Accelerated Pattern Recognition Processing Using Hybrid Spatial/Von Neumann 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

Jack Wadden

May 2018

Abstract

Newly available spatial architectures to accelerate finite automata-based pattern recognition processing have spurred a large amount of research and development of finite automata-based applications and accelerators. However, a lack of standard, open-source tools for automata processing application and architecture research has slowed the pace of innovation.

This dissertation first presents a new non-obvious application use-case for automata processing: efficient and high-quality pseudo-random number generation, further motivating research into automata processing acceleration. This dissertation then presents three new tools to enable and accelerate high-quality automata processing research. The first tool is a novel automata processing simulation, profiling, and optimization framework to accelerate automata application and architecture research. The second tool is a diverse benchmark suite of standardized automata graphs and inputs from published work to provide easy and fair comparisons among automata processing engines and architectures. The third tool is design-space exploration tool to help design and build spatial/reconfigurable automata processing architectures.

This dissertation then presents two novel architecture studies, enabled by the above toolchain. The first study characterizes spatial automata processor output processing and recognizes it as an important bottleneck to performance in real systems. We relax this bottleneck by designing a parameterizable reporting architecture to efficiently accommodate common-case behavior. The final study recognizes that automata states can have highly contrasting behavior. We dynamically profile automata graphs and find that some states perform orders of magnitude less computation than others. We propose a hybrid automata processing approach where highly active states are computed using spatial architectures, while rarely active states are offloaded to a von Neumann processor. This offloading can greatly reduce the resource pressure on the spatial automata processor, while maintaining a large proportion of acceleration potential.

Approval Sheet

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

Doctor of Philosophy (Computer Science)

Jack Wadden

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

Kevin Skadron , Adviser

Samira Khan , Committee Chair

Andrew Grimshaw

Gabriel Robins

Mircea Stan

Accepted for the School of Engineering and Applied Science:

Craig Benson, Dean, School of Engineering and Applied Science

May 2018

Acknowledgements

This dissertation would not have been possible without the support of a large group of people.

To the National Science Foundation (NSF), Semiconductor Research Corporation (SRC), and Achievement Awards for College Scientists (ARCS) for providing the resources to conduct this work, thank you.

To my parents, who offered me, and paid for, my every opportunity to succeed in life, thank you. Your sincere dedication to my education and success will be paid forward.

To my coaches and others who taught me the value of teamwork and perseverance, Laura Wadden, Pete Caragher, Dan Reid, Ben Lewis, and Peter Wells, thank you.

To those who instilled in me a love of science, technology, engineering, math, and computer science especially John Benson, Dave Dannels, Mark Vondracek, Russ Kohnken, Daniel Aalberts, Morgan McGuire, Duane Bailey, Sudhanva Gurumurthi, and Vilas Sridharan, thank you.

To my mentors at the University of Virginia, Mircea Stan, Samira Khan, and in particular my advisor Kevin Skadron, thank you. Your guidance is treasured and will always be remembered.

To my collaborators and friends at UVA, Nathan Brunelle, Tommy Tracy, Ke Wang, Mohamed El-Hadedy, Elaheh Sadredini, Chunkun Bo, Vinh Dang, Ted Xie, and Mateja Putic thank you for your help and support. And good luck.

And finally to loved ones who supported me throughout, and bore the brunt of my moods, brooding, and anxious, sleepless nights, thank you. Parker, I love you.

"Eye their circuitries to savor their intricacy and their varied palettes. First cousins to the cloisonne dish, the Persian carpet, the Vatican mosaics; what miniature panoplies live here. And live is *the word. For not only do these laboratory-found-objects look vital, they are juiced with creation. All the stuffs and junks and fabulous dreams of once sleepless men are shelved, stashed, and eye-droppered here....*

It's all a rare treat for the eye."

- Ray Bradbury

from his foreword in *State of the Art: A Photographic History of the Integrated Circuit*, by Stan Augarten

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As we enter the era of "big data," quickly filtering and extracting information from petabyte data-sets and tens to hundreds of gigabytes per second of network traffic has become increasingly important. One prominent methodology for identifying patterns in large files is to use regular expression pattern searches and finite automata. Regular expressions describe approximate patterns using a text-based representation language. To match a regular expression pattern over an input, regular expressions are usually converted to finite state machines or finite automata. Finite automata recognize patterns by traversing a set of states according to transition rules guided by symbols from an input text. If the automaton enters into a final report state, it has recognized a pattern in the text, and reports the location in the input stream and the pattern that was recognized. Figure 1.1 shows an example automata designed to recognize all correct spellings of the English word *color*, both the British and American spellings.

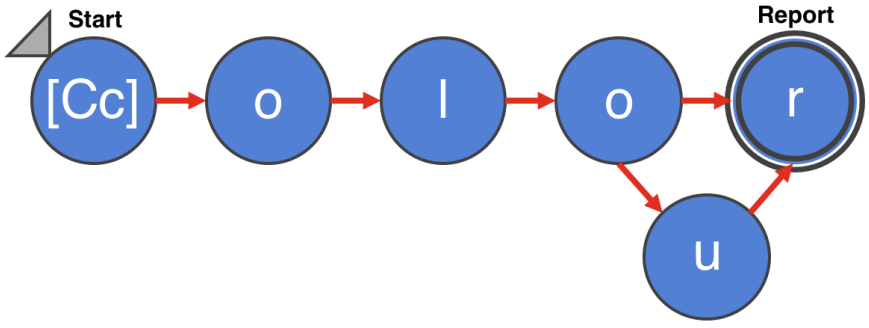

Figure 1.1: Automata designed to recognize all correct spellings of the English word color, both British and American spellings.

While automata processing is a well-known kernel in network analysis for deep packet inspection [4, 5, 6], and in security for virus detection [7], more recent research has identified new uses for automata processing

in machine learning [8, 9, 10], bio-informatics [11, 6, 12, 13], graph processing and mining [14], pattern-mining [15, 16, 17], entity resolution [18], natural-language processing [19, 20], and even high-energy particle physics [21]. These new targets for automata processing further motive the importance of research and development of high-performance automata processing software engines on von Neumann architectures, and novel hardware accelerators. This dissertation focuses on the characterization of automata application behavior in order to more efficiently accelerate applications using von Neumann architectures and/or spatial architectures.

On von Neumann architectures, simulation of large automata can require many hundreds, even thousands of parallel memory operations per input symbol. These memory operations can have little spatial and temporal locality, and can be likened to parallel "pointer chasing". Thus, high-performance automata processing on von Neumann architectures relies on the availability of large, fast caches to hide the performance impact of many long-latency, sequential memory accesses. Because automata can require many parallel accesses per input cycle, and can have graphs with hundreds of thousands of nodes that do not fit into on-chip caches, even today's server class CPUs struggle to meet the demands of modern automata-processing [22].

Because general-purpose CPUs struggle with automata processing, researchers have explored other, massively-parallel von Neumann accelerators as a possible solution. Graphics processing units (GPUs) offer a large amount of parallel resources, which can aid in hiding the latency of expensive DRAM accesses [23, 24, 25]. However, the often unpredictable memory accesses and varying amount of parallelism in automata simulation can be hard to map to a GPU's regular and rigid SIMD architecture [22].

While von Neumann CPU and GPU architectures struggle with accelerating the difficult parallel memory accesses and/or local memory capacity challenges inherent in automata processing workloads, spatial architectures (reconfigurable networks of processing elements such as FPGAs) excel. Spatial architectures can place-and-route automata states and connections within the reconfigurable fabric. Transitions between states can be implemented using point-to-point connections wired via an on-chip routing matrix. Once placed-and-routed, all automata states can compute in parallel, within a processor cycle, no matter how active the automata. This makes spatial architectures a natural target for the acceleration of finite automata processing.

Prior spatial automata acceleration has been investigated using FPGAs [26, 27, 28, 29, 30, 31]. Spatial techniques leverage the reconfigurable fabric to implement a network of automata states [30, 26, 32, 33, 28] implementing state matching logic in look-up-tables or embedded memories, and connections between states in the reconfigurable fabric.

Spatial execution offers potentially large benefits over von Neumann execution, but is only applicable if the automata graph can fit inside the available on-chip resources.

Micron's Automata Processor [34] (AP)–an announced, but not yet commercially available specialized spatial automata processing accelerator–uses a finite-automata-specific reconfigurable fabric to improve state density over techniques on more general purpose fabrics such as FPGAs [35]. However, because the AP is a spatial architecture, it is only usable if an automata graph can be placed and routed on-chip.

Note that the pros and cons of von Neumann and spatial architecture families are complementary. On one hand, von Neumann architectures can hold extremely large automata graphs in main memory, but perform poorly when computing large amounts of activity. On the other hand, spatial architectures have highly-limited capacity and routing constraints, but can efficiently compute arbitrary amounts of parallel activity.

In this dissertation, we hypothesize that *hybrid spatial/von Neumann automata processing* can realize a majority of the benefits of each architecture: executing small, densely-active portions of automata on the spatial side, and large, sparsely-active portions of automata on the von Neumann side. Intuition tells us that many automata should have this lopsided behavior, as automata are usually designed as input *filters*. In filter-style automata, a vast majority of candidate input matches are filtered out in the first levels of states (high activity), and states deeper in the filter are rarely used (low activity).

Depending on automata topology, dynamic behavior, and available spatial resources, hybrid architectures offer the following benefits over von Neumann or spatial architectures in isolation:

- **A reduction of spatial architecture requirements:** If 5% of automata states are required to compute 99.9999% of total work, a spatial core 20x smaller could efficiently handle the same problem. This reduction in spatial resources might enable small, low-cost integrated spatial cores to compute problems that typically would only fit on massive, expensive off-chip co-processors.

- **Increased capacity:** If a large spatial architecture is available, offloading 95% of automata states could make room for a larger problem size, or other co-processors for other kernels, greatly increasing the effective capacity of an existing spatial architecture.

- **Shorter place-and-route times:** Large graphs that utilize a significant percentage of spatial resources can have extremely long place-and-route times. Smaller graphs might lead to drastically shorter compile times, and lower power designs. Place and route can be a large bottleneck for automata processing in network security, where zero day exploits need to be patched as quickly as possible.

To evaluate this hypothesis, this dissertation focuses on three main research thrusts in the field of automata computing:

Thrust 1) This dissertation investigates novel, non-obvious use-cases and application domains for automata-based acceleration. If an application can be framed as an automata processing problem, it may

benefit from acceleration by both von Neumann and spatial architectures, and benefit from future research. New, important use-cases for automata processing also further motivates research in automata processing acceleration.

Thrust 2) This dissertation presents a suite of tools to support automata processing research. Prior to this work, automata processing tool-chains were either fractured, closed source, not freely licensed, or not geared towards spatial architecture research. In order foster new research in the field of automata processing, we present a new suite of open-source tools that allow development of new application domains, finite automata transformations and optimizations research, novel automata processing accelerator designs, and fair evaluation of new automata processing architectures.

Thrust 3) Using these above tools, this dissertation presents two architectural studies to support the main hypothesis: a study of the performance impacts of reporting in spatial automata architectures, and an evaluation of the potential benefits of hybrid CPU/FPGA automata processing. Reporting is often ignored in modern spatial automata processing research. Every time a match is encountered on chip, that match needs to be packaged and exported off chip. If match rates are high in the automata, report handling could greatly affect performance. We characterize reporting behavior in automata benchmarks and show that reporting can have a large impact on performance in real systems if reporting architectures are not carefully designed. This characterization, and impact study motivates the design of a new standard reporting architecture for spatial automata processors. Such reporting architectures are of utmost importance in a hybrid architecture, where communication between spatial and von Neumann cores is on the critical performance path.

Together, the developed tools, and reporting architecture study enabled us to design a hybrid spatial/von Neumann automata processing system, and evaluate the benefits of such a hybrid architecture. We present a hybrid architecture design, and an algorithm for determining appropriate partitioning points to balance resource reduction, communication costs, and von Neumann co-processor performance.

The results of these investigations may lead to transformational insights into how certain important application domains are best accelerated, and may also have far reaching effects into FPGA-based accelerator design and spatial computing.

1.1 Contributions

This dissertation first investigates a new, non-obvious use-case for automata processing: agent-based simulation and pseudo-random number generation [36]. By recognizing that probabilistic input to automata produces probabilistic automata behavior, we design non-deterministic finite automata to implement Markov Chains and evaluate their performance on current architectures. We then identify bottlenecks to performance and

propose hardware modifications for future architectures to greatly increase pseudo-random number generation throughput.

We present the following contributions:

- The design and investigation of fair, fully connected Markov Chains (N-sided dice) and their ability to generate high-quality pseudo-random output and other types of structured random behavior.
- An investigation of the performance of Micron's Automata Processor for high-quality pseudo-random number generation and a comparison of the performance and power efficiency of this technique to state-of-the-art pseudo-random number generation algorithms on GPUs.

This dissertation then presents three new, open-sourced tools to increase the ease of access and pace of automata processing research:

The first tool–VASim–is an open-source automata processing sofware development kit. VASim is, to the best of our knowledge, the first extensible, general-purpose automata processing framework that combines automata simulation, optimization, transformation, and performance modeling into one, unified and open source code base. This framework enables easy prototyping, debugging, simulation, profiling, and analysis of automata-based applications and architectures. VASim is flexible enough to support research and development of new automata processing techniques, and fast enough to compete with state-of-the-art automata processing engines.

The second tool–ANMLZoo–is a diverse suite of 14 standardized automata graphs and inputs that are derived from both well-known, regular-expression-based applications as well as newly-discovered application domains. We standardize each benchmark for fair cross-architecture evaluations (especially considering spatial/von Neumann architecture differences). Using these benchmarks, we show that spatial architectures tend to perform much better than von Neumann architectures. However, spatial architecture capacity is very sensitive to automata size and topography and cannot place-and-route automata states with complex connectivity.

The third tool–Automata-to-Routing(ATR)–is an open-source toolchain that can place-and-route any homogeneous automata application on a parameterizable spatial automata processing architecture [37]. ATR builds upon both ANMLZoo and VASim, as well as a well-known existing FPGA architecture research framework VPR [38]. The ATR toolchain enables research on novel spatial automata processing architectures, and can be used to evaluate design choices in existing, commercially available processors.

This dissertation then presents a characterization of reporting behavior in the ANMLZoo benchmark suite for spatial automata processing architectures. To the best of our knowledge, we are the first to characterize reporting behavior across a wide variety of automata benchmarks and recognize its importance in automata-

processing application and architecture design. We first characterize automata-processing output requirements using ANMLZoo, evaluate the impact of reporting on the Micron Automata Processor using a validated, parameterizable, cycle-accurate simulator. We then present two techniques–an automata transformation, and a new reporting architecture design–that improve reporting efficiency.

This dissertation then presents a study of the potential benefits of hybrid spatial/von Neumann automata processors. We first develop a profile-driven partitioning algorithm that identifies regions of automata graphs that are responsible for a large proportion of computation. We then present a characterization of automata behavior in the ANMLZoo benchmark suite. We show that most automata in the benchmark suite are filter-style automata, and contain large proportions of activity in a small proportion of states. We then present a realization of a hybrid automata processing system targeting a commercially available hybrid CPU/FPGA architecture. We leverage the open-source spatial and von Neumann automata processing engines to compose our hybrid system. Our experiments show hybrid automata processing enables large reductions in spatial resource requirements (up to 97%) and reduction in compile times (up to 3.6x), with low added performance overheads (less than 6.1%). We also show that hybrid architectures can use different algorithms to cooperate on one application kernel. The spatial architecture can use small, easily routable automata to pre-filter problem spaces and pass smaller problem sizes to the best performing CPU or GPU algorithms.

1.2 Organization

The remainder of this dissertation is organized as follows:

Chapter 2: Background introduces automata processing, discusses automata-related theory, prior approaches to automata processing, and the architecture of Micron's Automata Processor.

Chapter 3: Generating Efficient and High-Quality Pseudo-random Behavior on Automata Processors presents a novel use-case for accelerated automata processing: pseudo-random number generation. Chapter 3 also discusses other use-cases and techniques for taking advantage of random behavior in automata.

Chapter 4: An Open-Source Framework for Automata Processing Research presents the Virtual Automata Simulator or VASim. VASim is an open-source framework for automata processing research and development.

Chapter 5: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures presents ANMLZoo, a diverse benchmark suite for fair evaluation of various automata processing engines.

Chapter 6: An Open-Source Toolchain for Design-Space Exploration of Spatial Automata Processing Architectures presents Automata-to-Routing or ATR. ATR is an open-source tool for design-space exploration of spatial automata processing architectures.

Chapter 7: Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures presents a characterization of reporting behavior in the ANMLZoo benchmark suite. This chapter then presents an impact study of reporting behavior on the performance of existing spatial automata processing architectures, and proposes automata transformations and architecture changes to reduce the impact of reporting on performance.

