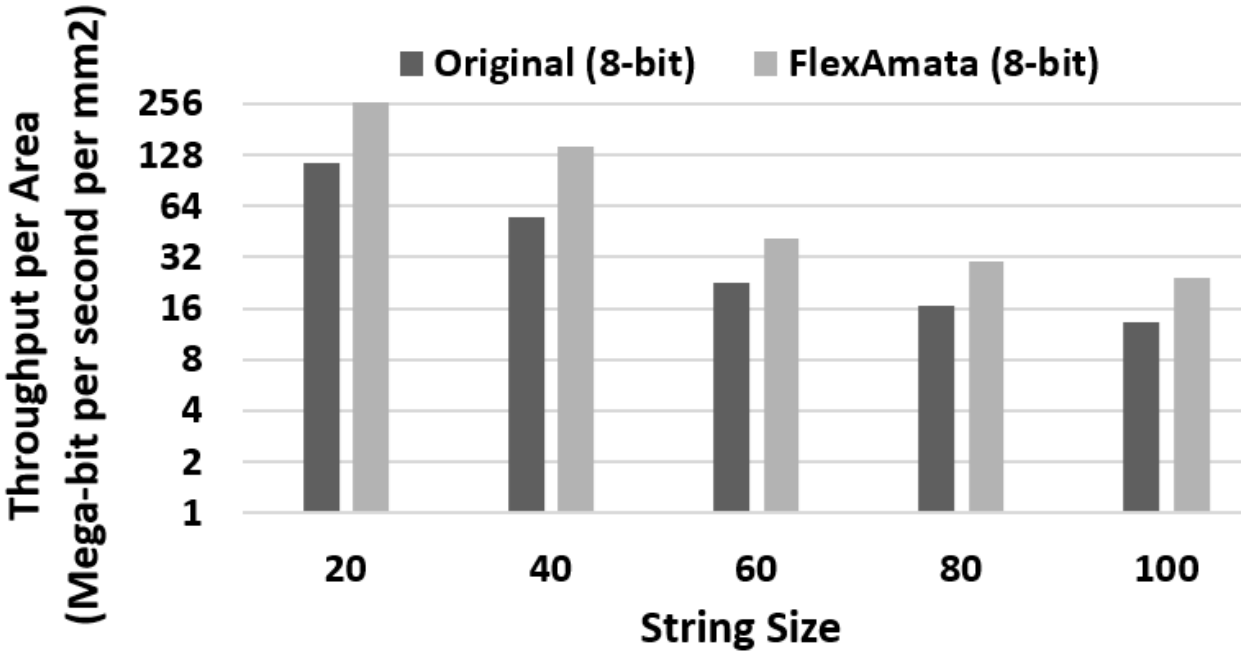

Figure 7.15: Comparing throughput per unit area for Levenshtein with different string lengths in original 8-bit automata design and optimized FlexAmata 8-bit design on 8-bit RDB. On average, FlexAmata design has 2.5× higher throughput per area than original design on RDB.

Fig. 7.15 compares the throughput per unit area for Levenshtein with different sizes in original 8-bit automata design (1 symbol per cycle) and FlexAmata 8-bit design (4 symbols per cycle). On average, FlexAmata design has 2.1× more states than the original design. However, because of its higher processing rate (4 symbols per cycle), it has 2.5× higher throughput per area.This all implies that using FlexAmata as a backend compiler can increase the throughput of applications with smaller symbol-set size without the need to change the existing hardware accelerators.

7.3.6 Feasibility Support for Large Symbol-Set Size

Many pattern matching kernels in data mining and natural language processing have a large symbol-set size. To evaluate the generality of FlexAmata, we use SPM (sequential pattern matching)[2] benchmark and generate frequent sequences for BIBLE dataset[3]. The automata are constructed from frequent sequences in several iterations. BIBLE has 13905 distinct items (symbols), so 14 bits are required.

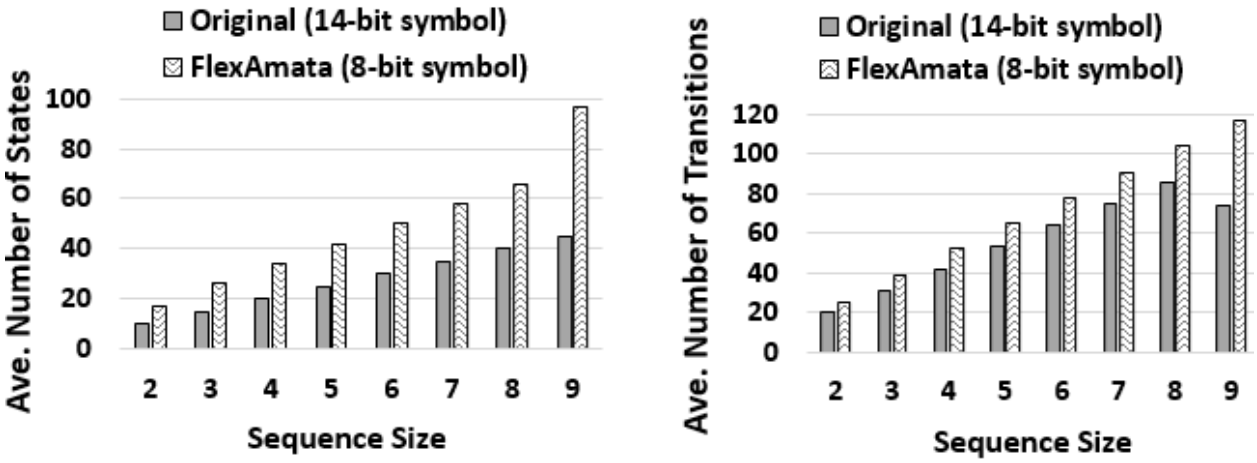

Figure 7.16: Comparing the number of states and transitions in different sequence sizes for original and 8-stride automata.

Fig. 7.16 shows the average number of states and transitions for SPM automata in each iteration. The original automata require a 14-bit symbol processing architecture. This means that the memory should provide 2^{14} rows in each subarray, which is very costly and extremely inefficient for one-hot encoding of symbols. The symbol-set size can easily increase in a larger dataset, which makes increasing the memory column size unfeasible.

FlexAmata provides a scalable and feasible solution by (1) generating 1-bit automata from the original 14-bit automata and then, (2) create 8-bit automata from the 1-stride automata. The resulting 8-bit design provides a low-overhead and feasible solution on the existing architectures and requires only up to 2.1× more states and 1.5× more transitions compared to the original automata (Figure 7.16). This is a very small price to pay for feasibility!

7.4 Conclusions and Future Work

This paper presents a software solution, FlexAmata, to transform an automaton structure to process symbols with various bitwidth sizes. This flexibility introduces software and hardware compatibility for automata

[2]https://github.com/tjt7a/AutomataZoo/tree/master/SeqMatch
[3]http://www.philippe-fournier-viger.com/spmf

processing in a broad range of new applications. Our explorations show that FlexAmata provides higher hardware utilization for applications with small alphabets and feasibility for the applications with very large alphabets. Inspired by the properties of transformed automata on a wide range of applications, we propose in-memory architectures and FPGA solutions to process automata with different bitwidths. Our investigation reveals that 4-bit automata processing on an in-memory architecture has 2.3× higher throughput per unit area than native 8-bit processing. Moreover, 16-bit automata processing performs 2.5× better than 8-bit processing on FPGAs for most of the applications. To summarize, our 4-bit in-memory solution has higher throughput per unit area than all the existing spatial architectures. Future work will explore long bitwidth processing (or multi-symbol processing) on in-memory architectures and FPGAs.

Chapter 8

Conclusions

This dissertation focuses on the design and evaluation of memory-centric accelerators for finite automata processing and deterministic pushdown automata. Accelerating finite automata processing benefits regular-expression workloads, such as network intrusion detection and virus scanning [4], and a wide range of other applications that do not map obviously to regular expressions, including pattern mining [6, 9, 7, 119, 16], bioinformatics [10, 120], natural language processing [121, 15], and machine learning [12]. Moreover, accelerating pushdown automata processing benefits tree-structured workloads such as natural language processing and XML parsing [34].

This dissertation first proposes a new application domain that could benefit from accelerated automata processing: rule-based part-of-speech tagging [14]. We use part-of-speech tagging as a case study to show that spatial automata processing hardware accelerators can make rule-based techniques orders of magnitude faster than statistical-based and machine learning-based taggers. This allows rule-based approaches to employ more rules and achieve competitive accuracy with state-of-the-art techniques. This motivates re-evaluation of rule-based approaches in natural language processing. This work shows that memory-centric hardware accelerators are up to 2,600× and 253× more efficient than CPU-based and GPU-based techniques in the literature, respectively.

This dissertation then introduces another new application domain that can exploit accelerated automata processing hardwares: subtree mining [6]. Subtree structures are more complex than sequences and cannot be represented with regular languages. Therefore, existing finite automata processing accelerators cannot provide an exact solution for tree-based structures. This work proposes a multi-stage pruning strategy on memory-centric automata processing platforms to reduce the search space of the subtree mining problem in a short amount of time, providing a fast and scalable solution at the cost of a small reduction in accuracy. Our

technique improves the performance of subtree mining problem up to 394× over state-of-the-art CPU and GPU solutions, while having 7.5% false positives. To provide a fast and accurate solution for tree mining, we then combine our pruning approach with an exact CPU solution. Our hybrid solution achieves up to 262× better performance than pure CPU solutions. The benefits of our pruning and hybrid approach increase with increasing tree candidates and tree dataset size.

This dissertation then presents a general-purpose, scalable, and reconfigurable in-SRAM architecture that supports rich pushdown automata processing for more complex patterns, such as tree-like data. This provides an exact solution for the tree mining application. We design a custom datapath that performs state matching, stack update, and transition routing using memory subarrays. We also develop a compiler for transforming tree structure to pushdown automata executable on our proposed architecture. By providing hardware support for DPDA, this study brings the efficiency of recent automata acceleration approaches to a new class of applications.

This dissertation then identifies existing in-memory automata processing accelerators suffer from inefficient routing architectures. They are either incapable of efficiently place-and-route a highly connected automaton or require an excessive amount of hardware resources, which both limits the efficiency of complex pattern matching on existing solutions. Motivated by connectivity patterns in the real-world automata benchmarks, we propose a high-speed, dense, and low-power reconfigurable in-memory *reduced* crossbar interconnect for state transitions in automata. RCB compacts the switch patterns in a full crossbar interconnect and provides a 7× reduction in the number of switches. This, in turn, reduces power consumption and delay due to shorter wires. Then, we map our interconnect model and state-matching resource to an efficient memory technology and propose eAP (embedded Automata Processor), a high-speed, dense, and low-power reconfigurable architecture for automata processing. We exploit inherent bit-level parallelism in memory to support multiple concurrent transitions in NFA and utilize subarray-level parallelism in memory to process thousands of automata in parallel. Overall, eAP presents 5.1× and 207× better throughput normalized to area compared to the previously designed in-memory automata accelerators, Cache Automata (CA) and the Automata Processor (AP) respectively. Benefits of eAP are even higher for larger applications

This dissertation then identifies a lack of toolset and investigation for variable bitwidth automata processing on memory-centric architectures. Therefore, we present a software solution, FlexAmata, to transform an automaton structure to process symbols with various bitwidth sizes. This flexibility introduces software and hardware compatibility for automata processing in different applications. Our explorations show that FlexAmata provides higher hardware utilization for applications with small symbol-set size and feasibility for the applications with very large symbol-set size. Inspired by the properties of transformed automata on a wide range of applications, we propose in-memory architectures and FPGA solutions to process

automata with different bitwidths. Our investigation reveals that 4-bit automata processing on an in-memory architecture outperforms native 8-bit processing. Moreover, 16-bit automata processing performs better than 8-bit processing on FPGAs for most of the applications. Overall, our 4-bit in-memory solution has higher throughput per unit area than all the existing spatial architectures.

The first insight is that rule-based methods on automata hardware accelerators can compete with the accuracy of statistical/ML-based approaches, especially in larger datasets. This sets up a very interesting tradeoff to evaluate when designing an application: a small decrease in accuracy in exchange for vastly faster execution. This suggests that rule-based approaches are valuable for use cases where performance is critical, as long as a small drop in accuracy can be tolerated.

The second insight is that the conventional 8-bit automata model does not provide the most efficient computation on memory-centric architecture for real-world automata applications. There are applications with smaller character-set, and their automata do not utilize the 8-bit supported hardware accelerators. On the other hands, there are applications with very large character-set, which cannot be mapped to the existing 8-bit automata accelerators. Moreover, to design the next generation of automata processor, 8-bit designs will not provide the best implementations for general automata processing. This research provides toolset and insights on (1) how the existing automata applications can fully utilize the existing architectures and (2) how to efficiently design future automata accelerators.

Bibliography

[1] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[2] Jack Wadden et al. ANMLZoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *IISWC*. IEEE, 2016.

[3] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[4] Cong Liu and Jie Wu. Fast deep packet inspection with a dual finite automata. *IEEE Transactions on Computers*, 62(2), 2013.

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

[6] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. Frequent subtree mining on the automata processor: challenges and opportunities. In *International Conference on Supercomputing (ICS)*. ACM, 2017.

[7] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Sequential pattern mining with the micron automata processor. In *International Conference on Computing Frontiers*. ACM, 2016.

[8] Ke Wang et al. Association rule mining with the micron automata processor. In *IPDPS*. IEEE, 2015.

[9] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Hierarchical pattern mining with the micron automata processor. In *International Journal of Parallel Programming (IJPP)*. 2017.

[10] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. Searching for potential gRNA off-target sites for CRISPR/Cas9 using automata processing across different platforms. In *24th International Symposium on High-Performance Computer Architecture*. IEEE, 2018.

[11] Indranil Roy and Srinivas Aluru. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM transactions on computational biology and bioinformatics*, 13(1):99–111, 2016.

[12] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. Towards machine learning on the automata processor. In *International Conference on High Performance Computing*. Springer, 2016.

[13] Mateja Putic, AJ Varshneya, and Mircea R Stan. Hierarchical temporal memory on the automata processor. *IEEE Micro*, 37(1):52–59, 2017.

[14] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 665–674. ACM, 2018.

[15] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. Brill tagging on the micron automata processor. In *International Conference on Semantic Computing (ICSC)*. IEEE, 2015.

[16] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. Entity resolution acceleration using the automata processor. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 311–318. IEEE, 2016.

[17] Michael HLS Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. Using the automata processor for fast pattern recognition in high energy physics experimentsa proof of concept. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 832, 2016.