Chapter 8: Hybrid Spatial/von Neumann Automata Processing presents the intuition behind hybrid spatial/von Neumann automata processing. This chapter evaluates whether automata can be partitioned such that a large proportion of automata computation can be accounted for with a small number of states. We then then evaluate the performance trade-offs of various partitions, and benefits of hybrid execution targeting a real-world hybrid CPU/FPGA architecture.

Chapter 9: Conclusions summarizes the dissertation and discusses the implications of this work and potential future directions of research.

Chapter 2

Background

2.1 Automata Processing

Informally, a finite automaton is defined as a directed graph of node "states" with transition rule edges between states. Each transition rule is guided by a globally visible input symbol read from a symbol tape. Each automaton has one or more start states that initiate computation. States that are currently performing computation are said to be *enabled*. During a compute cycle, each enabled state compares the current symbol on the input tape with its transition rules. If the symbol triggers a transition rule, the state *activates*, and transitions to each state where a rule matched. Computation proceeds with each symbol on the input tape being considered, and each state in the graph computing transitions.

Each automaton also has one or more *final*, *accept* or *report* states. If a report state activates, the ID of the report state and the current position in the input symbol tape are recorded. Automata are usually designed to report when a specified pattern is seen on the input tape.

In the classical non-deterministic finite automata model [39], processing elements can consider any number of transition rules [40]. *Homogeneous* finite automata (also known as Glushkov, or Position automata), are a restriction on classical finite automata where all incoming transitions to a state have the same (homogeneous) matching rule. This property is desirable for a variety of reasons. Because each transition into a state occurs using the same rule, transition rules can be computed once for all incoming transitions. Thus, homogeneous finite automata states can be thought of as boolean circuit elements. Each boolean input is a potential transition to the homogeneous state. The state computes the matching function once for any incoming transition, and accepts the transition if the matching function is true. For the next symbol, the homogeneous state then broadcasts its match or "activation" to all child elements and the process continues. Homogeneous

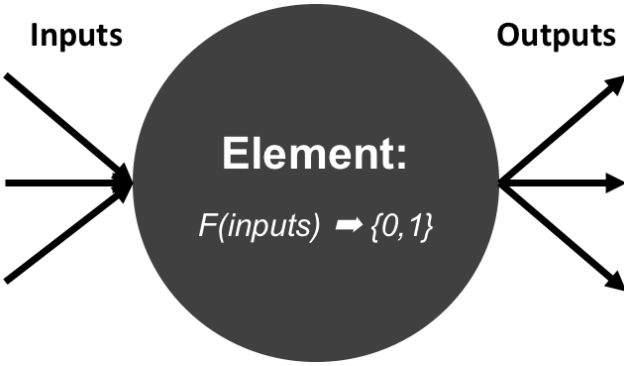

Figure 2.1: A generic homogeneous automata processing element. All elements compute a boolean function based on input signals. The result of this boolean function is then broadcast to element children.

automata are easy to reason about because transition computation and state storage are combined into a single element.

Because homogeneous automata can be thought of as connected graphs of boolean processing elements, some automata models also consider boolean logic elements other than the homogeneous state, for example AND gates. These elements perform computation on signals between symbol cycles, and do not consider the global input.

A picture of a generic boolean homogeneous automata element is shown in Figure 2.1.

2.1.1 Deterministic Finite Automata

Via transition rules, finite automata can enter into multiple states during a symbol cycle. Finite automata that can be in more than one state are referred to as *non-deterministic finite automata* or NFAs. The term non-deterministic here refers to the parallelism of the finite automata, rather than a stochastic sense of the word. *Deterministic finite automata* or DFAs are a subset of finite automata that guarantee only one state transition per input symbol. NFAs and DFAs are equivalent in computational power, but have trade-offs that can affect the performance of simulation. Because NFAs can be in more than one state for a given input symbol, they may require more computation per input symbol. DFAs on the other hand only ever have to compute the transition rules for a single state, and generally require less computation. The downside to DFAs is that they can require exponentially more states to represent. The DFA can intuitively be thought of as a unary representation of its equivalent NFA. Each DFA state represents a *configuration* of the corresponding NFA state. Because each NFA state can be occupied or not, the total number of possible NFA configurations is $\Theta(2^N)$ where N is the number of NFA states. Thus, converting an NFA to a DFA can take exponential time and space relative to the number of NFA states.

Because of this exponential "blow up," and the large size of NFAs in real-world automata processing, DFA conversion is usually not considered as a practical acceleration technique. Prior work has explored using a hybrid approach where parts of NFAs that have lots of parallel behavior are converted to DFAs, while the rarely active tails of automata are kept as NFAs [30]. Thus, the hybrid finite automata will on average reap the benefits of DFA transitions, but without paying the penalty of exponential state blowup. We use this same intuition in Chapter 8 to reduce the resource pressure on spatial automata processing architectures.

2.1.2 Regular Expressions

Informally, regular expressions (commonly shortened to regex) are a compact language for representing patterns in strings of characters. In this paper, we are primarily concerned with perl compatible regular expressions (PCRE) [41] because it is the most widely adopted format.

Regular expression patterns represent a certain set of strings. This set of strings is referred to as the *language* of the regular expression. Regular expressions are only powerful enough to represent a subset of all languages, referred to as *regular languages*. All regular expressions can be described by equivalent finite automata and vice versa, making regular expressions and finite automata equivalent in power. Regular expressions are said to be *generative*, meaning that they define a set of rules that can be used to generate strings in the corresponding regular language. Unlike *generators*, finite automata are *recognizers* and are machines designed to accept all strings in a language. Thus, we usually convert regular expressions to finite automata in order to find regular expression patterns in input strings.

Besides simple matching, PCRE regular expressions add a few additional concepts that greatly expand the expressiveness of the language.

Grouping Or: Parenthesis indicate a grouping of multiple different expressions. Each expression is separated by a | indicating that any of the expressions within the parenthesis can match in parallel. For example, $(gr(a|e)y)$ recognizes both the american and british spelling of the color gray.

Wildcards: Wildcards are additions to represent arbitrary characters in an input string. The . for example is meant to represent a single character of the input string, although this symbol is often omitted when used along with quantifiers.

Quantifiers: Quantifiers act on the previous expression and define repeating characters or sequences. The symbol ? specifies that there are must be either zero or exactly one of the previous expression. For example, $(colou?r)$ matches both the American and English spelling of the word color. The $*$ symbol, also known as the "Kleene star," matches zero or any number of the previous expression. And the $+$ symbol matches at least one of the previous expression. For example, $((B|b)oo+!*)$ will recognize the exclamation of

any ghost. Ranged or interval quantifiers put restrictions on the number of characters. For example, if we wanted to restrict the number of o's in Boo to be between 2 and 10, we could represent this with the regular expression $((B|b)oo\{2,10\}!*)$.

Character Classes: Character classes represent sets of characters and are shorthand for groupings of individual characters. For example, $(gr[ae]y)$ and $(gr(a|e)y)$ are equivalent. Character classes can also be described as ranges of characters. For example $[a-z]$ is all lowercase characters, while $[0-9]$ represents any digit.

Anchors: The $\wedge$ symbol anchors an expression to the beginning of a line or input sequence. For example $(\wedge[A-Z])$ will recognize all lines that begin with an uppercase character. The $ symbol behaves in the same way but anchors an expression to the end of a line or input sequence.

2.2 von Neumann Automata Processing

von Neumann computers are reconfigurable processing elements coupled with large memory stores. von Neumann computers perform computation by storing programs and data in a memory store. Instructions from the program are fetched from memory, and configure the processing element to compute a certain operation. If needed, data is fetched from memory to complete the operation. Results of the computation are then stored back out to memory. Performance of von Neumann computers relies heavily on efficient loads and stores to the memory. Memory performance trends have traditionally trailed those of processor performance, thus memory speeds have become a bottleneck for many applications; this problem has been dubbed the "memory wall" [42] and the memory bottleneck is often referred to the "von Neumann Bottleneck."

On von Neumann architectures, automata are usually computed using rule look-up tables. Each enabled state is considered in a loop. If an enabled state matches, child states in the graph are fetched and enabled. Because of the possibly unpredictable and large number of memory accesses, automata processing is usually bottlenecked by the von Neumann memory system. Caches can help, but automata in general not have the spatial or temporal locality necessary to efficiently use unmanaged caches. Thus, the runtime of automata processing on von Neumann computers is highly correlated with the average number of active states (active set)–which determines required memory bandwidth–and the size of the set of frequently visited states (visited set)–which determines the size of the cache or scratchpad required to efficiently serve all memory requests [22].

Therefore, most high-performance von Neumann automata processing systems rely heavily on optimizations and shortcuts that reduce the total number of memory accesses, or increase the efficiency of memory accesses [43, 44, 30].

Hyperscan [43] is a commercial-grade, open-source automata-based regular expression evaluation engine. Hyperscan analyzes automata and employs hybrid NFA, DFA, and literal exact match techniques to efficiently compute regular expression matches. While Hyperscan is high-performance and open source, it is built specifically for regular expression processing (rather than generic automata processing), and is a complex code-base that is not easily modified. Therefore, it is not a suitable research platform for automata processing research.

RE2 [44] is another open source automata-based regular expression evaluation engine. RE2 explores an NFA and dynamically constructs and caches DFA states. If RE2's DFA state cache grows too large, it defaults back to NFA execution. RE2's approach excels when automata are small (e.g. a small set of search rules), but becomes inefficient when the automata are very large because dynamic DFA construction becomes extremely expensive. In our experience, both Hyperscan and VASim (presented in Chapter 4) perform better than RE2 for even moderately sized rulesets. RE2 is also designed for regular expression processing and is therefore not a suitable research platform for experimental automata processing.

Becchi et al. [30] provide an open source, automata-based regular expression evaluation engine. This tool provides algorithms for constructing and gathering dynamic statistics about automata, but does not support a high performance simulation. Becchi's tool is most suitable for research, but is difficult to modify, and does not easily support the addition of new, hypothetical processing elements. Becchi explores many optimizations of finite automata and their impacts on performance [45, 46, 30].

GPU-based acceleration of automata processing has also been investigated [23, 24, 25, 22]. GPU-based acceleration can see improvements over CPU-based automata computation. However, GPUs are still von Neumann computers and rely on efficient parallel memory accesses to improve performance over CPU-based computation. If parallel memory accesses are not coalescable, or if parallelism is unpredictable, GPUs struggle to gain large performance advantages over CPUs [22]. Furthermore, extensive performance tuning is sometimes required to gain appropriate speedups [22, 25].

Many specialized von Neumann accelerators have been developed that gain efficiency over more general purpose CPU and GPU-based techniques. Kaneta et al. [31] implemented a von Neumann inspired design in an FPGA using the concept of bit parallelism [47]. Mitra et al. [48] convert PCRE rules from Snort IDS rules [4] to "opcodes" that are then processed in specialized cores for individual regular expressions. Gogte et al. [49] and Krishna et al. [50] design processors to compute regular expression matches using small, optimized lookup tables. Fang et al. [51] design a custom architecture with small von Neumann cores connected to large scratchpad memories.

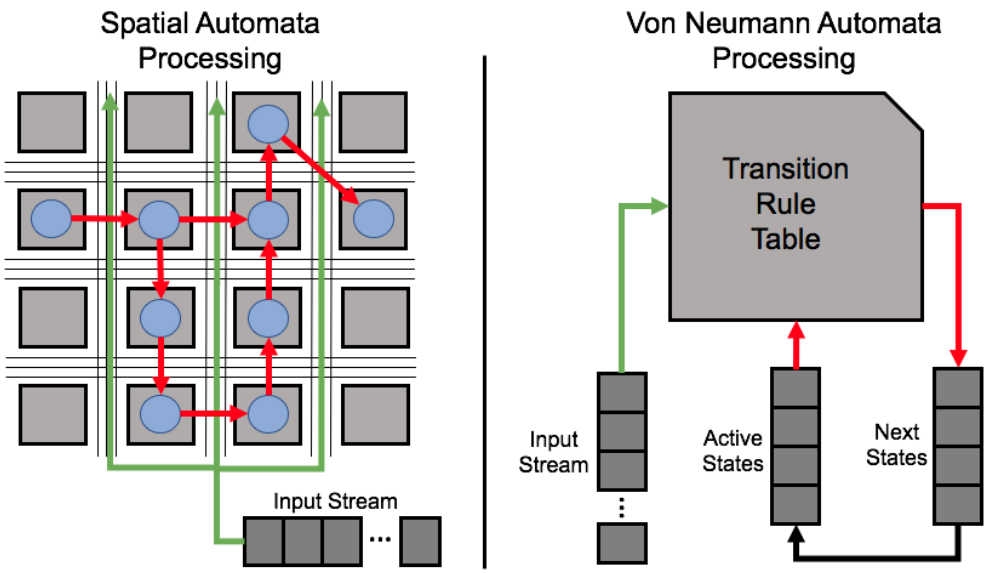

Figure 2.2: Spatial automata processors "lay out" automata states in a reconfigurable network of processing elements and broadcast input symbols to all states. States compute transition rules in parallel and transitions are propagated via the routing matrix. Von Neumann automata processors simulate automata by keeping track of a list of active states and using rule tables to compute transitions for the next input.

2.3 Spatial Automata Processing

In spatial automata processing, automata graphs are placed-and-routed in a reconfigurable fabric—similar to how logic gates in a circuit graph are placed-and-routed on an FPGA. Spatial processors offer a certain number of processing elements in a reconfigurable fabric and automata graphs are directly emulated in the fabric. Input symbols are broadcast simultaneously to all states in the graph, and all states compute and communicate in parallel within a single cycle. This is in contrast to von Neumann techniques discussed in the prior section that store graph information in some sort of memory, and simulate state transitions using transition rule lookups. For highly active automata, spatial architectures can be several orders of magnitude faster than von Neumann architectures [22].

Figure 2.2 shows at a high level the differences between these two approaches.

However, spatial architectures have limited capacity and routing resources. If an automaton has more states than are supported by the architecture, or if the automaton graph topology is too complex to be routable, the graph cannot be computed.

2.3.1 FPGA-based Spatial Automata Processing

A large amount of prior work has explored spatial automata processing in general purpose reconfigurable hardware, i.e. field programmable gate arrays (FPGAs) [32, 33, 45, 29, 26, 52, 53]. FPGAs offer a large

amount of flexibility in the design of regular expression processing engines and allow spatial implementations of automata graphs. Most prior work in designing spatial automata processing engines on FPGAs can be divided into logic vs. memory-based matching. Both techniques are described below.

Logic-based matching:

In the logic-based approach each automata state is represented by a simple flip-flop, and transition rules between states are computed using combinational decode logic placed and routing in an FPGAs reconfigurable fabric. A canonical example of this scheme is Prasanna et al. [32]. Other techniques use and optimize this approach [45, 33, 28, 52, 53].

Memory-based matching:

Because transition rule computation in logic-based techniques use custom decode logic, logic-based designs cannot be dynamically reconfigured without an expensive full compilation and place-and-route of the automata graph [31]. It is often desirable to quickly reprogram the same automata graph with different transition rules based on the results from prior computation [11, 18, 15, 16]. To enable this feature, memory-based transition rule computation uses embedded memories in the FPGA's reconfigurable fabric (rather than custom decode logic) as large look-up-tables for transition rule computation [53]. Embedded memories can be reprogrammed separately from the logic, and thus offer quick transition rule modification. Kaneta et al. refer to this property as "dynamic reconfigurability" [31, 34].

The REAPR FPGA Automata Engine:

More recently, the REAPR [53, 54] project has made both logic-based, and memory-based spatial acceleration techniques available as an open-source framework. The original REAPR paper [53] compared logic and memory-based techniques and showed that performance is similar, but that memory-based automata, which are soft-reprogrammable, consume more power. We use the logic-based REAPR automata engine for our FPGA-based evaluations in later chapters.

2.3.2 Micron's Automata Processor and SDK

Micron's Automata Processor [34] is an "automata specific" memory-based spatial architecture that uses a DRAM-based reconfigurable fabric to gain increased state density over more general-purpose reconfigurable fabrics [34, 35]. The AP re-purposes the parallel address decode logic and parallel bit look-up inherent in DRAM arrays as a large set of dense parallel rule look-up-tables. Coupled with a small amount of logic,

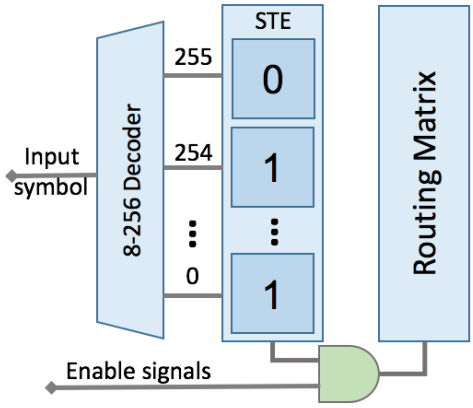

Figure 2.3: An Automata Processing state transition element or STE. STEs repurpose memory columns as 8-input/1-output look-up-tables used for matching symbols from the input stream. Coupled with state logic, STEs implement a homogeneous automata state. Similar to an FPGA, inputs and outputs to STEs are routed through a reconfigurable routing matrix.