[18] Kubilay Atasu, Florian Doerfler, Jan van Lunteren, and Christoph Hagleitner. Hardware-accelerated regular expression matching with overlap handling on ibm poweren processor. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1254–1265. IEEE, 2013.

[19] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[20] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In *International Conference on Computer Aided Verification*, pages 248–263. Springer, 2000.

[21] Cong Tian and Zhenhua Duan. Model checking propositional projection temporal logic based on spin. In *International Conference on Formal Engineering Methods*, pages 246–265. Springer, 2007.

[22] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. Fast support for unstructured data processing: the unified automata processor. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pages 533–545. IEEE, 2015.

[23] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. REAPR: Reconfigurable engine for automata processing. In *International Conference on Field Programmable Logic and App*. IEEE, 2017.

[24] Rasha Karakchi, Lothrop O Richards, and Jason D Bakos. A dynamically reconfigurable automata processor overlay. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*, pages 1–8. IEEE, 2017.

[25] Yi-Hua Yang and Viktor Prasanna. High-performance and compact architecture for regular expression matching on fpga. *IEEE Transactions on Computers*, 61(7), 2012.

[26] Xiang Wang. *Techniques for efficient regular expression matching across hardware architectures*. University of Missouri-Columbia, 2014.

[27] Ioannis Sourdis, João Bispo, Joao MP Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2008.

[28] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25(12), 2014.

[29] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[30] Noam Chomsky and George A. Miller. Introduction to the formal analysis of natural languages. In *Handbook of Mathematical Psychology*, volume 2, chapter 11, pages 269–322. 1963.

[31] Renáta Iváncsy and István Vajk. Automata theory approach for solving frequent pattern discovery problems. *International Journal of Computer, Electrical, Automation, Control and Information Engineering, World Academy of Science, Engineering and Technology*, 1(8), 2007.

[32] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: Nfa pattern matching on gpgpu devices. *ACM SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[33] Intel. https://github.com/01org/hyperscan.

[34] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, , Reza Rahimi, Kevin Skadron, Weimer, and Reetu Das. Aspen: A scalable in-sram architecture for pushdown automata. In *51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'51)*. IEEE, 2018.

[35] Leland Chang, David M Fried, Jack Hergenrother, Jeffrey W Sleight, Robert H Dennard, Robert K Montoye, Lidija Sekaric, Sharee J McNab, Anna W Topol, Charlotte D Adams, et al. Stable sram cell design for the 32 nm node and beyond. In *VLSI Technology, 2005. Digest of Technical Papers. 2005 Symposium on*, pages 128–129. IEEE, 2005.

[36] Ki Chul Chun, Pulkit Jain, Tae-Ho Kim, and Chris H Kim. A 667 mhz logic-compatible embedded dram featuring an asymmetric 2t gain cell for high speed on-die caches. *IEEE Journal of Solid-State Circuits*, 47(2):547–559, 2012.

[37] Ki Chul Chun, Pulkit Jain, Jung Hwa Lee, and Chris H Kim. A 3t gain cell embedded dram utilizing preferential boosting for high density and low power on-die caches. *IEEE Journal of Solid-State Circuits*, 46(6):1495–1505, 2011.

[38] Wei Zhang, Ki Chul Chun, and Chris H Kim. A write-back-free 2t1d embedded DRAM with local voltage sensing and a dual-row-access low power mode. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(8):2030–2038, 2013.

[39] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

[40] Google. Re2. https://github.com/google/re2.

[41] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. An overview of micron's automata processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on*, pages 1–3. IEEE, 2016.

[42] Jack Wadden et al. Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *in Workload Characterization (IISWC)*. IEEE, 2016.

[43] Victor Mikhaylovich Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 1961.

[44] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-speed regular expression matching engine using multi-character nfa. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 131–136. IEEE, 2008.

[45] Prateek Tandon, Faissal M Sleiman, Michael J Cafarella, and Thomas F Wenisch. Hawk: Hardware support for unstructured log processing. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 469–480. IEEE, 2016.

[46] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 461–472. IEEE Computer Society, 2012.

[47] Ferran Pla, Antonio Molina, and Natividad Prieto. Tagging and chunking with bigrams. In *Proceedings of the 18th conference on Computational linguistics-Volume 2*. Association for Computational Linguistics, 2000.

[48] Claire Cardie. Empirical methods in information extraction. *AI magazine*, 18(4), 1997.

[49] Vivek Kumar Singh, Mousumi Mukherjee, and Ghanshyam Kumar Mehta. Sentiment and mood analysis of weblogs using pos tagging based approach. In *International Conference on Contemporary Computing*. Springer, 2011.

[50] Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013.

[51] Yuan Tian and David Lo. A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2015.

[52] Ramin Shokripour, John Anvik, Zarinah M. Kasirun, and Sima Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013.

[53] Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational linguistics*, 21(4), 1995.

[54] Marzieh Lenjani and Mahmoud Reza Hashemi. Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities. *IET Computers & Digital Techniques*, 8(1):30–48, 2014.

[55] Pradeep G. Mutalik, Aniruddha Deshpande, and Prakash M. Nadkarni. Use of general-purpose negation detection to augment concept indexing of medical documents. *Journal of the American Medical Informatics Association*, 8(6), 2001.

[56] Wendy W. Chapman, Will Bridewell, Paul Hanbury, Gregory F. Cooper, and Bruce G Buchanan. A simple algorithm for identifying negated findings and diseases in discharge summaries. *Journal of biomedical informatics*, 34(5), 2001.

[57] Eric Brill. A simple rule-based part of speech tagger. In *Proceedings of the workshop on Speech and Natural Language*. Association for Computational Linguistics, 1992.

[58] Eric Brill and Mihai Pop. Unsupervised learning of disambiguation rules for part-of-speech tagging. In *Natural language processing using very large corpora*. Springer, 1999.

[59] Saif Mohammad and Ted Pedersen. Guaranteed pre-tagging for the brill tagger. In *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, 2003.

[60] Thorsten Brants. Tnt: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*. Association for Computational Linguistics, 2000.

[61] Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Conference on Empirical Methods in Natural Language Processing*, 1996.

[62] Christopher D Manning. Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In *Computational Linguistics and Intelligent Text Processing*. Springer, 2011.

[63] Jan Hajič, Jan Raab, Miroslav Spousta, et al. Semi-supervised training for the averaged perceptron pos tagger. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 2009.

[64] Juan Antonio Perez-Ortiz and Mikel L Forcada. Part-of-speech tagging with recurrent neural networks. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3. IEEE, 2001.

[65] Ted Xie, Vinh Dang, Chunkun Bo, Jack Wadden, Mircea Stan, and Kevin Skadron. An end-to-end reconfigurable engine for automata processing. In *50th Conference on Government Microcircuit Applications and Critical Technology (GOMACTech)*, 2018.

[66] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Hierarchical pattern mining with the micron automata processor. In *International Journal of Parallel Programming (IJPP)*, 2017.

[67] Grace Ngai and Radu Florian. Transformation-based learning in the fast lane. *arXiv preprint cs/0107020*, 2001.

[68] Dan Klein and Christopher D Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in neural information processing systems*, 2003.

[69] Canasai Kruengkrai, Kiyotaka Uchimoto, Jun'ichi Kazama, Yiou Wang, Kentaro Torisawa, and Hitoshi Isahara. An error-driven word-character hybrid model for joint chinese word segmentation and pos tagging. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics, 2009.

[70] Tetsuji Nakagawa and Kiyotaka Uchimoto. A hybrid approach to word segmentation and pos tagging. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics, 2007.

[71] Cicero D. Santos and Bianca Zadrozny. Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014.

[72] Xiang Yu, Agnieszka Faleńska, and Ngoc Thang Vu. A general-purpose tagger with convolutional neural networks. *arXiv preprint arXiv:1706.01723*, 2017.

[73] Apoorv Agarwal et al. Sentiment analysis of twitter data. In *Proceedings of the Workshop on Languages in Social Media*. Association for Computational Linguistics, 2011.

[74] Mohammed J Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *TKDE, IEEE*, 17(8), 2005.

[75] Shirish Tatikonda and Srinivasan Parthasarathy. Mining tree-structured data on multicore systems. *VLDB, ACM*, 2009.

[76] Yun Chi and Joost Kok. Frequent subtree mining–an overview. *Fundamenta Informaticae*, 21, 2001.

[77] Mohammed J Zaki. Efficiently mining frequent trees in a forest. In *KDD*. ACM, 2002.

[78] Chen Wang et al. Efficient pattern-growth methods for frequent tree pattern mining. In *Proc. of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2004.

[79] Henry Tan et al. Tree model guided candidate generation for mining frequent subtrees from xml documents. *TKDD, ACM*, 2008.

[80] Shirish Tatikonda et al. Trips and tides: new algorithms for tree mining. In *15th ACM CIKM*, 2006.

[81] Computer Sciences Corporation. Big data universe beginning to explode. http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode, 2012.

[82] DNV GL. Are you able to leverage big data to boost your productivity and value creation? https://www.dnvgl.com/assurance/viewpoint/viewpoint-surveys/big-data.html, 2016.

[83] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[84] Zefu Dai, Nick Ni, and Jianwen Zhu. A 1 cycle-per-byte xml parsing accelerator. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 199–208. ACM, 2010.

[85] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[86] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *International Symposium on Microarchitecture*, pages 259–272, 2017.

[87] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, January 1965.

[88] Matthew M. Geller, Michael A. Harrison, and Ivan M. Havel. Normal forms of deterministic grammars. *Discrete Mathematics*, 16(4):313 – 321, 1976.

[89] Michael A. Harrison and Ivan M. Havel. Real-time strict deterministic languages. *SIAM Journal on Computing*, 1(4):333–349, 1972.

[90] William J. Bowhill, Blaine A. Stackhouse, Nevine Nassif, Zibing Yang, Arvind Raghavan, Oscar Mendoza, Charles Morganti, Chris Houghton, Dan Krueger, Olivier Franza, Jayen Desai, Jason Crop, Brian Brock, Dave Bradley, Chris Bostak, Sal Bhimji, and Matt Becker. The Xeon® processor E5-2600 v3: a 22 nm 18-core product family. *J. Solid-State Circuits*, 51(1):92–104, 2016.

[91] Wei Chen, Szu-Liang Chen, Siufu Chiu, Raghuraman Ganesan, Venkata Lukka, Wei Wing Mar, and Stefan Rusu. A 22nm 2.5 mb slice on-die l3 cache for the next generation Xeon® processor. In *Symposium on VLSI Technology*, pages C132–C133, 2013.

[92] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. An energy efficient 32-nm 20-mb shared on-die L3 cache for Intel® Xeon® processor E5 family. *J. Solid-State Circuits*, 48(8):1954–1962, 2013.

[93] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998.

[94] Intel. Cache Allocation Technology, 2017.

[95] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[96] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *International Symposium on Low-Power Electronics and Design*, 2010.

[97] nvprof profiling tool. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview.

[98] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. Frequent subtree mining on the automata processor: challenges and opportunities. In *International Conference on Supercomputing*, page 4, 2017.

[99] Linley Gwennap. New chip speeds nfa processing using DRAM architectures. In *In Microprocessor Report*, 2014.

[100] Michela Becchi, Mark Franklin, and Patrick Crowley. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008.

[101] Micron. RLDRAM memory. https://www.micron.com/products/dram/\rldram-memory, 2019.

[102] E Amat, A Calomarde, F Moll, R Canal, and A Rubio. Feasibility of embedded DRAM cells on FinFET technology. *IEEE Transactions on Computers*, 65(4):1068–1074, 2016.

[103] Ajay N Bhoj and Niraj K Jha. Pragmatic design of gated-diode FinFET DRAMs. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 390–397. IEEE, 2009.

[104] Zoran Jaksic. *Cache memory design in the FinFET era*. PhD thesis, Universitat Politècnica de Catalunya, 2015.

[105] Rui Liu, Xiaochen Peng, Xiaoyu Sun, Win-San Khwa, Xin Si, Jia-Jing Chen, Jia-Fang Li, Meng-Fan Chang, and Shimeng Yu. Parallelizing sram arrays with customized bit-cell for binary neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, page 21. ACM, 2018.

[106] Amogh Agrawal, Akhilesh Jaiswal, Bing Han, Gopalakrishnan Srinivasan, and Kaushik Roy. Xcel-ram: Accelerating binary neural networks in high-throughput sram compute arrays. *arXiv preprint arXiv:1807.00343*, 2018.

[107] Adam Teman, Pascal Meinerzhagen, Andreas Burg, and Alexander Fish. Review and classification of gain cell eDRAM implementations. In *Electrical & Electronics Engineers in Israel (IEEEI), 2012 IEEE 27th Convention of*, pages 1–5. IEEE, 2012.

[108] Pascal Meinerzhagen, Adam Teman, Robert Giterman, Andreas Burg, and Alexander Fish. Exploration of sub-vt and near-vt 2t gain-cell memories for ultra-low power applications under technology scaling. *Journal of Low Power Electronics and Applications*, 3(2):54–72, 2013.

[109] Ki Chul Chun, Wei Zhang, Pulkit Jain, and Chris H Kim. A 700mhz 2t1c embedded DRAM macro in a generic logic process with no boosted supplies. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 506–507. IEEE, 2011.

[110] Zoran Jaksic and Ramon Canal. Enhancing 3T DRAMs for SRAM replacement under 10nm tri-gate SOI FinFETs. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 309–314. IEEE, 2012.

[111] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (SALP) in DRAM. *ACM SIGARCH Computer Architecture News*, 40(3):368–379, 2012.

[112] Jack Wadden, Kevin Angstadt, and Kevin Skadron. Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 749–761. IEEE, 2018.

[113] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[114] Aditya Agrawal. *REFRESH REDUCTION IN DYNAMIC MEMORIES*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

[115] Intel. 2017. Cache allocation technology. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[116] Fatih Hamzaoglu, Umut Arslan, Nabhendra Bisnik, Swaroop Ghosh, Manoj B Lal, Nick Lindert, Mesut Meterelliyoz, Randy B Osborne, Joodong Park, Shigeki Tomishima, et al. A 1gb 2ghz embedded DRAM in 22nm tri-gate CMOS technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 230–231. IEEE, 2014.

[117] Shufang Zhu, Geguang Pu, and Moshe Y Vardi. First-order vs. second-order encodings for ltlf-to-automata translation. *arXiv preprint arXiv:1901.06108*, 2019.

[118] Jack Wadden et al. AutomataZoo: A modern automata processing benchmark suite. In *IISWC*. IEEE, 2018.

[119] Ke Wang, Yanjun Qi, J.J. Fox, M.R. Stan, and K. Skadron. Association rule mining with the micron automata processor. In *Proc. of IPDPS'15*, May 2015.