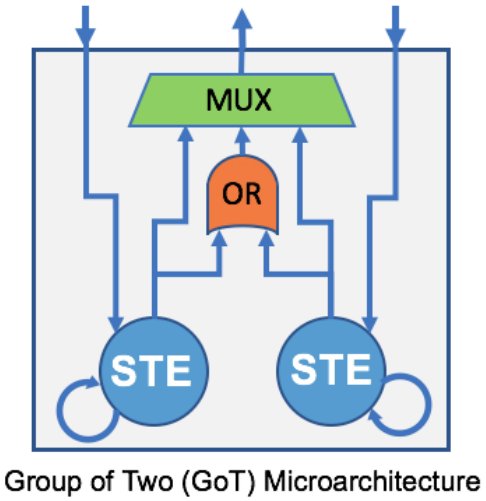

Figure 2.4: Groups-of-Two (GoT) [2] can hold two STE nodes. Each STE has an input, can enable its pair STE, or enable itself. The output enable signal is chosen between the left STE, right STE, or the logical OR of both outputs.

each DRAM column can encode a homogeneous automata state transition rule and state storage bit, and is dubbed a "state transition element" or STE. An illustrative diagram describing the micro-architecture of an STE is shown in Figure 2.3.

STEs can be configured to *report on activation*, producing a 1-bit output. This converts an STE to a "final" state, and is analogous to accepting or matching an input string in a classical NFA.

An STE acts as a type of logic gate. If an STE received an input signal on the previous cycle, it checks its DRAM column look-up-table to see if the current symbol is in its symbol set. If the STE matches the current input symbol, it activates and transmits an output signal to child STEs. Each STE receives and transmits

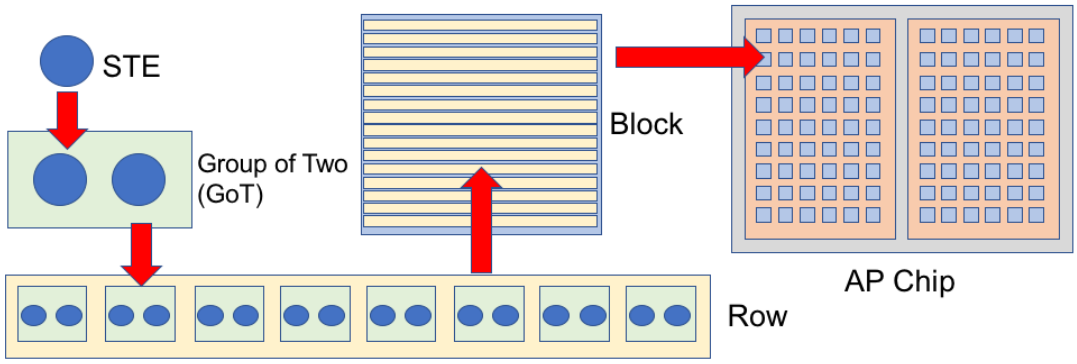

Figure 2.5: Elements in the Micron AP architecture. State Transition Elements (STEs) are groupted into "Groups of Two" (GoTs). Eight GoTs form a Row. Sixteen Rows form a Block. Ninety-six blocks form a half-chip. Two distinct half-chips form the AP. We choose the Row as an appropriate tile for the ATR model AP because it is the first element that has access to the AP's routing matrix.

signals to other STEs via a hierarchical, on-chip routing matrix. Input symbol broadcast, matching, and enable-signal broadcast happen within a single AP cycle.

The first-generation AP is divided into two disjoint "half-cores." Each half-core has 96 blocks. Each block has 16 Rows and thus each AP chip has 3,072 Rows. Each Row has 8 "groups-of-two" or GoTs. Each GoT has two STEs. GoTs have two input ports but only one output port. The output port is MUXed to select between either STE's output or the logical OR or their output signals. The micro-architecture of a GoT is detailed in Figure 2.4. The hierarchy and organization of AP structures is shown in Figure 2.5.

STEs "report" by sending their signal from a GoT to one of two reporting ports in a Row. Each Row's reporting ports are statically routed to a particular portion of a reporting region [1] and do not use the general purpose routing fabric. Each half-core of the AP chip has three reporting regions. Each region has the capacity to support reports from 1,024 GoTs–2 reports allowed from each row within 32 consecutive blocks. The reporting architecture of the D480 AP is discussed in more detail in Chapter 7.

The AP model extends the homogeneous NFA execution model by adding boolean logic elements, which provide AND, OR, NOT, $NAND$, NOR, sum of products, and product of sums capability. The AP model also provides special counter elements (similar to those proposed in the literature [30]) which only activate after a pre-set threshhold of input activations is reached. Neither of these elements (boolean and counter) are available in classical NFAs and they expand the set of languages recognized by the device past regular languages.

Micron's AP Software Development Kit

Micron's Automata Processor is accompanied by a software development kit to help automata application developers prototype, debug, and construct large and complex automata [55]. The SDK provides a graphical user interface that allows a developer to manipulate finite automata and visualize execution. C, Python, and Java bindings are also included to allow developers to programmatically define large automata networks.

The AP SDK can export automata networks to an XML-derived automata language called *Automata Network Markup Language* or ANML. ANML is an abstract intermediate representation of automata, and allows automata networks to be saved and passed between different tools. Micron also provides tools for optimizing and compiling (placing and routing) ANML files for execution on the AP as well as emulation of compiled automata.

Chapter 3

Generating Efficient and High-Quality Pseudo-Random Behavior on Automata Processors

Spatial automata processors such as Micron's Automata Processor (AP) are extremely powerful and efficient pattern matchers, and have been shown to provide large speedups over von Neumann architectures such as CPUs and GPUs for massively parallel regular expression, rule-based pattern mining applications [22]. However, the exact use cases for accelerated automata processing outside of the well-known area of regular expression processing, and its advantage over other architectures such as CPUs and GPUs remains an open research question. Thus, new application domains that benefit from automata processing should be investigated.

This chapter explores a novel application of the AP: a high-quality source of pseudo-random behavior for pseudo-random number generation (PRNG) [1] and other potential simulations that can be defined with Markov chains (such as agent-based models). We aim to achieve efficient, high-quality, massively parallel, pseudo-random behavior by constructing and running many parallel Markov chains, simulated using NFAs.

Instead of simulating automata transitions using conventional input data (e.g. a packet stream), we consider simulating automata transitions using random or pseudo-random input. Because transitions between states in automata are conditional on the input stream, a probabilistic input stream immediately provides probabilistic automata transitions, *even though the transition rules are deterministic*. Thus, probabilistic automata, including finite state Markov chains, can be emulated using automata.

[1] Nathan Brunelle contributed significantly to algorithmic design and theoretic justification for true PRNG in this chapter.

Using this intuition, we develop a novel method for creating high-quality pseudo-random behavior using Markov chains modeled by NFAs, and accelerated using Micron's Automata Processor. We call this technique *AP PRNG*. We use parallel Markov chains to model rolls of fair dice, and then combine the results of each roll into a new pseudo-random output string. By combining the output of parallel rolls, driven by a single stream of random or pseudo-random input symbols provided by a host processor (e.g. the system CPU), we can construct new pseudo-random output hundreds of times larger than the input used to drive transitions on chip.

However, because we emulate Markov chains using NFAs with fixed transition functions, *any* non-trivial number of parallel Markov chains that consume the same input will produce output that is eventually correlated and patterned. Put another way, some output configurations of the states of Markov chains will be more probable than others, therefore, pseudo-random output will eventually appear non-uniform.

Thus, we also study the quality of pseudo-random behavior generated by AP PRNG. We identify that AP PRNG can generate high-throughput, high-quality pseudo-random output. This result is a strict prerequisite for evaluating the reliability of any simulation accelerated using shared-input parallel Markov chains. If AP PRNG cannot generate high-quality pseudo-random behavior, it should not be used as a basis for simulation.

3.1 Random and Pseudo-random Number Generation

3.1.1 Random and Pseudo-Random Number Generation

Pseudo-random number generation (PRNG) lies at the core of simulation and cryptographic applications. For example, Monte Carlo methods are pervasive simulation tools in physical and social sciences and rely on continuous random sampling to drive simulation of unpredictable processes. Monte Carlo simulations were among the first use cases for computers [56], and are arguably some the most important algorithms ever invented. Because fast and high-quality pseudo-random number generation is on the critical path of these applications, developing fast and high-quality PRNGs is of the utmost importance to improving the quality and speed of any computational science.

Today, while there are many PRNG algorithms, not all are created equal. No matter the method for generation of pseudo-random numbers, the harder it is to distinguish pseudo-random numbers from a truly random stream of numbers, the better it represents a truly random number stream and the higher its *quality*. The literature has adopted two avenues for evaluating the quality of PRNGs. Cryptographic applications require a PRNG to reduce to a hardness assumption, some problem widely believed intractable. Because we

aim to motivate the use of AP PRNG for Monte Carlo-style simulation, and not cryptographic applications, the literature suggests we evaluate random behavior empirically using *statistical tests*.

Statistical tests distinguish random from pseudo-random input by searching for over-prevalent or under-prevalent patterns. The most comprehensive and stringent collection tests are the BigCrush test battery from TestU01 suite [57], which includes the functionality of the Knuth tests [58], DIEHARD [59], and the NIST statistical test suite [60]. A test in the suite fails if it identifies a property of the pseudo-random sequence that should not exist in true randomness. If all tests pass, the pseudo-random numbers have been deemed indistinguishable from true randomness.

3.1.2 State-of-the-Art Parallel PRNG Algorithms

It is desirable that psuedo-random number generation algorithms have two properties:

1. The algorithm should produce high-quality pseudo-random behavior. While the quality requirements for pseudo-random behavior are application dependent, highest-quality output (i.e. output indistinguishable from random) is usually desired.

2. The algorithm should be performant on modern computer architectures. For von Neumann machines, this usually means a small number of instructions and a small amount of local state are required to generate each byte of pseudo-random output.

Mersenne Twister [61] is a popular, and well-known PRNG algorithm used pervasively in Monte Carlo simulations. However, it is famously difficult to port onto parallel architectures that provide limited storage per thread (e.g. GPU, XeonPhi) because each instance of the algorithm requires a large amount of state [62]. When porting simulations to many-core architectures or GPUs, cache and scratchpad memory become scarce resources per parallel thread, meaning that scalable, parallel PRNGs ideally need to use minimal state to avoid low core utilization of computational resources on chip [62]. Not only is Mersenne Twister hard to parallelize, it also fails many tests in TestU01's BigCrush test battery [62], and is thus a dubious choice as the gold standard PRNG for future high-performance scientific simulation. Increasingly, scientists are moving away from Mersenne Twister in favor of algorithms that produce higher-quality pseudo-random behavior, and are easier to parallelize.

Philox [63] is a newer parallel PRNG algorithm that drastically reduces the amount of state required per instance of the algorithm. Philox also relies exclusively on integer computation, reducing the complexity of required hardware, and improving performance over floating-point-based algorithms. Because of the small state per thread requirements, and integer-based computation, Philox is amenable to acceleration on

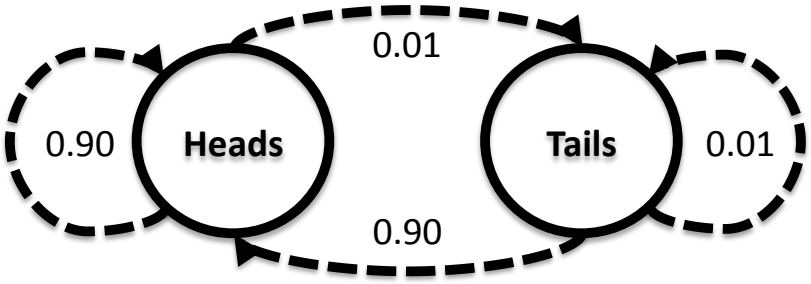

Figure 3.1: A simple Markov chain that simulates an unfair coin toss with two states: *Heads*, and *Tails*. Transition probabilities between these states are *unfair*, i.e. the probability of transitioning to *Heads* is different than *Tails*.

SIMD-based graphics processing units. Research has reported throughputs of up to $145GB/s$ of on-board PRNG generation (exclusive of board I/O) on on an NVidia GTX580 [63]. In our own tests, we were able to achiev $85GB/s$ with no performance tuning, using the built-in Philox4×32_10 implementation in the CUDA curand library on an NVidia K20 GPU.

Importantly, Philox also passes all tests in the BigCrush test suite. Because Philox is fast, easily parallelizable, and passes all tests in the BigCrush test suites, we consider this algorithm the current state-of-the-art in both performance and quality.

3.2 Using Markov Chains to Generate Pseudo-Random Behavior

3.2.1 Markov Chains

In informal terms, Markov chains are automata with probabilistic transitions between states. To be formally considered a Markov chain, transitions in the automaton must 1) be stochastic processes (i.e. they occur with some probability), and 2) respect the *Markov property*, which states that every probabilistic transition depends only on the current state, and is not influenced by memory of prior states. An example Markov Chain describing an unfair coin is illustrated in Figure 3.1. We focus on emulation of discrete time, finite state Markov chains in this thesis, and we leave research into emulation of other models as future work.

Markov chains are defined by stochastic *transition matrices*, which hold all transition probabilities from a start state (row) to a end state (column). Each row of the transition matrix must be *stochastic*, i.e. each element of the row must add up to 1. Logically, this makes sense because we must always make some transition in a time step, even if it is to the current node. The transition matrix for the unfair coin example in Figure 3.1 is shown in Table 3.1. Note that to build a fair coin, we simply set all probabilities in the transition matrix to $1/NumStates$.

	To Heads	To Tails
From Heads	0.90	0.10
From Tails	0.90	0.10

Table 3.1: Stochastic transition matrix of a Markov chain representing an unfair coin.

3.2.2 Using Markov Chains to Generate Pseudo-Random Behavior

Just like the unfair coin example in the previous section, we can construct fair Markov chains that have an equal probability of entering into any one state by setting all transition probabilities in the stochastic matrix to be equal.

Each state is then encoded with a binary label, and on every time step, the binary encoding of the state of the Markov Chain corresponds to a random output. Naively, number of random bits generated by a fair, fully connected Markov chain is $floor(log2(N))$. If the number of states in the chain is not a power of two, we can only use the bits that are equally likely to appear in the output. Thus, we only consider fair, fully-connected Markov chains where $N = 2^k$. Because the example Markov chain has four states ($N = 4$), each time-step generates $log2(4)$ or 2 random bits. These bits can then be concatenated to form random bytes, integers, or floating point numbers.

Multiple Markov chains can be run in parallel to generate more random output per time-step. As long as the transition probabilities in each parallel Markov chain are independent, an arbitrary amount of parallel output can be generated by simulating any number of parallel Markov chains. We use this intuition to generate high-throughput PRNG using the simulation of parallel Markov chains. The next chapter describes how we simulate Markov chains using automata with deterministic transitions.

3.3 Simulating Markov Chains Using Finite Automata

Markov chains and finite automata have similar computation models: they both have states with transition rules between states. However, transitions between two states in a Markov chain are governed by a simple probability, while finite automata transitions are governed by a transition rule and an input tape. Note that the transitions in a finite automata actually do have a probability: the probability that a symbol occurs in the input stream that will cause a transition.

The probability of a transition from node A to node B on symbol c is the probability that c occurs in the input stream. Thus, even though the transitions are deterministic given a certain input symbol, tuning the probability distribution of the input can cause that transition to become probabilistic. Using this intuition, we can simulate Markov chains by randomizing the input stream, and carefully constructing automata transitions.

Assuming a uniformly distributed input stream of symbols, we can easily construct finite automata transition with a certain probability by making sure that the transition rule is triggered by a proportion of input symbols that matches the transition probability. For example, if we want to model a Markov chain transition that occurs with 50% probability, we construct a finite automata transition rule that is triggered by 50% of symbols that could be seen in the input. If symbols occur uniformly at random, this transition has a 50% probability of occurring!

3.3.1 Markov Chain to NFA Construction Algorithm

Consider the unfair coin example described in the previous section and shown in Figure 3.1. To produce the probabilistic transitions of the Markov chain in a finite automaton we first assume the input symbol stream is a source of uniformly distributed random symbols.