[120] Indranil Roy and Srinivas Aluru. Finding motifs in biological sequences using the micron automata processor. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014.

[121] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators. In *24th ACMSIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*. ACM, 2018.

[122] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005.

[123] Mohammed J Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 66(1-2), 2005.

[124] Jack Wadden, Samira Khan, and Kevin Skadron. Automata-to-routing: An open-source toolchain for design-space exploration of spatial automata processing architectures. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 180–187. IEEE, 2017.

[125] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[126] Jack Wadden and Kevin Skadron. VASim: An open virtual automata simulator for automata processing application and architecture research. Technical report, Technical Report CS2016-03, University of Virginia, 2016.

[127] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 737–748. IEEE, 2018.

[128] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015.

[129] ITRS.

[130] Fatih Hamzaoglu, Umut Arslan, Nabhendra Bisnik, Swaroop Ghosh, Manoj B Lal, Nick Lindert, Mesut Meterelliyoz, Randy B Osborne, Joodong Park, Shigeki Tomishima, et al. A 1 gb 2 ghz 128 gb/s bandwidth embedded DRAM in 22 nm tri-gate CMOS technology. *IEEE Journal of Solid-State Circuits*, 50(1):150–157, 2015.

[131] Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits*, volume 2. 2002.

[132] W Regitz and J Karp. A three transistor-cell, 1024-bit, 500 ns mos ram. In *Solid-State Circuits Conference. Digest of Technical Papers. 1970 IEEE International*, volume 13, pages 42–43. IEEE, 1970.

[133] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*. ACM, 2007.

[134] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment*, 3(1-2), 2010.

[135] David BrownHarold, Harold Noyes, Irene Junjuan Xu, and Paul Glendenning. Methods and systems for routing in a state machine. In *US Patent 8680888B2*, 2014.

[136] Richard E Matick and Stanley E Schuster. Logic-based eDRAM: Origins and rationale for use. *IBM Journal of Research and Development*, 49(1):145–165, 2005.

[137] John Barth, Don Plass, Erik Nelson, Charlie Hwang, Gregory Fredeman, Michael Sperling, Abraham Mathews, Toshiaki Kirihata, William R Reohr, Kavita Nair, et al. A 45 nm soi embedded DRAM macro for the power processor 32 mbyte on-chip l3 cache. *IEEE Journal of Solid-State Circuits*, 46(1):64–75, 2011.

[138] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[139] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, 2015.

[140] Woong Choi, Gyuseong Kang, and Jongsun Park. A refresh-less eDRAM macro with embedded voltage reference and selective read for an area and power efficient viterbi decoder. *IEEE Journal of Solid-State Circuits*, 50(10):2451–2462, 2015.

[141] Nasser Kurd, Muntaquim Chowdhury, Edward Burton, Thomas P Thomas, Christopher Mozak, Brent Boswell, Praveen Mosalikanti, Mark Neidengard, Anant Deval, Ashish Khanna, et al. Haswell: A family of ia 22 nm processors. *IEEE Journal of Solid-State Circuits*, 50(1):49–58, 2015.

[142] Gregory Fredeman, Donald W Plass, Abraham Mathews, Janakiraman Viraraghavan, Kenneth Reyer, Thomas J Knips, Thomas Miller, Elizabeth L Gerhard, Dinesh Kannambadi, Chris Paone, et al. A 14 nm 1.1 mb embedded DRAM macro with 1 ns access. *IEEE Journal of Solid-State Circuits*, 51(1):230–239, 2016.

[143] Xilinx. UltraScale architecture memory resources. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf, 2019.

[144] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. An energy efficient 32-nm 20-mb shared on-die l3 cache for intel® xeon® processor e5 family. *IEEE Journal of Solid-State Circuits*, 48(8):1954–1962, 2013.

[145] II Tommy Tracy, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabriel Robins. Nondeterministic finite automata in hardware-the case of the levenshtein automaton.

[146] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[147] Ye Zhang, Wai-Shing Luk, Hai Zhou, Changhao Yan, and Xuan Zeng. Layout decomposition with pairwise coloring for multiple patterning lithography. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 170–177. IEEE, 2013.

[148] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[149] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[150] G Wang, D Anand, N Butt, A Cestero, M Chudzik, J Ervin, S Fang, G Freeman, H Ho, B Khan, et al. Scaling deep trench based eDRAM on soi to 32nm and beyond. In *Electron Devices Meeting (IEDM), 2009 IEEE International*, pages 1–4. IEEE, 2009.

[151] Bill Bowhill, Blaine Stackhouse, Nevine Nassif, Zibing Yang, Arvind Raghavan, Oscar Mendoza, Charles Morganti, Chris Houghton, Dan Krueger, Olivier Franza, et al. The xeon® processor e5-2600 v3: A 22 nm 18-core product family. *IEEE Journal of Solid-State Circuits*, 51(1):92–104, 2016.

[152] Ang Li, Weifeng Liu, Mads RB Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 26. ACM, 2017.

[153] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154. ACM, 2007.

[154] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2009.

[155] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 93–102. IEEE, 2006.

[156] Xiaodong Yu, Bill Lin, and Michela Becchi. Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence. *IEEE Journal on Selected Areas in Communications*, 32(10):1822–1833, 2014.

[157] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. In *ACM SIGPLAN Notices*, volume 47, pages 129–140. ACM, 2012.

[158] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. Cached DRAM for ilp processor memory access latency reduction. *IEEE Micro*, 21(4):22–32, 2001.

[159] Pedro Reviriego, Alfonso Sanchez-Macian, and Juan Antonio Maestro. Low power embedded DRAM caches using bch code partitioning. In *On-Line Testing Symposium (IOLTS), 2012 IEEE 18th International*, pages 79–83. IEEE, 2012.

[160] Wikipedia. Skylake microarchitecture. https://en.wikipedia.org/wiki/Skylake_(microarchitecture).

[161] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 1–17. Springer, 1996.

[162] Transcend. https://us.transcend-info.com/.

[163] Lei Zou, Yansheng Lu, Huaming Zhang, and Rong Hu. Mining frequent induced subtree patterns with subtree-constraint. In *Data Mining Workshops, 2006. ICDM Workshops 2006. Sixth IEEE International Conference on*. IEEE, 2006.

[164] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. Mining induced and embedded subtrees in ordered, unordered, and partially-ordered trees. In *Foundations of Intelligent Systems*. Springer, 2008.

[165] Mostafa Haghir Chehreghani, Morteza Haghir Chehreghani, Caro Lucas, and Masoud Rahgozar. Oinduced: An efficient algorithm for mining induced patterns from rooted ordered trees. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 41(5), 2011.

[166] Henry Tan, Tharam S Dillon, Fedja Hadzic, Elizabeth Chang, and Ling Feng. Imb3-miner: mining induced/embedded subtrees by constraining the level of embedding. In *Advances in Knowledge Discovery and Data Mining*. Springer, 2006.

[167] Kenji Abe, Shinji Kawasoe, Tatsuya Asai, Hiroki Arimura, and Setsuo Arikawa. Optimized substructure discovery for semi-structured data. In *Principles of Data Mining and Knowledge Discovery*. Springer, 2002.

[168] Fedja Hadzic, Michael Hecker, and Andrea Tagarelli. Ordered subtree mining via transactional mapping using a structure-preserving tree database schema. *Information Sciences*, 310:97–117, 2015.

[169] Sangeetha Kutty, Richi Nayak, and Yuefeng Li. Hcx: an efficient hybrid clustering approach for xml documents. In *Proceedings of the 9th ACM symposium on Document engineering*. ACM, 2009.

[170] Fedja Hadzic, Henry Tan, and Tharam S Dillon. *Mining of data with complex structures*, volume 333. Springer.

[171] Pekka Kilpeläinen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2), 1995.

[172] Mostafa Haghir Chehreghani and Maurice Bruynooghe. Mining rooted ordered trees under subtree homeomorphism. *Data Mining and Knowledge Discovery*, 2015.

[173] Yun Chi, Yirong Yang, and Richard R Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowledge and Information Systems*, 8(2), 2005.

[174] Henry Tan Setiawan. *Tree Model Guided (TMG) Enumeration as the Basis for Mining Frequent Patterns from XML Documents*. PhD thesis, University of Technology, Sydney, 2007.

[175] Fedja Hadzic. A structure preserving flat data format representation for tree-structured data. In *New Frontiers in Applied Data Mining*. Springer, 2012.

[176] Fedja Hadzic, Henry Tan, and Tharam S Dillon. Uni3-efficient algorithm for mining unordered induced subtrees using tmg candidate generation. In *Computational Intelligence and Data Mining, 2007. CIDM 2007. IEEE Symposium on*. IEEE, 2007.

[177] Alessandro Moschitti. Making tree kernels practical for natural language learning. volume 113. EACL, 2006.

[178] Indranil Roy. *Algorithmic techniques for the micron automata processor*. PhD thesis, Georgia Institute of Technology, 2015.

[179] Ke Wang and Kevin Skadron. Cellular automata on the micron automata processor. Technical Report CS-2015-03, University of Virginia Department of Computer Science, April 2015.

[180] Shashank Agarwal and Hong Yu. Biomedical negation scope detection with conditional random fields. *Journal of the American medical informatics association*, 17(6), 2010.

[181] Chao-Yue Lai. *Efficient Parallelization of Natural Language Applications using GPUs*. PhD thesis, University of California at Berkeley, 2012.

[182] Nigel Collier and Koichi Takeuchi. Comparison of character-level and part of speech features for name recognition in biomedical texts. *Journal of Biomedical Informatics*, 37(6), 2004.

[183] Pos tagging (state of the art), 2016.

[184] Jack Wadden, Tommy Tracy II, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Matthew Wallace, Udall, Jeffry, Mircea Stan, and Kevin Skadron. Automatazoo: A modern automata processing benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[185] Emmanuel Roche and Yves Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational linguistics*, 21(2), 1995.

[186] Nitin Indurkhya and Fred J Damerau. *Handbook of natural language processing*, volume 2. CRC Press, 2010.

[187] Michele Banko and Robert C Moore. Part of speech tagging in context. In *Proceedings of the 20th international conference on Computational Linguistics*. Association for Computational Linguistics, 2004.

[188] Sujith Ravi and Kevin Knight. Minimized models for unsupervised part-of-speech tagging. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics, 2009.

[189] Helmut Schmid. Treetagger: a language independent part-of-speech tagger. *CIn ACL SIGDAT Workshop*, 1995.

[190] *iNFAnt2*. https://github.com/vqd8a/iNFAnt2.

[191] *DFAGE*. https://github.com/vqd8a/DFAGE.

[192] *Perceptron Tagger*. https://explosion.ai/blog/part-of-speech-pos-tagger-in-python.

[193] Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. In *Encyclopedia of machine learning*. Springer, 2011.

[194] Brent Keeth. *DRAM circuit design: fundamental and high-speed topics*. John Wiley & Sons, 2007.

[195] Yi-Hua E Yang and Viktor K Prasanna. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *INFOCOM, 2011 Proceedings IEEE*, pages 1853–1861. IEEE, 2011.

[196] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59. ACM, 2008.

[197] TF SMITE. Identification of common molecular subsequences. *J. Mol. Bwl*, 147:195–197, 1981.

[198] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[199] Qiong Wang, Mohamed El-Hadedy, Kevin Skadron, and Ke Wang. Accelerating weeder: A dna motif search tool using the micron automata processor and fpga. *IEICE Transactions on Information and Systems*, 100(10):2470–2477, 2017.

[200] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction*, pages 73–88, 2004.

[201] E. Klein and M. Martin. The parser generating system PGS. *Software: Practice and Experience*, 19(11):1015–1028, 1989.

[202] Peter Dencker, Karl Dürre, and Johannes Heuft. Optimization of parser tables for portable compilers. *ACM Trans. Program. Lang. Syst.*, 6(4):546–572, October 1984.

[203] V. B. Schneider and M. D. Mickunas. Optimal compression of parsing tables in a parsergenerating system. Technical Report 75-150, Purdue University, 1975.

[204] I. Roy, N. Jammula, and S. Aluru. Algorithmic techniques for solving graph problems on the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, pages 283–292, May 2016.

[205] Tommy Tracy II, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. Nondeterministic finite automata in hardware—the case of the Levenshtein automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[206] 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 *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 622–629, Oct 2016.

[207] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[208] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109, 2005.

[209] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2003.

[210] Kevin Angstadt, Westley Weimer, and Kevin Skadron. Rapid programming of pattern-recognition processors. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 593–605, 2016.

[211] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-0: An imperative language that supports declarative programming. Technical report, 1997.

[212] Jack Wadden, Samira Khan, and Kevin Skadron. Automata-to-Routing: An Open Source Toolchain for Design-Space Exploration of Spatial Automata Processing Architectures. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[213] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[214] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. Demystifying automata processing: Gpus, fpgas or micron's ap? In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 1:1–1:11, New York, NY, USA, 2017. ACM.

[215] Michela Becchi. Regular expression processor. http://regex.wustl.edu, 2011. Accessed 2017-04-06.

[216] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of Architectures for Networking and Communications Systems*, ANCS '08, pages 50–59, 2008.

[217] C. Bo, K. Wang, J. J. Fox, and K. Skadron. Entity resolution acceleration using the automata processor. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 311–318, Dec 2016.

[218] Chunkun Bo, Ke Wang, Yanjun Qi, and Kevin Skadron. String kernel testing acceleration using the Micron Automata Processor. In *Workshop on Computer Architecture for Machine Learning*, 2015.

[219] Capgemini. Big & fast data : The rise of insight-driven business. http://www.capgemini.com/resource-file-access/resource/pdf/big_fast_data_the_rise_of_insight-driven_business-report.pdf, 2015.

[220] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1):75–90, 2000.

[221] H. D. Cheng and K. S. Fu. VLSI architectures for string matching and pattern matching. *Pattern Recognition*, 20(1):125–144, 1987.

[222] Micron Technoloy. Calculating Hamming distance. http://www.micronautomata.com/documentation/cookbook/c_hamming_distance.html.

[223] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[224] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru. High performance pattern matching using the automata processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, pages 1123–1132, 2016.

[225] Michael H.L.S. Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted 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*, 2016, to appear.

[226] Arne Halaas. A systolic VLSI matrix for a family of fundamental searching problems. *Integration VLSI Journal*, 1(4):269–282, 1983.

[227] Arne Halaas, Børge Svingen, Magnar Nedland, Pål Sætrom, Ola Snøve, Jr., and Olaf René Birkeland. A recursive MISD architecture for pattern matching. *IEEE Transactions on Very Large Scale Integrated Systems*, 12(7):727–734, 2004.