An NFA simulating a Markov chain can be constructed using the following algorithm provided a stochastic transition matrix:

Data: Square Stochastic Matrix $StochMat$; Set of possible input symbols Σ
Result: NFA Markov Chain Simulator
INITIALIZATION;
foreach $MarkovState$ **do**
  Create a corresponding NFA state $State \in States$;
end
Select an arbitrary $State$ to be a start state, activating on start of data;
CONSTRUCTION;
foreach $FromState \in States$ **do**
  foreach $ToState \in StatesFromState$ **do**
    Create transition rule $\delta(FromState, C)$;
    $TransProb \leftarrow StochMat[FromState][ToState]$;
    Without replacement, randomly select $TransProb * |\Sigma|$ symbols from Σ as the character class recognized by $EdgeNode$;
    Add edge from $FromState$ to $EdgeNode$;
  end
end

Algorithm 1: Construct an NFA that simulates a Markov chain given its corresponding transition matrix

The algorithm first converts each Markov chain state into an NFA state. The algorithm then builds a fully connected finite automata where a transition from node $FromState$ to $ToState$ contains a randomly selected (without replacement) proportion of the input alphabet corresponding to the proportion in the stochastic transition matrix.

3.3.2 Markov Chain to Homogeneous NFA Construction Algorithm

As discussed in Chapter 2, homogeneous NFAs are preferred to theoretical NFAs for a variety of reasons. Because Micron's Automata Processor and other automata processing systems focus exclusively on homogeneous automata, we present an algorithm for directly constructing homogeneous automata from Markov chain stochastic matrices.

A homogeneous NFA simulating a Markov chain can be constructed using the following algorithm provided a stochastic transition matrix:

Data: Square Stochastic Matrix $StochMat$; Set of possible input symbols Σ
Result: Homogeneous Markov Chain Simulator
INITIALIZATION;
foreach $FromState$ **do**
  Create a reporting STE "state node" representing $FromState$ that recognizes all input symbols;
end
Select an arbitrary $FromState$ to be start state, activating on start of data;
CONSTRUCTION;
foreach $FromState$ **do**
  foreach $ToState$ **do**
    Create "edge node" STE $EdgeNode$;
    $TransProb \leftarrow StochMat[FromState][ToState]$;
    Without replacement, randomly select $TransProb * |\Sigma|$ symbols from Σ as the character class recognized by $EdgeNode$;
    Add edge from $FromState$ to $EdgeNode$;
    Add edge from $EdgeNode$ to $ToState$;
  end
end

Algorithm 2: Construct AP Markov Chain Simulation

An example of this construction for the unfair coin example is shown in Figure 3.2. For illustrative purposes, we restrict the input symbols to be within the character class $[0-9]$. The proportion of symbols in each EdgeNode corresponds to the transition probability recorded in the transition matrix.

Note that unlike the non-homogeneous construction in Section 3.2.2, this construction takes two cycles to generate an output, one to move from a "from state" node to an edge node, and another to move from an edge node to a "to state" node. Algorithm 2 can easily be modified to generate an output on every cycle by also setting a randomly selected edge node, along with an arbitrary state node, to act as a start state. We omit this construction for clarity but assume one random output per cycle when modeling performance of Markov chains on AP hardware.

While other Markov chain constructions exist, we identified this as the most efficient in terms of overall STE usage, connectivity, and performance. We leave research and evaluation of other constructions as future work. Markov chains can also be used to construct probabilistic automata that model more complex systems.

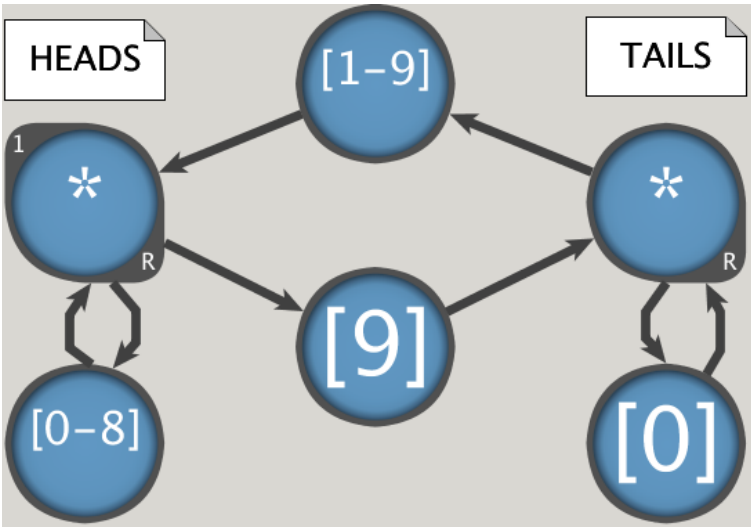

Figure 3.2: A Markov chain implemented on the AP corresponding to the theoretical Markov chain in Figure 3.1, with two reporting "state nodes" representing *Heads* and *Tails*. Reporting states are indicated by an "R" subscript. The start state is indicated by a "1" superscript. Transition probabilities between these states are *unfair* and are modeled by dividing the possible input symbols $[0-9]$ into random groups, proportional to the transition probabilities.

We discuss some examples (such as agent agent-based simulations) in Section 3.7 but leave evaluation of the quality of these more complex systems to future work.

3.3.3 Correlation Among Parallel NFA-based Markov Chains

Because we construct pseudo-random output using NFA-based Markov chains with randomly generated, fixed transition functions, *any* number of parallel NFA-based Markov chains that consume the same input *must* produce output that is correlated.

For example, consider a 2-state chain with identical, but fair, transition tables. In this case, the NFA-based Markov chains can only ever make the same transitions on any random input, and will emit output of the form $((00)|(11))^*$, obviously a non-random sequence! If the second transition table is the inverse of the first, the NFAs will emit output of the form $((01)|(10))^*$, an output just as patterend as was produced by identical tables.

While it is fairly easy to construct transition tables that produce non-random output, as we increase the number of parallel Markov chains, size of the input alphabet, and the number of states in the Markov chain, the properties of multiple transition tables that produce high-quality random output become much harder to identify.

When constructing new NFA-based Markov chains, we choose to randomly construct its transition table. While this strategy does not preclude the construction of highly-correlated groups of chains which produce low-quality pseudo-random output, we hypothesize that random construction performs well enough to pass

all statistical tests for a certain number of Markov chain time steps. We leave potential improvements on this strategy to future work.

3.4 Generating Efficient and High-Quality Pseudo-Random Behavior on Micron's Automata Processor

By constructing a certain number of NFA-based Markov chains that model rolls of fair dice, and simulating all automata using the same pseudo-random input, we can create a large amount of pseudo-random behavior, with a small input stimulus.

However, because there may be many hundreds to thousands of transitions per symbol in parallel NFAs, CPU-based simulation of these systems will almost certainly not be able to compete with state-of-the-art PRNG techniques like Philox. Therefore, we propose to use Micron's Automata Processor to accelerate NFA-based PRNG. We dub this methodology *AP PRNG*. This section describes AP PRNG, and the potential PRNG throughput given reasonable architectural restrictions.

3.4.1 AP PRNG System Design

System Architecture

The proposed system architecture is shown in figure 3.3.

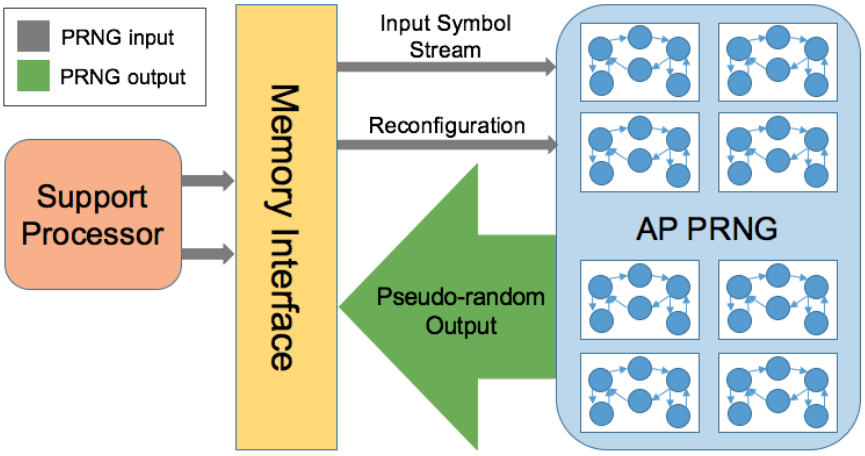

Figure 3.3: AP PRNG system-level diagram. A host processor provides a small amount of pseudo-random input to drive transitions and reconfiguration.

A certain number of parallel, NFA-based Markov chains are placed and routed on Micron's AP. AP PRNG can be configured to support a number of NFA-based Markov Chains (N_{MC}) that drive random output. Each

individual NFA-based Markov chain can be configured with a certain number of final states (N_S), which is also the number of states in the simulated Markov chain.

All chains share the same pseudo-random input symbol stream and make transitions according to the randomized transitions outlined in Algorithm 2. For every input symbol, each chain will enter into a state that reports its encoded binary ID. These IDs are then combined by the AP architecture, and exported off-chip as pseudo-random output.

Because AP PRNG output is correlated, eventually, statistical tests will be able to identify patterns in the output. Thus, the randomized transition rules in each NFA-based Markov chain need to be reconfigured periodically after a certain number of input symbols to preserve quality. The lower the input reconfiguration threshold (I_R), the more frequently the support processor will need to reconfigure AP PRNG transition tables, and the higher the performance penalty.

Note that a support processor is required to provide the random or pseudo-random input symbol stream. The support processor is also responsible for randomly configuring the symbol sets of the STEs responsible for pseudo-random transitions in the NFA-based Markov chains.

An AP PRNG configuration depends on the type of NFA-based Markov chains generated (N_S), the number of NFAs of that type (N_{MC}), and the input reconfiguration threshold (I_R). We discuss the effects of various NFA-based Markov chain configurations on capacity, AP PRNG output quality, and AP PRNG throughput in the following sections.

AP Chip Capacity

Each NFA-based Markov chain generated by Algorithm 2 requires a certain number of reporting states, and a certain number of total NFA states to be implemented. Because only 32 STEs out of 256 in an AP block are capable of reporting in the first generation AP architecture [34], each parallel NFA-based Markov chain may be limited by either reporting elements or total STEs per block depending on how many Markov chain states it models. An N-state fair Markov chain requires N reporting STEs to model as an NFA, thus we can instantiate a maximum of 16, 8, and 4 chains per AP block for 2, 4, and 8-state chains respectively. A N-state fair Markov chain requires $N^2 + N$ total STEs to model as an NFA, thus, based on STE capacity, we can instantiate a maximum of 42, 12, and 3 chains per block for 2, 4, and 8-state chains respectively. Thus, considering both constraints, we can implement a maximum of 16, 8, and 3 chains per block for 2, 4, and 8-state chains respectively.

Potential AP PRNG Performance

Micron's AP is capable of processing one symbol cycle every 7.5ns, meaning that AP PRNG can process a Markov-chain time-step every 7.5ns. Given the chip capacity constraints discussed above, and assuming that each reporting element can be used to create random output, and no chip output reporting bottlenecks, a single AP chip can create up to 51.2GB/s of pseudo-random output. Given that an AP board contains 32 AP chips, the maximum throughput per board is upwards of 1.6TB/s of pseudo-random throughput, greater than 10 times the fastest GPU-based, board-level PRNG throughput reported in the literature [62].

Table 3.2 summarizes the constraints derived in the previous section and reports the maximum potential random output for 2, 4, and 8-state chains on the AP given our assumptions.

N_S	Reporting STEs	Total STEs	N_{MC} per AP Block	Output Bits per Cycle per Chip	Input Random Throughput Required	Max Throughput per Chip
2	2	6	16	3,072	133MB/s	51,200MB/s
4	4	20	8	3,072	133MB/s	51,200MB/s
8	8	72	3	1,728	133MB/s	28,800MB/s

Table 3.2: Hardware resource requirements and best case performance metrics for 2, 4, and 8-state Markov chains as constructed by Algorithm 2 on a single AP chip.

Note that we also report the required input pseudo-random or random throughput necessary to drive all transitions on chip. This is important to track to make sure it is reasonable that a given support processor will not bottleneck AP PRNG. Because all automata on an AP chip share the same input symbol stream, and the AP processes one input symbol every 7.5ns, the maximum required input throughput necessary is 133MB/s.

While the potential PRNG throughput of AP PRNG is impressive, correlation among Markov chains and practical output throughput bottlenecks of AP chips will prevent AP PRNG from reaching this theoretical upper-bound. In the next sections, we simulate the APPRNG system to analyze the effects of AP PRNG configurations on PRNG quality. We then analyze the AP chip's output throughput bottlenecks and propose solutions to improve performance.

3.5 Effects of AP PRNG Configurations on AP PRNG Quality

The known existence of correlation among parallel NFA-based Markov chains introduces two important questions: (1) how does the number (N_{MC}) and size (N_S) of parallel Markov chains affect quality of PRNG, and (2) given the configuration that provides the best quality PRNG, how long can we run AP PRNG before statistical tests are able to reliably detect patterned output, and the system must be reconfigured (I_R)?

The following sections simulate the AP PRNG system and use the TestU01 statistical test battery [57] to evaluate the quality of the resulting pseudo-random output. The results of these sensitivity analyses will motivate configurations for the final AP PRNG system.

3.5.1 Experimental Framework

To test AP PRNG quality, we designed a program to build NFA-based Markov chains using Algorithm 2 and simulate them. We gather the PRNG output from the simulator and measure its quality using the test batteries in the TestU01 statistical test suite [57]. Because first-generation AP hardware is not yet available, we use a functional simulation of AP PRNG behavior, which was verified using Micron's AP SDK.

Statistical Random Quality Tests

TestU01 is made of three main test batteries: SmallCrush, Crush, and BigCrush. SmallCrush consists of 10 statistical tests, and is meant to quickly check obvious statistical flaws in PRNG output. Crush applies 96 distinct statistical tests (144 total test statistics), while BigCrush applies 106 distinct tests (160 total test statistics). Because production AP hardware is not currently available, the BigCrush test battery can take between 3-7 days to complete on a single CPU core, depending on the configuration. We therefore do initial sensitivity analyses using SmallCrush, and Crush, in order to quickly identify trends in relative quality of random output between different AP PRNG configurations.

Support Processor Model

Because AP PRNG requires a support processor to supply random input to the hypothetical AP, and to randomly configure Markov chain transition tables, we must provide a source of pseudo-random numbers that simulates the role of an AP PRNG support processor. For this evaluation we use the $Philox32x4\_10$ generator [63] to provide all random input. As one should expect for any PRNG, we found that using lower quality input sources, such as the C standard library's *rand*() or Mersenne Twister, translated to significantly lower quality AP PRNG output. We therefore use the Philox algorithm, as it is the most performant and highest quality generator available, and available as an open source C++ library [63]. We use this library implementation to drive all random transition table configuration and streaming input to AP PRNG.

3.5.2 Effect of Markov Chain Size on PRNG Quality

To show how the quality of pseudo-random output is affected by Markov chains of different sizes, we run multiple trials of SmallCrush using the AP PRNG functional simulator. Although SmallCrush may not be

the most stringent test suite, and thus less useful when comparing to state-of-the-art PRNGs, it is suitable to quickly identify relative patterns in failures among different AP PRNG configurations, and motivate appropriate parameters for more stringent tests.

We configured AP PRNG with a reconfiguration threshold (I_R) of 50,000 input symbols, 384 parallel Markov chains (N_{MC}), varying the size of the Markov chains as 2, 4, or 8 states (N_S). For each state configuration, we ran 16 trials of SmallCrush and collected data on all test failures. The results are shown in Table 3.3.

Number of Markov Chain States	**2**	**4**	**8**
Average Number of Failures	5.5	2	0
Distinct Number of Failures	6	2	0

Table 3.3: The effects of increasing the number of states per Markov chain on random quality. It is statistically harder to identify correlation between chains with more states.

We can see that random quality is highly sensitive to the number of states used to build each Markov chain. 2-state Markov chains fail an average of 5.5 tests over the 16 trials, failing 6 unique tests in the test suite. 4-state Markov chains consistently fail the same two tests over the 16 trials. 8-state Markov chains are completely resistant to failure over the 16 trials, passing all tests. This trend is likely due to the huge increase in the possible outputs that could be created by Markov chains with more states. A larger number of states allows for a larger number of possible configurations of a single Markov chain. Therefore, Markov chains with 8 states will take longer to repeat configurations and exhibit statistically identifiable non-random behavior than the same number of chains with 2 states.

We therefore only consider 8-state chains in the following sensitivity analyses.