[228] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: Writing parallel program portable between cpu and gpu. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[229] Yusaku Kaneta, Shingo Yoshizawa, SI Minato, and Hiroki Arimura. High-Speed String and Regular Expression Matching on FPGA. In *Proceedings of the Asia-Pacific Signal and Information Processing Association (APSIPA-ASC)*, 2011.

[230] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Parallel Architectures and Compilation Techniques*, pages 389–400, 2012.

[231] Michael E Lesk and Eric Schmidt. Lex: A lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories Computing Science, 1975.

[232] Hongbin Lu, Kai Zheng, Bin Liu, Xin Zhang, and Yunhao Liu. A memory-efficient parallel string matching architecture for high-speed intrusion detection. *IEEE Journal on Selected Areas in Communications*, 24(10):1793–1804, 2006.

[233] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 USENIX Winter Conference*, USENIX'93, pages 295–270, 1993.

[234] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron. MNCaRT: An open-source, multi-architecture automata-processing research and execution ecosystem. *IEEE Computer Architecture Letters*, 17(1):84–87, Jan 2018.

[235] Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron. MNRL and MNCaRT: An open-source, multi-architecture state machine research and execution ecosystem. Technical Report CS2017-01, University of Virginia, 2017.

[236] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[237] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *International Symposium on Computer Architecture*, pages 600–612, New York, NY, USA, 2017.

[238] Reetinder Sidhu and Viktor K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.

[239] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact Architecture for High-throughput Regular Expression Matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 30–39, New York, NY, USA, 2008. ACM.

[240] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. R. Stan. REAPR: Reconfigurable engine for automata processing. In *International Conference on Field-Programmable Logic and Applications*, 2017.

[241] Xiang Wang. Techniques for efficient regular expression matching across hardware architectures. Master's thesis, University of Missouri-Columbia, 2014.

[242] Indranil Roy and Srinivas Aluru. Finding motifs in biological sequences using the Micron Automata Processor. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, pages 415–424, 2014.

[243] Indranil Roy. *Algorithmic Techniques for the Micron Automata Processor*. PhD thesis, Georgia Institute of Technology, 2015.

[244] Christopher Sabotta. Advantages and challenges of programming the Micron Automata Processor, 2013.

[245] Ioannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2008.

[246] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, 2012.

[247] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Sequential pattern mining with the Micron Automata Processor. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 135–144, New York, NY, USA, 2016. ACM.

[248] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196. Springer-Verlag, 2002.

[249] Titan IC Systems. RXP regular eXpression processor soft IP. http://titanicsystems.com/Products/Regular-eXpression-Processor-(RXP).

[250] Titan IC Systems. Helios RXPF soft IP for FPGA security analytics acceleration. http://titan-ic.com/products/helios-rxpf, 2017. Accessed 2017-04-05.

[251] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. Towards machine learning on the Automata Processor. In *Proceedings of ISC High Performance Computing*, pages 200–218, 2016.

[252] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the ACM International Symposium on Microarchitecture*, Micro '15, pages 533–545, 2015.

[253] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger. Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 232–239, April 2012.

[254] Jack Wadden and Kevin Skadron. VASim: An open virtual automata simulator for automata processing application and architecture research. Technical Report CS2016-03, University of Virginia, 2016.

[255] Peter J. L. Wallis. The design of a portable programming language. *Acta Informatica*, 10(2):157–167, Jun 1978.

[256] Ke Wang, Mircea Stan, and Kevin Skadron. Association rule mining with the Micron Automata Processor. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, 2015.

[257] Hachiro Yamada, Masaki Hirata, Hajime Nagai, and Kousuke Takahashi. A high-speed string-search engine. *IEEE Journal of Solid-State Circuits*, 22(5):829–834, 1987.

[258] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong. Map-reduce as a programming model for custom computing machines. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 149–159, April 2008.

[259] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. Brill tagging on the Micron Automata Processor. In *Proceedings of the 9th IEEE International Conference on Semantic Computing*, pages 236–239, 2015.

[260] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1994.

[261] Firstname1 Lastname1 and Firstname2 Lastname2. A very nice paper to cite. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[262] Firstname1 Lastname1, Firstname2 Lastname2, and Firstname3 Lastname3. Another very nice paper to cite. In *Intl. Symp. on Microarchitecture (MICRO)*, Oct. 2012.

[263] Firstname1 Lastname1, Firstname2 Lastname2, Firstname3 Lastname3, Firstname4 Lastname4, and Firstname5 Lastname5. Yet another very nice paper to cite, with many author names all spelled out. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2011.

[264] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. Entity resolution acceleration using micron's automata processor. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[265] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Architectural Support for Programming Languages and Operating Systems*, pages 529–542, 2014.

[266] Margus Veanes, Todd Mytkowicz, David Molnar, and Benjamin Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 139–152, 2015.

[267] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[268] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[269] Micron. Micron Automata Processing. 2017.

[270] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[271] Kevin Angstadt, Westley Weimer, and Kevin Skadron. Rapid programming of pattern-recognition processors. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 593–605. ACM, 2016.

[272] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved dfa for fast regular expression matching. *ACM SIGCOMM Computer Communication Review*, 38(5):29–40, 2008.

[273] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 339–350. ACM, 2006.

[274] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, volume 49, pages 309–328. ACM, 2014.

[275] Sutapa Datta and Subhasis Mukhopadhyay. A grammar inference approach for predicting kinase specific phosphorylation sites. *PloS one*, 10(4):e0122294, 2015.

[276] Zhen-Gang Wang, Johann Elbaz, Françoise Remacle, RD Levine, and Itamar Willner. All-dna finite-state automata with finite memory. *Proceedings of the National Academy of Sciences*, 107(51):21996–22001, 2010.

[277] Yangjun Chen, Duren Che, and Karl Aberer. On the efficient evaluation of relaxed queries in biological databases. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 227–236. ACM, 2002.

[278] Alexandre Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Modeling and verification of parallel processes*, pages 196–205. Springer, 2001.

[279] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[280] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 175–188. ACM, 1998.

[281] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[282] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for FSM parallelization. In *Architectural Support for Programming Languages and Operating Systems*, pages 619–630, 2015.

[283] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the "embarrassingly sequential": parallelizing finite state machine-based computations through principled speculation. In *Architectural Support for Programming Languages and Operating Systems*, pages 543–558, 2014.

[284] Junqiao Qiu, Zhijia Zhao, and Bin Ren. Microspec: Speculation-centric fine-grained parallelization for FSM computations. In *Parallel Architectures and Compilation*, pages 221–233, 2016.

[285] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[286] B Kaplan. Speculative parsing path. http://bugzilla.mozilla.org.

[287] Shmuel Tomi Klein and Yair Wiseman. Parallel huffman decoding with applications to jpeg files. *The Computer Journal*, 46(5):487–497, 2003.

[288] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 284–303. Springer, 2009.

[289] II Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. Towards machine learning on the automata processor. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, volume 9697, page 200. Springer, 2016.

[290] Jack Wadden, Nathan Brunelle, Ke Wang, Mohamed El-Hadedy, Gabriel Robins, Mircea Stan, and Kevin Skadron. Generating efficient and high-quality pseudo-random behavior on automata processors.

[291] Micron Automata Processing D480 Software Development Kit. AP Flow Concepts. http://micronautomata.com/apsdk_documentation/latest/h1_ap.html.

[292] Micron Automata Processing D480 Documentation Design Notes. http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html.

[293] Linley Gwennap. Micron accelerates automata:new chip speeds nfa processing using dram architectures. In *Microprocessor Report*, 2014.

[294] Michela Becchi, Mark A. Franklin, and Patrick 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*, pages 79–89, 2008.

[295] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 2008 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2008, San Jose, California, USA, November 6-7, 2008*, pages 50–59, 2008.

[296] Crowley. P Becchi. M. An improved algorithm to accelerate regular expression evaluation. *Intl. Conf. on Architectures for Networking and Communication Systems*, (18):145–154, 2007.

[297] Jan van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *International Symposium on Microarchitecture*, pages 461–472, 2012.

[298] Prateek Tandon, Faissal M. Sleiman, Michael J. Cafarella, and Thomas F. Wenisch. HAWK: hardware support for unstructured log processing. In *International Conference on Data Engineering*, pages 469–480, 2016.

[299] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. HARE: hardware accelerator for regular expressions. In *International Symposium on Microarchitecture*, pages 1–12, 2016.

[300] Yuanwei Fang, Tung Thanh Hoang, Michela Becchi, and Andrew A. Chien. Fast support for unstructured data processing: the unified automata processor. In *International Symposium on Microarchitecture*, pages 533–545, 2015.

[301] Peng Jiang and Gagan Agrawal. Combining SIMD and many/multi-core parallelism for finite state machines with enumerative speculation. In *Principles and Practice of Parallel Programming*, pages 179–191, 2017.

[302] Yi-Hua E. Yang and Viktor K. Prasanna. Optimizing regular expression matching with SR-NFA on multi-core systems. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, pages 424–433, 2011.

[303] Bingsheng He, Qiong Luo, and Byron Choi. Cache-conscious automata for XML filtering. *IEEE Trans. Knowl. Data Eng.*, 18(12):1629–1644, 2006.

[304] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *International Symposium on High Performance Computer Architecture*, pages 373–384, 2012.

[305] Subhasis Das, Tor M. Aamodt, and William J. Dally. SLIP: reducing wire energy in the memory hierarchy. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 349–361, 2015.

[306] DDR3 SDRAM System-Power Calculator. http://www.micronautomata.com/.

[307] Korey Sewell, Ronald G. Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Ross Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F. Wenisch, Dennis Sylvester, David Blaauw, and Trevor N. Mudge. Swizzle-switch networks for many-core systems. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 2(2):278–294, 2012.

[308] Nilmini Abeyratne, Reetuparna Das, Qingkun Li, Korey Sewell, Bharan Giridhar, Ronald G. Dreslinski, David Blaauw, and Trevor N. Mudge. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, pages 496–507, 2013.

[309] Chia-Hsin Owen Chen, Sunghyun Park, Tushar Krishna, Suvinay Subramanian, Anantha P. Chandrakasan, and Li-Shiuan Peh. SMART: a single-cycle reconfigurable noc for soc applications. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 338–343, 2013.

[310] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. High performance pattern matching using the automata processor. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1123–1132, 2016.

[311] Indranil Roy, Ankit Srivastava, and Srinivas Aluru. Programming techniques for the automata processor. In *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*, pages 205–210, 2016.

[312] Indranil Roy, Nagakishore Jammula, and Srinivas Aluru. Algorithmic techniques for solving graph problems on the automata processor. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 283–292, 2016.

[313] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy II, Jack Wadden, Mircea R. Stan, and Kevin Skadron. An overview of micron's automata processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*, pages 14:1–14:3, 2016.

[314] Ke Wang, Yanjun Qi, Jeffrey J. Fox, Mircea R. Stan, and Kevin 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*, pages 689–699, 2015.

[315] Methods and systems for routing in a state machine. Patent US 9275290 B2.

[316] Micron. Method and system to dynamically power-down a block of a pattern-recognition processor., 2016. Patent US 9389833 B2.

[317] Helios Regular Expression Processor.

[318] Sudhir Satpathy, Korey Sewell, Thomas Manville, Yen-Po Chen, Ronald G. Dreslinski, Dennis Sylvester, Trevor N. Mudge, and David Blaauw. A 4.5tb/s 3.4tb/s/w 64x64 switch fabric with self-updating least recently granted priority and quality of service arbitration in 45nm cmos. In *ISSCC*, 2012.

[319] Omar Naji, Christian Weis, Matthias Jung, Norbert Wehn, and Andreas Hansson. A high-level dram timing, power and area exploration tool. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 149–156. IEEE, 2015.

[320] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R. Johnson, Stephen W. Keckler, Minsoo Rhu, and William J. Dally. Architecting an energy-efficient dram system for gpus. In *High Performance Computer Architecture (HPCA)*, 2017.

[321] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2009.

[322] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 93–102. IEEE, 2006.

[323] Xiaodong Yu and Michela Becchi. Gpu acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 18:1–18:10, New York, NY, USA, 2013. ACM.

[324] Bingsheng He, Qiong Luo, and Byron Choi. Cache-conscious automata for XML filtering. *IEEE Trans. Knowl. Data Eng.*, 18(12):1629–1644, 2006.

[325] Bruce W Watson. Practical optimizations for automata. In *International Workshop on Implementing Automata*, pages 232–240. Springer, 1997.

[326] George Anton Kiraz. Compressed storage of sparse finite-state transducers. In *International Workshop on Implementing Automata*, pages 109–121. Springer, 1999.

[327] Linux kernel. Huge Pages. 2017.

[328] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven design of router microarchitectures in on-chip networks. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 105. IEEE Computer Society, 2003.

[329] Pekka Kilpeläinen et al. Tree matching problems with applications to structured text databases. 1992.

[330] Yun Chi, Richard R Muntz, Siegfried Nijssen, and Joost N Kok. Frequent subtree mining—an overview. *Fundamenta Informaticae*, 66(1-2):161–198, 2005.

[331] James Clark. The Expat XML parser. http://expat.sourceforge.net.

[332] Apache Software Foundation. Xerces C++ XML parser. http://xerces.apache.org/xerces-c/.

[333] Ximpleware XML dataset. http://www.ximpleware.com/xmls.zip.

[334] XML Data Repository. http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html.

[335] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 158–169. IEEE, 2015.

[336] Jan Van Lunteren, Ton Engbersen, Joe Bostian, Bill Carey, and Chris Larsson. Xml accelerator engine. In *The First International Workshop on High Performance XML Processing*, 2004.

[337] Yuanwei Fang, Chen Zou, Aaron J Elmore, and Andrew A Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. In *International Symposium on Microarchitecture*, pages 55–68. ACM, 2017.

[338] Bharat Sukhwani, Thomas Roewer, Charles L Haymes, Kyu-Hyoun Kim, Adam J McPadden, Daniel M Dreps, Dean Sanner, Jan Van Lunteren, and Sameh Asaad. Contutto: a novel fpga-based prototyping platform enabling innovation in the memory subsystem of a server class processor. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 15–26. ACM, 2017.

[339] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. Dasx: Hardware accelerator for software data structures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 361–372. ACM, 2015.

[340] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 389–400. ACM, 2012.

[341] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In *Programming Language Design and Implementation*, pages 425–436, 2011.

[342] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz— open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[343] John Levine and Levine John. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009.

[344] INRIA. Lexer and parser generators (ocamllex, ocamlyacc). http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html.

[345] David Beazley. PLY (python lex-yacc). http://www.dabeaz.com/ply/index.html.

[346] Peter Ogden, David Thomas, and Peter Pietzuch. Scalable XML query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment*, 6(14):1738–1749, 2013.