3.5.3 Effect of Parallel Markov Chains on Random Quality

To show how the number of parallel, 8-state Markov chains affects the random quality produced, we explore performance of AP PRNG with from 32 to 768 Markov chains operating in parallel. For these experiments, we use Crush to evaluate random quality. Figure 3.4 summarizes the experimental results. Apart from two obvious outliers, when using 352 and 704 parallel Markov chains, as the number of parallel Markov chains increases, quality of randomness is stable, suggesting that practically, larger numbers of parallel Markov chains do not significantly decrease quality of random output.

We hypothesized that the marked increases in the failure rate for 352 (32×11) and 704 (64×11) parallel Markov chains was due to an undesirable property of our algorithm for converting output bits from AP PRNG to 32-bit integers. Because the value of an integer is influenced more by high-significance bits, non-random

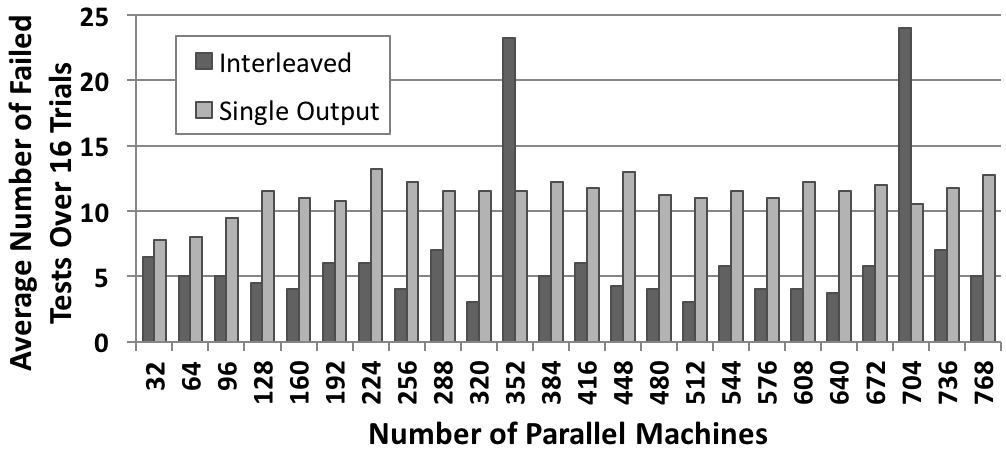

Figure 3.4: Average number of Crush failures over four trials for parallel 8-state Markov chains with a reconfiguration threshold of 200,000. The darker bars represent failure rates when interleaving output bits. Spikes in failure rates occur when the same Markov chains always contribute to the same bits in output integers. The lighter bars represent failure rates when successive output from a Markov chain contributes to a single output integer. This eliminates the spike in failures, but reduces overall quality.

behavior in high-significance bits will be noticed quicker than non-random behavior in low-significance bits on the numeric-based tests.

Because we construct 32-bit integers by concatenating three output bits from every 8-state chain in a round-robin fashion, certain numbers of chains will align their output, such that a single chain will always contribute to the same digits. This puts undue reliance on chains that contribute highly significant bits of integers, and may amplify patterns in the psuedo-random output. To test this hypothesis, we modified how integers are constructed to have a Markov chain provide all the bits for a single 32-bit value before the value is consumed by the statistical tests. Thus, all Markov chains contribute uniformly to the digits in a single integer, rather than a particular set of digits. The results of this experiment are shown in Figure 3.4.

The spikes in failure rates are eliminated, although failure rates are higher on average, supporting our hypothesis. We therefore adjust AP PRNG to use the largest prime number of Markov chains able to fit onto an AP chip. A prime number of machines will prevent any one machine from contributing to the same location in an supplied 32-bit integer at least until we provide 32 times that prime number of input symbols. The maximum number of 8-state Markov chains that will fit onto an AP chip is 576; we therefore use 571 (the largest prime less than 576) 8-state chains per AP chip as the highest performing configuration for final quality and performance evaluations.

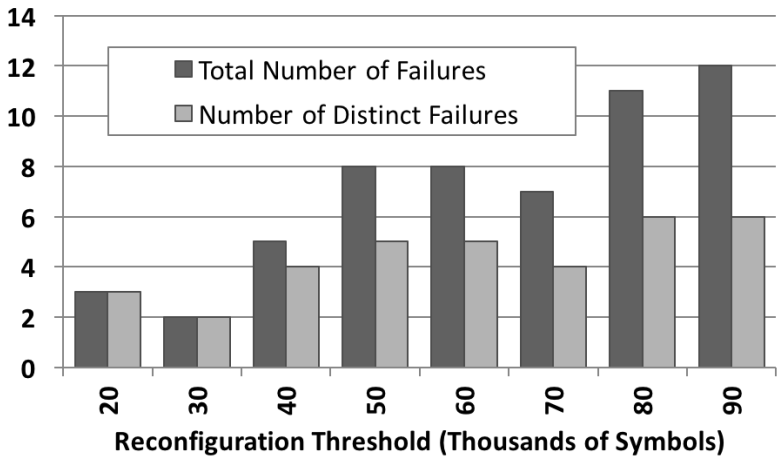

Figure 3.5: As the reconfiguration threshold increases, it is becomes easier for statistical tests to identify non-random behavior.

3.5.4 Effect of Input Size on Random Quality

To show how the quality of random output is affected by the number of input symbols, we ran four trials of 761 8-state Markov chains through the Crush test suite, varying the number of input symbols from 20,000 to 90,000. We hypothesized that the more symbols we allowed each parallel machine to consume without reconfiguration, the more likely the output is to look non-random. Therefore, *shorter* reconfiguration thresholds will likely increase quality of random output. However, shorter reconfiguration thresholds force the AP to reconfigure more often, incurring a reconfiguration penalty.

The results of the experiment are shown in Figure 3.5.

Figure 3.5 shows that, as we increase the number of input symbols between reconfigurations, the quality of random output decreases. However, even for 20,000 symbols, 761, 8-state Markov chains do not consistently pass all Crush tests. In initial exploratory tests, some BigCrush tests even failed with reconfiguration thresholds as low as 10,000, meaning that even shorter reconfiguration thresholds are required in order to match the quality of the Philox algorithm. As the AP only takes $7.5ns$ to consume a symbol, but approximately $45ms$ to reconfigure [15], a reconfiguration threshold of 10,000 would cause the AP to spend 99.83% of its time reconfiguring, translating to $48MB/s$ of output, less random output than input.

We hypothesize that performance of AP PRNG could be drastically increased if we could reduce the impact of *neighbor dependence*. Because our algorithm always forms integers by interleaving bits of neighboring Markov chains in a deterministic manner, we put undue pressure on close groups of machines to produce uncorrelated output. Previously, we saw that neighbor dependence induced catastrophic failure rates when the same configuration of machines was always used to contribute the same digits of output integers. By using a prime number of machines, we could force this configuration to at least rotate. However, rotation

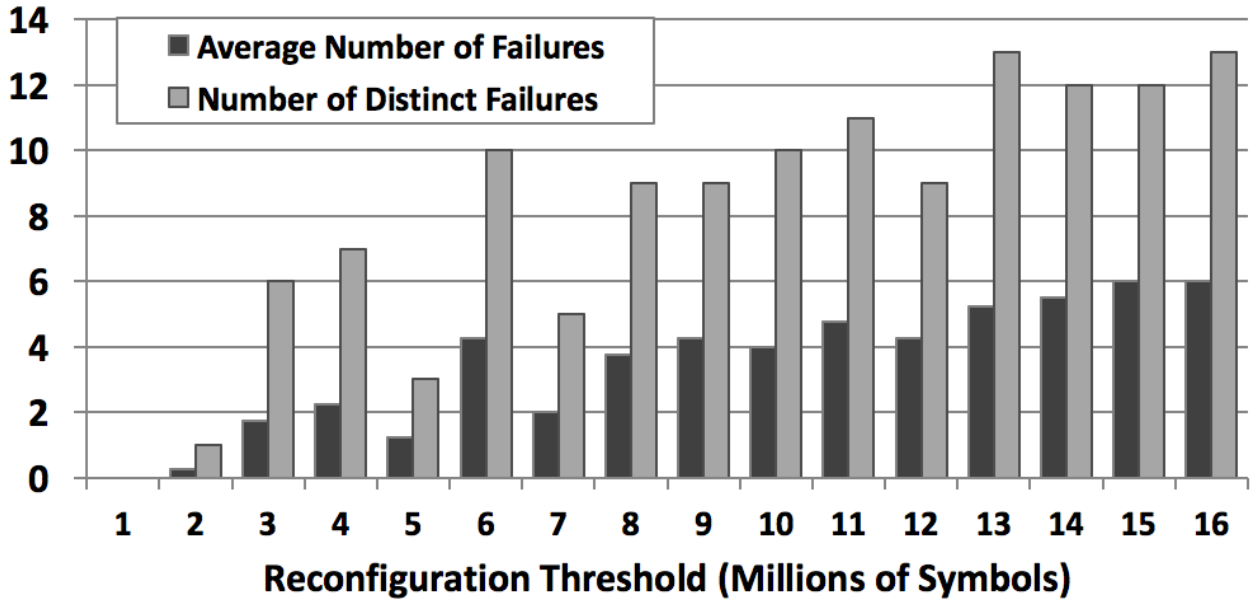

Figure 3.6: Output quality of AP PRNG with output permutation hardware greatly increases quality of random output. AP PRNG passes all tests in BigCrush with a reconfiguration threshold of at least $1,000,000$

still preserves the relative order of machines contributing to any individual output integer. To eliminate neighbor dependence without decreasing throughput, we investigate potential impact of output permutation via pipelined support hardware or software to reorder output bits of a group of Markov chains before we combine the bits into integer values for analysis.

We configured AP PRNG to use 571 8-state Markov chains with 32-wide permutation hardware that changes the output ordering of every 32 Markov chains every $1,000$ symbol cycles. We then ran four trials of BigCrush to assess improvements in quality of random output. Results are shown in Figure 3.6. Compared to the results from Figure 3.5, random output quality greatly improved. Previously, AP PRNG failed all tests in Crush with a reconfiguration threshold of $20,000$ input symbols. However, AP PRNG with permutation capability passes all BigCrush tests with a reconfiguration threshold of at least $1,000,000$ symbols. This indicates that neighbor dependence was a major contributor to poor output quality. AP PRNG with permutation capability can generate at least $50\times$ more random output before full reconfiguration is required.

While this functionality is simulated in software, we hypothesize its implementation as an addition to the reporting architecture, support ASIC, or via the host or support processor. The most efficient implementation will depend on the particular deployment scenario of the AP. Output permutation is only one way of increasing quality of output, and this result motivates further studies to identify other methods (especially to replace the simple round-robin scheme) to cheaply mitigate the effects of neighbor dependence in hardware, or software.

3.6 AP PRNG Performance Model

The above sensitivity analyses motivate using 571, 8-state Markov chains, with a reconfiguration threshold of at least $1,000,000$ input symbols, and a permutation threshold of $1,000$ as a high-quality PRNG. To analyze the practical performance of this AP PRNG configuration, we constructed a performance model based on these parameters and the reported configuration of the first-generation AP architecture [34]. Because the AP operates at $133MHz$, consuming one 8-bit symbol and executing all transition rules every $7.5ns$ cycle, an AP performance model does not require simulation. Below we describe the model inputs, output metrics, and sensitivity to certain architectural parameters.

First Generation AP Architectural Parameters	
Frequency	133MHz
Cycle Time (T_c)	7.5ns
STE Size	256 bits
Random State per Chip ($ChipState$)	1.17MB
Est. AP Reconfiguration Time (T_r)	45ms

AP PRNG Parameters	
States per Markov chain (N_s)	8
Markov chains per AP Chip (N_{mc})	571
Input Reconfiguration Threshold (I_R)	$1,000,000$
Permutation Width (P_W)	32
Permutation Reconfiguration Threshold (P_R)	$1,000$

AP PRNG Performance Model	
Chip Output per Input Symbol (O)	$\log_2(N_s) * N_{mc}$
Random Generation Time (T_R)	$I_R * T_c$
Runs per second ($Runs$)	$1/(T_{Run} + T_r)$
AP PRNG Throughput (P)	$Runs * O$
Random Input Stream Rate (In_s)	$Runs * I_R$
Random Input Required for Reconfiguration (In_r)	$Runs * ChipState$
Random Input Required per Permutation (In_p)	$P_W log_2(P_W)$

3.6.1 Performance Sensitivity to Reconfiguration Threshold

Figure 3.7 shows the throughput of AP PRNG predicted by the performance model for various reconfiguration thresholds (I_R). When $I_R = 1,000,000$, AP PRNG produces $4.1GB/s$ of random output per proposed first-gen AP chip.

One unique feature of AP PRNG is that it allows the user to easily trade random quality for higher random throughput. This is desirable if an application or simulation does not require extremely high-quality randomness and is constrained by power, or performance. If strict random quality is not required, and the

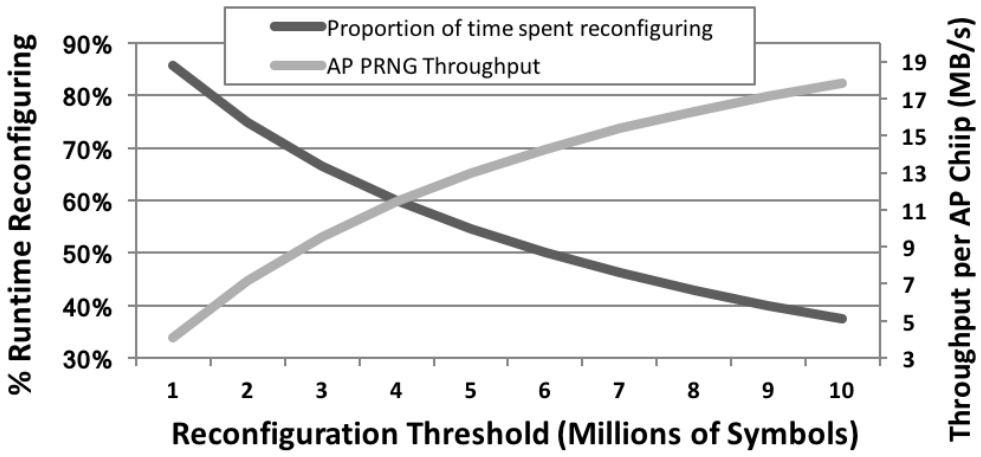

Figure 3.7: Percentage of runtime spent reconfiguring vs. AP PRNG throughput with different reconfiguration thresholds. Performance increases dramatically if AP PRNG is able to reconfigure less frequently.

user allows for a longer I_R, the model predicts that performance can be increased dramatically. Figure 3.7 shows that for an $I_R = 10,000,000$, a single first-gen AP chip is capable of producing $17.8GB/s$ of random output. End users will therefore be able to reduce the number of AP chips, and power consumption, or increase performance, if statistically perfect randomness is not required for a particular application.

AP PRNG also requires a source of random input to drive automata transitions, reconfiguration, and permutation. While we assume the system host processor is able to implement the $Philox32 \times 4\_10$ algorithm and provide this random input, the resulting need for random input throughput can be significant. For $I_R = 1,000,000$, the model predicts we need $200.7MB/s$ of random input per chip. For $I_R = 10,000,000$, the model predicts we need $176.3MB/s$ of random input per chip.

3.6.2 Performance on Future AP Hardware

On the first generation AP chip hardware, maximum output throughput is projected to be $436.9MB/s$ [1], thus limiting the practical amount of random output that can be exported off chip. STE reconfiguration time, which we have shown to be the next most significant performance bottleneck, is projected to be $45ms$ on first-generation AP hardware. While these parameters represent the projected performance of the first-generation AP chip architecture and implementation, they do not reflect fundamental bottlenecks to AP PRNG. For example, output report vectors and STE columns are implemented as DRAM memory; therefore, it is not unreasonable to assume that both STE reconfiguration and output reporting could happen at or near native memory I/O speeds, drastically decreasing reconfiguration times and increasing practical AP PRNG output throughput.

We consider the performance of AP PRNG where writes could occur at native DDR3, DDR4 and Hybrid

Memory Technology	**DDR3**	**DDR4**	**HMC 2.0**
Peak Throughput (GB/s)	12.8	17.0	320
T_r (μs)	91.4	68.8	7.3
AP Chip Output (GB/s)	28.2	28.3	28.5
Throughput Limited Output (GB/s)	12.8	17.0	28.5

Table 3.4: AP PRNG performance modeled on different memory technologies. AP PRNG throughput is limited by peak memory throughput for DDR3 and DDR4 technologies.

Memory Cube (HMC) throughputs. HMC technology accomplishes massive I/O throughput by stacking DRAM layers directly on top of logic, and inserting vertical communication links with through-silicon vias [64]. One could integrate the AP into HMC as one or many layers of a stacked HMC design as a part of a heterogeneous memory module. Table 3.4 shows the performance of DDR3, DDR4, and HMC2.0 technologies with derived reconfiguration times (T_R), and AP PRNG throughput.

For 571, 8-state Markov chains we only need to reconfigure all transition STEs (64 for 8-state chains). For a single chip, this translates to $1.17MB$ of state. Table 3.4 shows that even if we reconfigure the AP using native DDR3 bandwidth, the AP PRNG performance model predicts each AP chip can produce $28.2GB/s$ of high-quality pseudo-random output. However, peak DDR3 throughput is only $12.8GB/s$, and thus limits the practical amount of output we can export off chip.

If we consider future high-throughput memory technologies, such as Hybrid Memory Cube (HMC) [64], the AP PRNG performance model predicts that each AP chip will produce a comparable $28.33GB/s$ of pseudo-random output. However, HMC's much larger output bandwidth allows us to easily export this output off chip.

As we increase the number of reconfigurations per second, AP PRNG also requires more random input throughput. The HMC configuration, with $T_R = 1,000,000$, requires $289MB/s$ of random input per chip, about 7× larger than AP PRNG on the first-generation AP chip. While this is not an insignificant amount of random input, the host processor can easily supply it. The significant amount of state needed for STE reconfiguration motivates research into techniques to increase T_R (such as output permutation) without sacrificing random quality.

Future AP architectures implemented on cutting-edge transistor process nodes will also most likely have larger STE capacities. Adjusting the model for a reasonable 1.41× increase in STE capacity (corresponding to 1.41× more Markov chains) per AP core, AP PRNG can produce $40.5GB/s$ of random output per chip, while only requiring $355MB/s$ of random input from the host processor to configure transition tables, drive automata transitions, and drive permutation reconfiguration.

3.6.3 Estimating AP PRNG Power Advantage

Ultimately, performance and power advantages over current PRNG implementations will greatly depend on the implementation and deployment scenario of AP PRNG or other applications of Markov chains such as discrete event simulations, however, we project that AP PRNG will be much more power efficient than GPU PRNG and Markov-chain based discrete-event simulations. For example, the GTX 580 GPU used in Salmon et.al [63] has a TDP of 244W, while each DDR3-based AP chip has a projected TDP of 4W, and stacked HMC-based memories are projected to use 70% less energy than DDR3. Figure 3.8 shows the PRNG efficiency of six different realistic AP PRNG deployment scenarios.

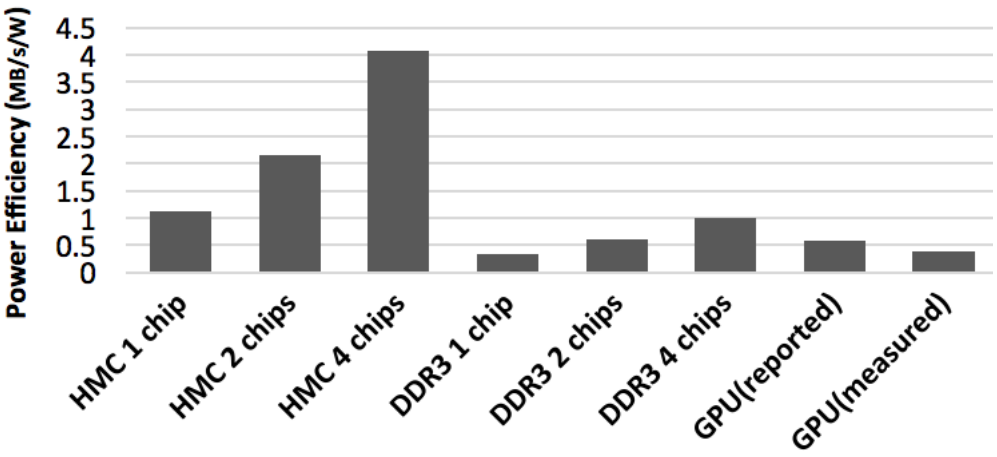

Figure 3.8: AP PRNG is up to 6.8× more power efficient than the highest-throughput reported GPU PRNG depending on the deployment scenario.

All AP deployment scenarios require a support processor to generate random input and configure the AP. We assume that a typical single CPU support processor core consumes 35W. The configuration with 4 AP chips implemented in an HMC technology produces 4MB/s/W, 6.8× more power efficient than the best performing GPU PRNG reported in the literature [63], and 10.8× more power efficient than our measured experiments using the curand library implementation of $Philox32x4\_10$ on an NVidia K20C GPU. Disregarding support processor power consumption, and conservatively assuming a 4W TDP per AP chip, AP chips are 16.8× more power efficient than the reported GPU implementation.

3.7 Other Uses for Pseudo-Random Behavior

Any agent-based model can be represented using Markov chains [65]. The following sections show different examples of how accelerated random Markov chain simulation using automata could be used to model useful applications that rely on agent-based simulation.

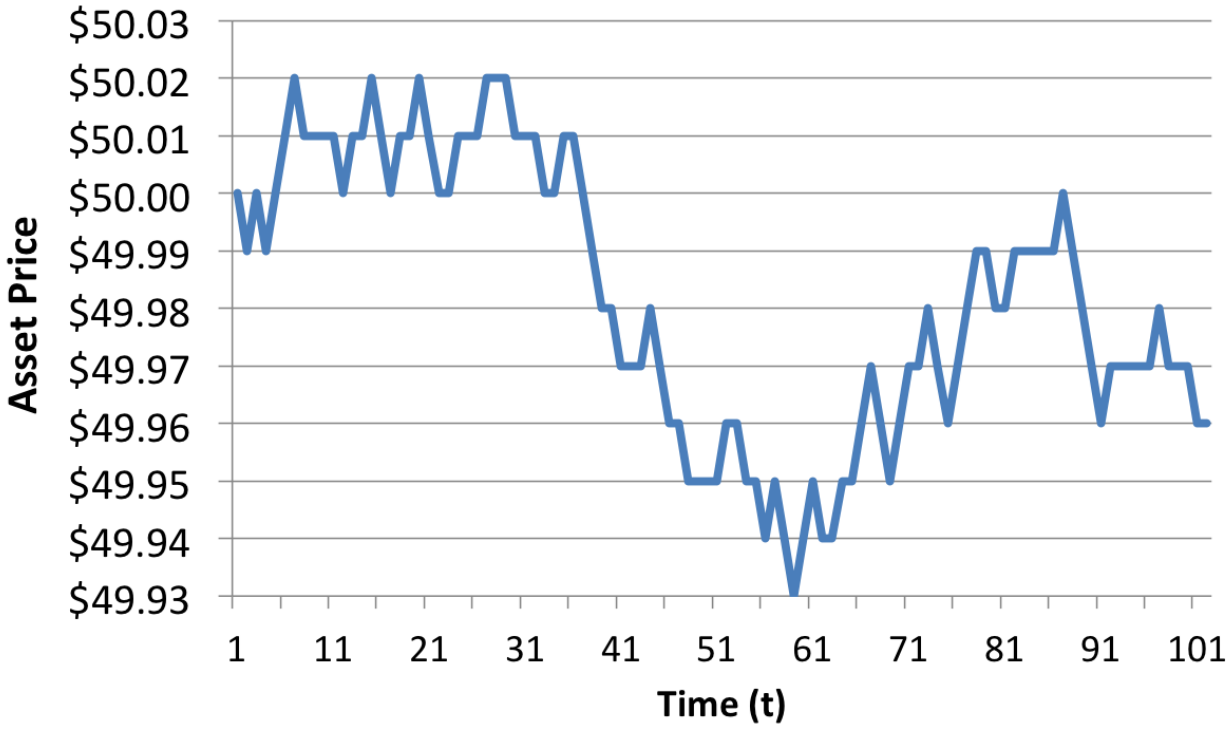

Figure 3.9: Asset price simulation modeling a random walk with transitions of +/-$0.01 or no change.

3.7.1 Simulating Asset Price Motion

Asset price motion is concerned with simulating and predicting the price of a financial asset over some set amount of time. Based on randomly generated input, an asset price is adjusted over time according to a motion function. This function is often chosen to be *Brownian motion*. Brownian motion was originally developed as a part of particle theory and defined to model the random motion of particles in a system, but has been re-purposed as a proposed model for behavior of the price of assets and financial markets.

Because Brownian motion is a stochastic process, it can be simulated using *random walks* on Markov chains. Random walks are driven by random transitions (defined by a motion function) between states of a *walker* over some dimensional space of states. In our asset price example, the state space is simply all discrete prices that the asset could occupy, thus a $1D$ number line. At any given time, the price could go up or down depending on a random input, and the added or subtracted amount would be determined by the motion function. At the end of the simulation, the final state of the walker is the predicted asset price of the simulation.

An example of a $1D$ random walk simulation of an asset price is illustrated in Figure 3.9. $1D$ random walks can be simulated using linear Markov chains, or Markov chains with diagonal or banded stochastic matrices. As long as the transitions to neighboring (or nearly neighboring) states correctly encode the

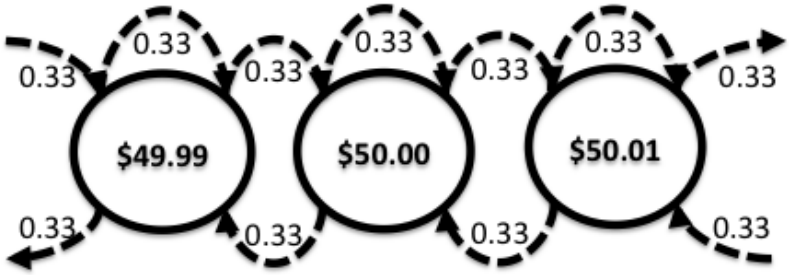

Figure 3.10: Linear Markov chain modeling a random walk with transitions of +/-$0.01 or no change.

proposed transition function. An example of the linear Markov chain that produced the prior simulation is shown in Figure 3.10. Note that by definition, our linear Markov chain is only capable of modeling discrete, non-continuous values. However, as long as we implement enough states, we can model to arbitrary precision with arbitrary price bounds. Our random walk construction described in the next section shows how we can implement this arbitrary precision on the AP without this 1-1 correspondence between states and discrete values.

For our explanatory example, we will simply set the cost function to be +/-$0.01 with equal probability, and then explain how to construct simulated arithmetic Brownian motion cost functions.

3.7.2 Mapping an Asset Price Simulation to the AP Hardware

We now show how Markov chains on the AP can be used to simulate asset price motion via random walks on a linear Markov chain. We first present the construction of a +/-1 linear Markov chain "walker" simulation, and then present a method to avoid the 1-1 correspondence between states and discrete values in the simulation.

Linear Markov Chain Random Walker:

To construct a linear Markov chain "walker," we first need to define the appropriate diagonal stochastic transition matrix according to our transition function. Because we define our current transition function to increment or decrement by $0.01, or remain at the current price with equal probability, we know that the transitions from each state to its neighbor will have a probability of 0.33. Thus the transition matrix looks like the following:

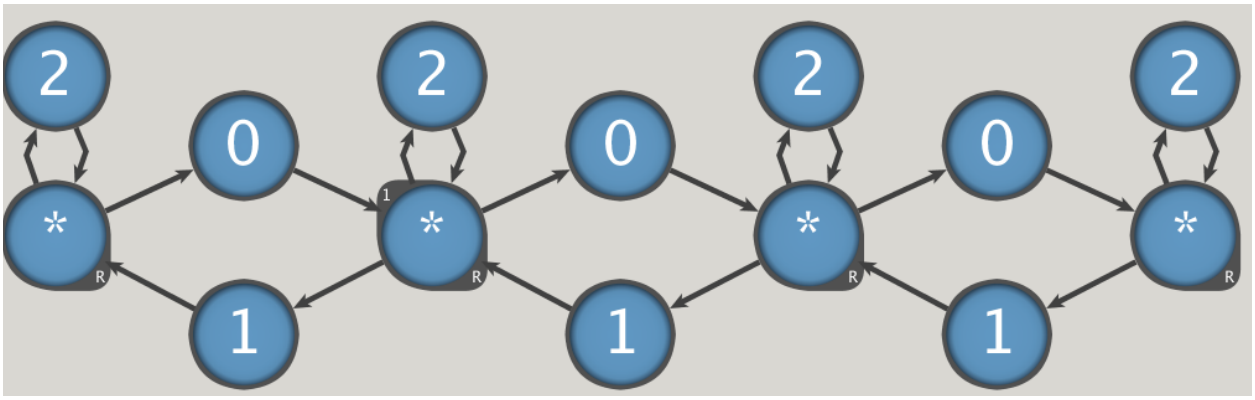

Figure 3.11: How a linear Markov chain can be implemented on the AP. This chain corresponds to the transition matrix in section 3.7.2.

$$\begin{vmatrix} 0.33 & .33 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ .33 & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & .33 & .33 & 0 & 0 & 0 & 0 & 0 \\ 0 & \cdots & .33 & .33 & .33 & 0 & 0 & 0 & 0 \\ 0 & \cdots & 0 & .33 & .33 & .33 & 0 & 0 & 0 \\ 0 & \cdots & 0 & 0 & .33 & .33 & .33 & 0 & 0 \\ 0 & \cdots & 0 & 0 & 0 & .33 & .33 & .33 & 0 \\ \vdots & \cdots & 0 & 0 & 0 & 0 & .33 & .33 & .33 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & .33 & .33 \end{vmatrix}$$

Each state then represents a certain value of the asset with cent precision, and can transition to its +\$0.01 or -\$0.01 neighbor or remain at the current price with equal probability. Later we will discuss how to modify the transition matrix to simulate other, more complicated transition functions.

Because we only have three transitions, *up*, *down*, or *stay* with equal probability we can simplify the construction of the Markov chain to consider only random input in the range $[0-2]$, where a 1 will force a transition to the neighboring up state, and 0 to the neighboring down state, and a 2 will cause the walker to remain in the current state. An illustration of a section of such a Markov chain walker as it would be implemented on the AP is shown in Figure 3.11.

Bounding Walkers with Counters:

With the above construction, we only have enough STEs per AP core to represent 10,000 discrete prices, or values from \$0.00 to \$100.00. This is not a realistic implementation as prices may be out of this range, and also may have much larger variation, or need more precision. As well, because this construction utilizes

Figure 3.12: A linear Markov chain walker bounded by counters. This construction can represent discrete values with arbitrary precision without a linear increase in states.

almost all STEs on a core, it would preclude parallel simulations or heterogeneous functionality. Therefore, we propose bounding small walkers with counters as a way to reduce the STE costs of such simulations.

The intuition behind counter bounding is that if we can keep track of how many times we "fall off" the edge of a small walker in both directions, we can simulate an infinite sized walker. We accomplish this by using two counters: one to count up falls, and one to count down falls. If the simulation ever falls off either end of the walker, it activates both the corresponding counter, as well as the STE on the other end of the walker, thus "wrapping around" in a ring. At the end of the simulation, the end price will be the current price plus the difference between the counters multiplied by the size of the counter. An example of such a construction is shown in Figure 3.12.

We can also use this technique to bound higher dimensional random walk Markov chains. For every dimension, we need two counters to keep track of whenever we "fall off" the top or bottom of each. The difference in the counters times the size of the walker plus the value in the walker, will always give the coordinate of the random walk in that dimension.

Extracting Values from Counters:

As the value stored within counters is not readily available, we need some way to extract it. We accomplish this be reserving a single symbol for "counter pumping." Consider a counter with target 16 and internal value 13. To know 13, we activate the counter using the pumping symbol until it activates (3 cycles). In

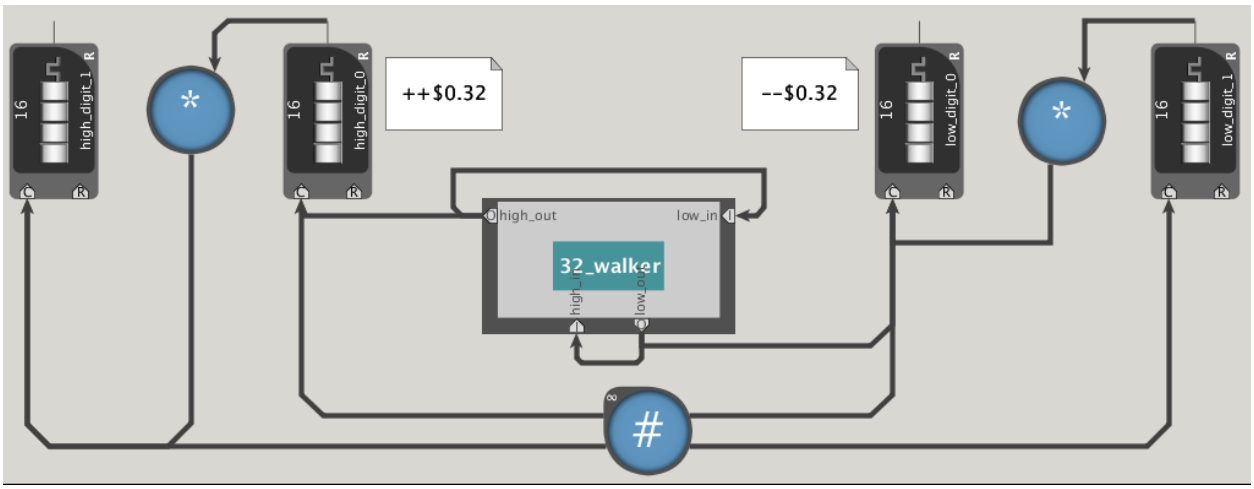

Figure 3.13: A linear Markov chain walker bounded by counters. This construction can represent discrete values with arbitrary precision without a linear increase in states.

post-processing, we now know that the value stored in the counter was the target (16) minus the number of cycles pumped before activation (3), thus 13 is known.

Because counters can have large targets, we may need to pump a large number of times, wasting time and power. Therefore, in practice, two counters may need to be used to keep track of digits of the count, bounding the number of pumps to be the radix of the chosen counting scheme. For example, to construct a two digit counter, have the first digit activate the second digit when it reaches its target and then reset its count to 0. Pumping these counters will therefore efficiently extract each digit of the number.

3.7.3 Final Construction

We construct a prototype asset price simulator based on the example described in the previous section. The full prototype, including walker, bounding counter structures, and pumping system, are shown in Figure 3.13.

We purposefully use four counters in this construction because in that case, we can tune the size of the walker such that the entire simulation fits inside a single AP block.

While simulation of Brownian motion is just one example of the uses of Markov chains, other practical uses for simulations of Markov chains exist. For example, Markov Chain Monte Carlo simulation, which was voted one of the ten most important algorithms of all time, is heavily used in theoretical and applied science and mathematics, and although not feasible for large, continuous state spaces, could be practically implemented on the AP when states are sparsely connected (linear arithmetic Brownian motion), or when the application calls for a relatively small, known transition matrix (credit default modeling).

Chapter 4

VASim: An Open Source Platform for Finite Automata Applications and Architecture Research

4.1 Introduction

Although there is exciting research driving both application and architectural innovations in automata processing, the tools available to researchers for prototyping, debugging, and simulating finite automata are severely fragmented and narrow in scope. This places an unnecessary burden on researchers to track down, learn, and integrate a menagerie of tools and automata formats to evaluate new research against prior state-of-the-art. In particular, we observe the following limitations with existing tools:

- *Closed-source:* many tools designed to manipulate and simulate finite automata are not open source [55]. This means that bug-fixes, performance improvements, or any additional features cannot be added by researchers. Closed source tools often prevent researchers from answering even basic research questions about the structure or behavior of automata.
- *Difficult to obtain:* tools can be hidden on personal or academic websites [66] or behind a gate-keeper [67]. difficult to obtain [67], forcing the researcher to re-implement the algorithm.
- *Restrictively licensed:* even if tools are easily available and open-source, some tools have restrictive open-source licenses that prevent use in for-profit scenarious [66]. This prevents any work in academic settings from being adopted and used by industry.

- *Lacking in features:* many automata tools lack state-of-the-art automata transformation algorithms. If the tools are closed source, these features cannot be added [55]. Sometimes it is possible to write scripts to convert automata to specific serialization formats understood by individual tools [67]. In this way, automata can be passed from tool to tool to used various available features. However, tools are not always designed in a way to make arbitrary automata I/O easy or even feasible at all [43]. A proper research framework should support easy addition of other hypothetical features that may not be useful to a majority of users.

- *Poor performance:* current tools available for simulation can take hours to process reasonably-sized automata and input stimuli [55]. Some tools also consume large amounts of memory, leading to memory thrashing and untenable performance degradation [55].

To address the shortcomings of existing tools, we present the Virtual Automata Simulator, or VASim. VASim is, to the best of our knowledge, the first extensible, general-purpose framework that combines automata simulation, optimization, transformation, and analysis into one, unified and open source code base. This framework enables easy prototyping, debugging, simulation, and analysis of automata-based applications and architectures.

In this chapter, we describe the implementation of VASim, provide clear evidence of its versatility, and demonstrate its usefulness for novel, state-of-the-art automata processing research.

Unlike currently available tools, *VASim is modular, extensible, and general.* The platform can support simulation and analysis of a diverse set of finite automata models, such as classical NFAs, AP-style NFAs, JFAs [68], Counting finite automata [69], and hybrid approaches [46]. To highlight VASim's flexibility, we show how a hypothetical new processing element can be easily added to the execution model. This particular extension was accomplished with the addition or modification of only 9 lines of code.

VASim can serialize automata into several common formats, including the DOT graph language and synthesizable functional Verilog code for FPGA execution. VASim is also capable of exporting automata to output formats for execution on existing CPU [67] and GPU engines [23]. This serialization can allow easy integration and comparison with existing tools with various serialization formats, and allow execution using various automata processing engines on CPUs, GPUs, FPGAs, and custom automata processing hardware.

VASim provides a common codebase for applying state-of-the-art optimizations, transformations, and static and dynamic analyses to finite automata. This platform allows researchers to easily and quickly share new algorithms, and perform fair apples-to-apples comparisons to prior work, accelerating automata-processing research. We provide several optimizations in the core of VASim, including an NFA state reduction algorithm.

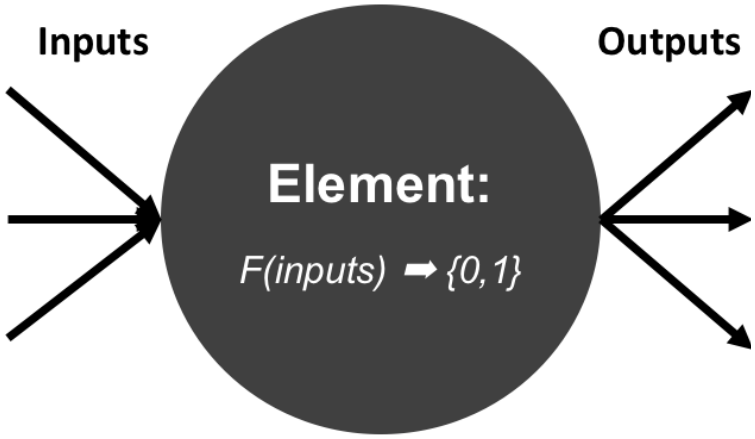

Figure 4.1: A generic automata processing element. All elements compute a boolean function based on input signals. The result of this boolean function is then broadcast to element children.

While performance was not the main driving consideration during development, *VASim is fast*, and is currently 4×-694× faster than existing simulation tools for the Micron's Automata Processor. Furthermore, VASim is parametrically multi-threaded in two dimensions of parallelism common to automata-based applications, allowing for zero-effort improved performance on multi-core systems. VASim's high performance greatly improves the pace of automata processing software and hardware design space exploration.

Together, these contributions form a powerful, comprehensive, full-stack automata processing platform, providing fast and efficient tools for automata application, optimization, and processing research. The next sections describe VASim's software architecture, and simulation methodology.

4.2 VASim Architecture

VASim is based on an abstract, object-oriented view of automata graphs. In VASim, an automata graph is made up of nodes called *Elements*. Elements represent abstract processing elements in an automata network and have an associated boolean activation function as shown in Figure 4.1.

Each Element object contains a list of parent elements, and child elements that make up the edges of the automata graph. Each element can receive a boolean signal from an input or parent node. If an element receives an input signal, or otherwise needs to compute its activation function it is considered *enabled*.

Once enabled, an element computes some sort of activation function. This function can use any combination of its inputs, and/or local state, and/or some global state (e.g. a symbol from an automata input tape) to compute. If an element computes true, it *activates* and can propagate signals to child elements.

Elements themselves do not define any computation, and the Element class merely defines the directed graph structure of the automata (parents and children). Element sub-classes are responsible for defining the

boolean activation function. By default, VASim provides two different types of Element subclasses: the STE and Special Element. The STE is a classic automata processing element, and stores a character set that represents all possible input symbols the STE matches against. An STE examines the global input symbol and defines its boolean activation function as whether or not the current global input symbol is contained in the character set.

Special Elements do not inspect the global input symbol and compute boolean functions only based on the values of their enable signals. An important distinction between Special Elements and any other element in the automata graph is that they compute after STEs within a single symbol cycle. An example of a Special Element currently implemented in VASim is an AND gate. AND gates only activate if either of their inputs are high. Special elements can be chained together allowing for complex combinatorial computation within a symbol cycle. Another example of a special element is a Counter. Counters hold an internal count of the number of cycles an input enable signal was high, and activate when a target threshold is reached. Figure 4.2 shows a high-level overview of VASim's extensible class hierarchy, including STEs, and various special elements.

STEs have a special start property that defines the behavior of the state on the first input symbol, or when an end-of-line symbol sequence is encountered. Finite automata also have final or report states. Because computation can end in either STEs or Special Elements, the Element parent class has a special report property that defines reporting behavior. Reporting elements can also be tagged with an optional list of report codes.

4.2.1 Extending the Virtual Execution Model

VASim's extensible, object oriented design makes adding support for other types of processing elements straightforward and low-effort. We describe how support for an up-down counter—a Special Element counter that can count both up and down—can be easily added to VASim.

The first generation D480 AP architecture only supports counters that can count in one direction. These counters were originally designed to make counting regular expression sub-expression quantifications more efficient [55].

Up-down counters add a third, *count down* port to traditional counters, allowing them to adjust the value in the counter in both directions. A single up-down counter can be treated as a semaphore or 1-bit stack and efficiently compute tasks classical NFAs cannot, e.g. whether or not parenthesis are properly nested in an arbitrarily long input. Both AP-style counters and up-down counters are shown in Figure 4.3.

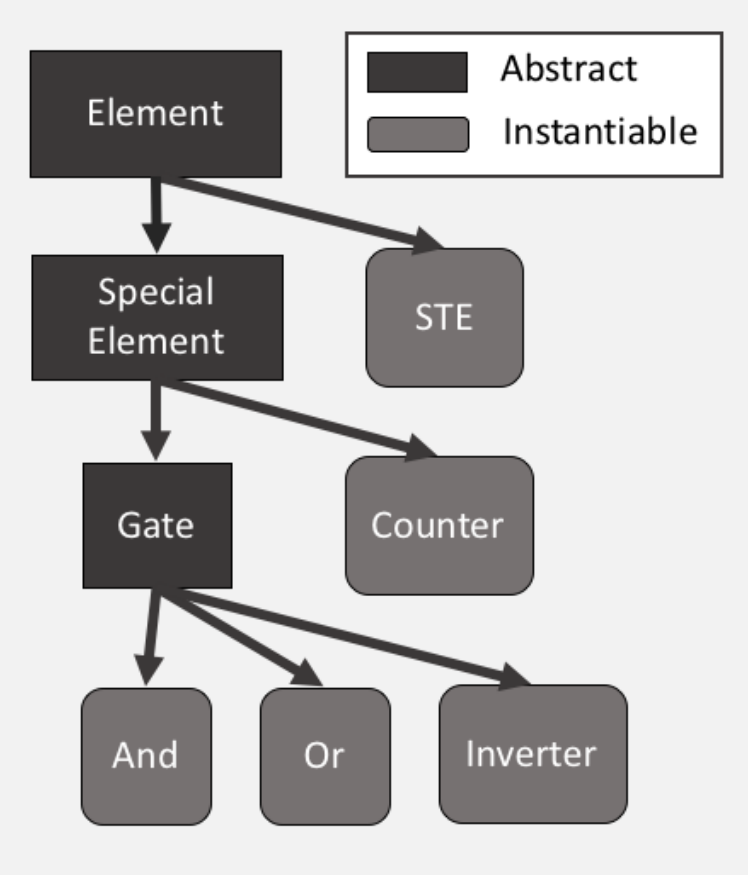

Figure 4.2: VASim class hierarchy. STEs consult the global input symbol before computing a boolean activation function. Special elements do not consult the global input and only compute on input enable signals within a symbol cycle.

Because the up-down counter behaves nearly identically to an AP-style counter, this new class can inherit almost all of the functionality of the counter provided by VASim. The only method that needs to be overridden in the new up-down counter class is the counter's boolean activation function. The original counter's boolean activation function was implemented in 70 lines of code. The up-down counter method required the addition or modification of only 9 lines of code in total.

Because the functionality of counters and up-down counters are so similar, the number of non-boilerplate lines of code is expected to be small. However, the number of lines of code required to implement other arbitrary special elements are generally small as well because they tend to only require the modification of the boolean activation function. For example, we implemented an XOR gate, a gate not included in the AP execution model, by modifying only one line of code.

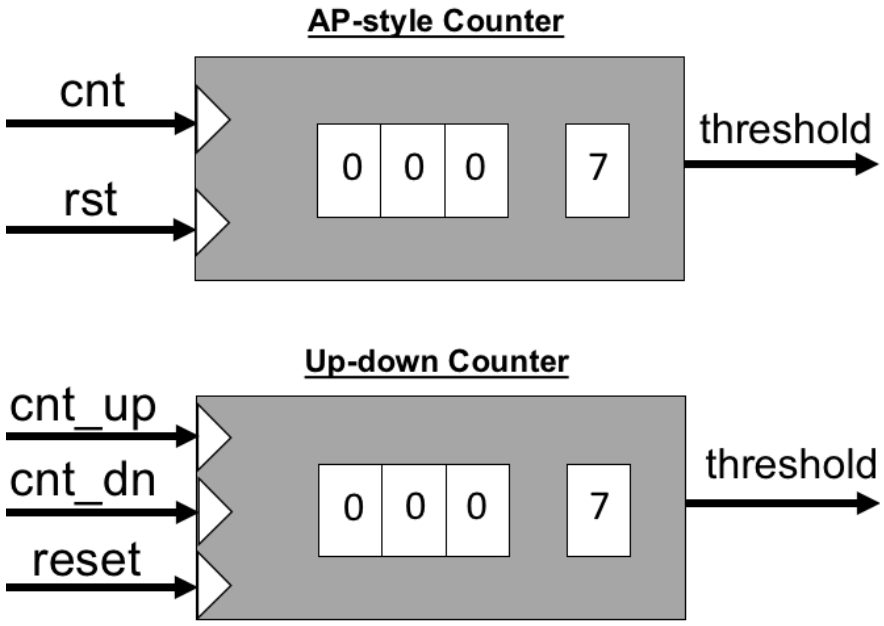

Figure 4.3: AP-style counters (top) are only capable of counting up. Up-down counters (bottom) add a new input port with the ability to count down.

4.3 Automata Simulation

Automata compute by considering an input stream of symbols. For each input symbol, any STE that was enabled on the previous cycle attempts to match the current symbol. If the STE matches the symbol, it activates, and enables all of its children.

After each enabled STE attempts a match, any special elements that are enabled or otherwise need to perform computation compute their activation functions. This stage is analogous to a combinational circuit simulation. Once all Special Elements have computed their transition functions and reached a steady state, the next symbol is considered, and the process starts over again.

VASim divides the automata simulation of each individual symbol into three distinct stages:

Stage 1: each enabled STE attempts to match on the current input symbol. If the STE matches, it is stored to an intermediate data structure. If an STE matches and it is also a reporting STE, then it records a report in a global result data structure.

Stage 2: Each activated STE is considered and each child of all activated STEs are then enabled and pushed to an intermediate data structure.

Stage 3: Because start states are enabled by default, each start state is enabled and pushed to an intermediate data structure.

Stage 4: Special elements are now enabled and can be simulated. Because Special Elements are unbuffered,

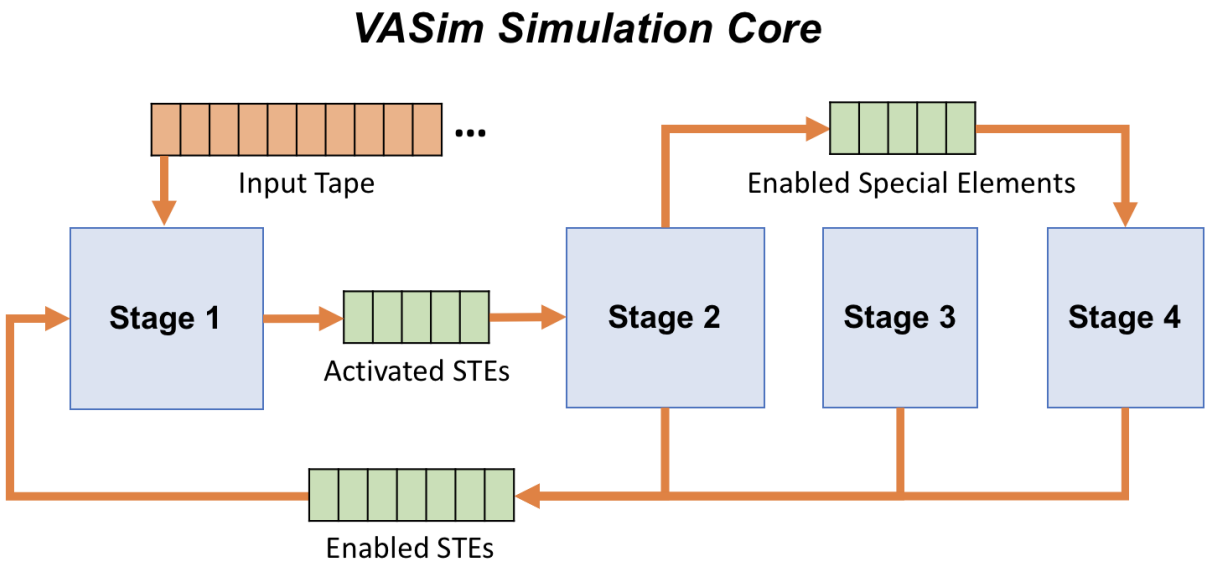

Figure 4.4: VASim simulation pipeline. Stage one computes whether or not each state that was enabled on the previous cycle matches the current input symbol. Matching states activate. Stage two identifies the children of the activated states and enables them. Stage three enables all start states. Stage four computes the boolean functions of all special elements. Stage three must be run to initialize simulation prior to simulating the first simulation cycle.

they require a much more expensive circuit simulation step where steady states in feedback loops must be identified and reached.

Because start states must be enabled before the first iteration of Stage 1, simulation needs to be initialized by running Stage 3 once before simulation begins.

Figure 4.4 shows a high-level overview of VASim's simulation core.

Note that VASim is designed so that researchers can easily augment VASim's simulation core to support arbitrary functionality. Additional stages can be added to support new functionality that relies on global data structures or data structures local to each element. New Elements can be constructed that have arbitrary functionality within special processing stages. VASim is meant to be extensible, and is organized to make these hypothetical modifications low-effort.

4.4 Automata Optimization and Transformations

An automaton that recognizes a particular regular language may not be the most compact, or minimal representation. As a trivial example, imagine two automata that are identical in structure. Both will compute the same language, follow the same paths, and recognize the same patterns, but one is redundant, and the automata can be compressed by at least 50%. Even individual automaton often have opportunities for

compression. An automaton is not guaranteed to be "minimal" unless it is proved to be so. Unfortunately, algorithms for minimizing NFAs are NP-complete and PSPACE-complete [70], making practical minimization of large sets of automata impractical. Thus, heuristics have been studied to improve the performance of NFA minimization. Specifically, NFA reduction aims to greatly reduce the number of states in the NFA, without providing a technically minimal solution.

NFA reduction has two main benefits: state reduction, and redundant computation elimination. State-reduction is beneficial to both spatial and von Neumann processing techniques. For spatial architectures, a reduction in automata states means that we need fewer spatial resources to implement the same automata functionality. This can have important implications if large amounts of redundancy exist. For instance, an automata before compression might not fit into an available spatial architecture, but after compression is implementable. For von Neumann architectures, a reduction in automata states means a reduction in redundant computation. If large amounts of automata computation is redundant, and can be removed, large program speedups could be realized.

Prior work has used pre-orders and binary relations over all automata states to identify redundancies [71, 72, 73]. The basic idea of these algorithms relies on the intuition that any two states that are always reachable by identical sets of strings (prefixes), are redundant, and can be merged. Naive algorithms analyze all possible prefix paths for pairs of states and are, thus, computationally expensive. Most work involves refinement of this naive algorithm to improve performance. In this section, we propose an even simpler heuristic that is easy to implement that also captures a large proportion of compression opportunities for common applications. Our algorithm captures all of the compression opportunities of the pre-order algorithm when automata are acyclic. Cycles in automata are rare in practice, and so, this heuristic is highly effective.

4.4.1 VASim's Common Prefix Merging Algorithm

VASim's common prefix merging algorithm relies on the assumption that most automata do not have looping structures. We make a simple change from the naive pre-order algorithm discussed previously. The pre-order algorithm considers two states and merges them if they 1) have identical character sets and 2) the set of all strings that cause transitions to those two states (prefixes) are identical. Instead of comparing the set of all strings that cause transitions to two states (prefixes), we simply compare the parents of two states. If both states share the same parents, then trivially, they must have the same prefix!

The algorithm first begins by considering the automata start states. If the start states 1) have identical character sets and 2) have identical parents, then they can be merged. To merge states A and B, we adjust all of A's outgoing edges to originate from B, and then delete A. Once the start states are merged, all children of

these states are marked considered, and compared with each other in a breadth first manner. The algorithm continues in this way, considering states unless they have already been considered and marked, until no children are left to consider. Listing 4.4.1 shows pseudo C++ code for this prefix merging heuristic algorithm implemented in VASim.

```
/* Merges all states with identical parents and character sets */
int mergeCommonPrefixes(queue<STE *> &workq) {
    queue<STE*> next_level;
    queue<STE*> next_workq;
    int merged = 0;
    // merge stes with identical parents and character sets
    while(!workq.empty()) {
        STE *ste = workq.front();
        workq.pop();
        while(!workq.empty()) {
            STE *merge_candidate = workq.front();
            workq.pop();
            //if the two STEs have identical prefixes and character sets, merge
            if(ste->compare(merge_candidate)) {
                merged++;
                merge(ste, merge_candidate);
                //else push back onto workq
            } else {
                next_workq.push(merge_candidate);
            }
        }
        // Add all children of first to the next level
        for(STE child : ste->getChildren()) {
            if(!child->isMarked()){
                child->mark();
                next_level.push(child);
            }
        }
        // Recurse, merging all children
        if(next_level.size() > 0)
            merged += mergeCommonPrefixes(next_level);
        // Try another candidate
        while(!next_workq.empty()) {
            workq.push(next_workq.front());
            next_workq.pop();
        }
    }
    return merged;
}
```

Listing 4.1: Pseudocode for heuristic prefix merge algorithm implemented in VASim

4.4.2 Subset Construction

Deterministic finite automata (DFAs) restrict the automata paradigm to allow only one state to be active at any one time. NFAs and DFAs are equivalent in computational power, but DFAs are often easier to execute on von Veumann architectures because they only require one transition rule fetch per input symbol cycle in the automata. Because a DFA state represents a possible configuration of all states in a corresponding NFA, a DFA may be exponentially larger, and take exponential time to construct.

VASim includes an implementation of the classic NFA to DFA subset construction algorithm [39]. Even though the classic subset construction algorithm does not consider homogeneous automata, prior work has proven that subset construction preserves the homogeneity property, and is thus suitable to apply to STEs [40].

Some novel automata processing research relies on taking advantage of the subset construction algorithm, such as hybrid NFA/DFA designs [30]. We leave implementation of these algorithms for future work, but the building blocks exist in VASim to easily implement these hybrid designs.

4.4.3 Automata Striding

Automata striding is a well-known technique to increase the throughput of automata simulation [30]. Automata striding transforms automata so that instead of considering one symbol per cycle, the automata states consider N symbols per cycle. While the new automata considers N symbols, it till only makes a single transition (rather than N transitions), thus reducing the number of transitions in the automata, and increasing throughput by N times.

Similar to subset construction, automata striding transforms automata by partially simulating the automata graph and converting a configuration of the NFA into a new state. By feeding in combinations of N symbols, and generating new states that represent these symbols, we can generate an automata that make a single transition on N symbols. For example, to 2-stride an automata, we consider the configuration of the automata after each combination of any 2 symbols, constructing a new state that represents a single transition after the automata considers that ordered pair. We then repeat this process, simulating the automata for every combination of 2 symbols until no new configurations can be generated. Note that the new, 2-strided automata is driven by symbols with twice as many bits. Care must be taken when designing striding algorithms or automata engines to make sure that systems can handle these larger symbols. Becchi et al. presented algorithms that automatically attempt to compress the number of symbols used by automata in order to enable striding opportunities [30].

Use-Case: Bit-level Striding for File Carving

While striding has mainly been investigated for the purpose of increasing the throughput of automata processing applications, we implemented a striding algorithm in VASim for a different purpose: ease of pattern expression.

Regular expressions and finite automata are usually defined using 8-bit symbols. This is due to the fact that most regular expression patterns are meant to be applied to data that is built up of 8-bit bytes. Currently, there is no way to explicitly write a regular expression with a 1-bit pattern. However, applications of automata processing can benefit from the ability to define bit-level patterns, or patterns that are finer-grained than byte-level. One example is identifying fine-grained patterns in file metadata for the purpose of file carving.

File carving is the task of identifying files from a stream of bytes. File carving is used for data recovery from corrupted data stores (e.g. a corrupted file system), and in data forensics for law enforcement purposes. Current popular techniques look for exact signature matches in file headers [74], and reconstruct files based on simple header/footer identification and pairing. Unfortunately, exact signatures in file headers and footers can be short, and occur many times in a byte stream generating unacceptable false positive rates. For example, the Scalpel file carving tool looks for Zip files by locating Zip file header and footer signatures. A Zip file header signature is only 4 bytes long (\x50\x4b\x03\x04) [75], and is likely to occur probabilistically in other regions that are not Zip file headers. Thus, while Scalpel is fast, Scalpel's false positive rates are high, making the tool less useful. More accurate header analysis is specifically stated as a weakness of the tool, and an area of future work, by its authors [74].

We can improve the accuracy of Scalpel-style file carving by using more precise, regular expressions and corresponding automata graphs, rather than exact match signatures. Regular expression patterns will allow us to specify more precise header and footer patterns, and thus reduce false positive rates. However, some patterns in file metadata are specified at granularities smaller than 8-bit bytes! For example, consider the date and time stamps in a PKZip file that use the standard MS-DOS format [75]:

- **Bits 0-4:** Seconds divided by two, meaning that these bits can take values from 0 to 29 (58 seconds).
- **Bits 5-10:** Minutes, meaning that these bits can take values from 0 to 59.
- **Bits 11-15:** Hours, meaning that these bits can take values from 0-23.

Trying to represent this pattern as a regular expression is a difficult task and presents a few daunting problems. The first problem is that each seconds, minutes, hours bit field is not 8 bits wide. This means that the regular expression writer must consider how two parts of a character set change. The second problem is that one field crosses the boundary of two bytes, forcing the regular expression writer to consider combinations

of patterns. Building an automata, rather than a regular expression, might relieve some of the complexity, but does not make either of the above problems easier.

We propose to solve the above problem using automata striding. We first construct automata with a bit-level alphabet, choosing bit possibilities one at a time rather than attempting to think about byte level patterns. Once a bit-level automaton has been generated, we then apply striding three times (once to form 2-bit symbols, twice to form 4-bit symbols, and a third time to form 8-bit symbols) to automatically generate a byte-level automata that can be understood by any existing regular expression or automata processing engine.

In this way, any complex pattern can be easily constructed using bit-level descriptions, but can also–via striding–be computed efficiently on existing platforms.

To show the usefulness of this approach, we implemented a version of the 2-striding algorithm outlined by Becchi [30] in VASim. We then built a program to construct the bit-level automata based on publicly available descriptions of the PKZip file format header [75]. We then verified that this approach by using the strided automata to identified a set of headers in benchmark zip files.

We expect this algorithm to be hugely influential in any pattern search where bit-level patterns exist, such as file carving, virus detection, and malware scanning, as well as applications where existing alphabets are small, such as DNA analysis.

4.5 Automata Serialization and Code Generation

VASim is capable of serializing and emitting automata graphs into several formats and codes. Serialization can allow arbitrary automata to be processed, visualized, and evaluated on variety of underlying automata processing engines. This capability increases productivity, freeing researchers from dependence on a single platform, and allows researchers to easily compare performance of automata, and high-level languages that emit automata [76] on different software or hardware engines.

As an example of the power of automata transformations, we briefly describe how VASim converts abstract automata to the DOT graph language for visualization, and a functional Verilog HDL code for execution on an FPGA.

4.5.1 DOT File Format for Automata Visualization

Visualization of automata is a highly desirable feature for debugging of applications and different automata transformations and optimizations. We implement a pass over the automata graph that emits the graph in the DOT file format for graph visualization using Graphviz layout engines [77]. In our current implementation,

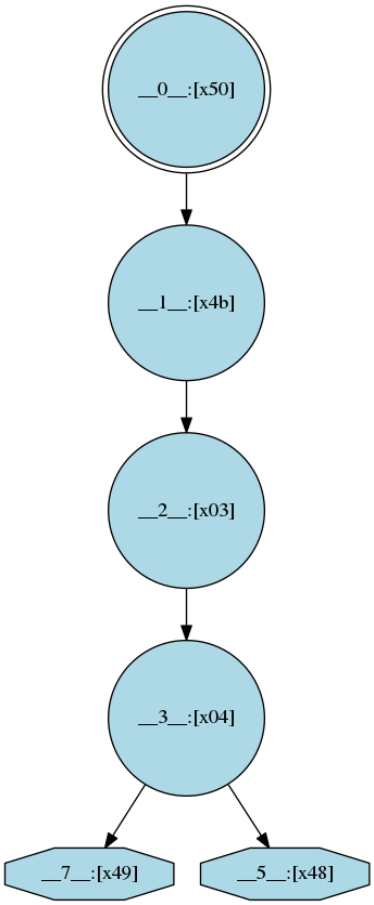

Figure 4.5: Automata graph visualized by VASim's DOT file emission algorithm. Octogons are reporting states. Double circles are start states. Each state is labeled with its ID and character set.

STEs are visualized as circles. Each STE is labeled with its STE ID and the character set it matches on. Special elements are visualized as rectangles. Start states are visualized as double circles. Report states are visualized as octagons. An example visualization using our DOT generation pass is shown in Figure 4.5.

While visualization is extremely helpful for debugging automata construction and transformations, visualization can also help debug dynamic behavior, and also identify optimization opportunities. As an example of the power of visualization of dynamic behavior, we instrumented our DOT file generation algorithm to also include profiling information from the siulation of automata. The algorithm generates a heat map of automata behavior, showing how often automata states compute. Figure 4.6 shows a Hamming distance automata [6] after simulation.

Redder states represent more activity, while greener states represent less activity. Blue states do less than 1% of the computation of the hottest red states. White states were never enabled during simulation and do no computation at all over the profile run. This image was the initial intuition behind automata partitioning and hybrid execution explored in Chapter 8.

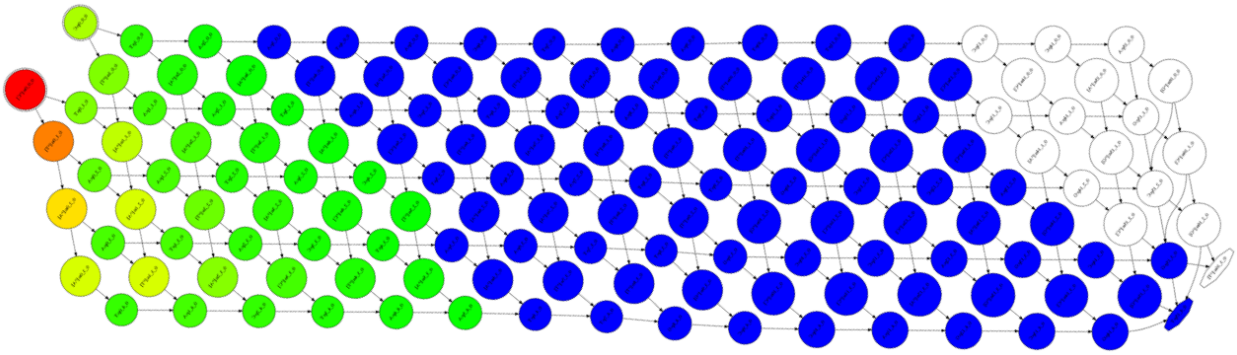

Figure 4.6: Hamming distance automata that was simulated and visualized using VASim's DOT file conversion algorithm. Both structure and dynamic behavior can be visualized allowing users to glean important information about the automata graph and how it computes.

4.5.2 Verilog State Machine Emission for FPGA Evaluation

Reconfigurable fabrics other than Micron's AP (e.g. FPGAs) can also be used to simulate automata in a spatial manner. However, these fabrics lack software tools to convert automata graph representations to synthesizable FPGA designs. To address this problem, we implement a conversion from abstract automata objects in VASim to synthesizable Verilog for execution on FPGAs. Listing 4.5.2 shows a snippet of Verilog that implements a portion of an automata graph.

```
1  ////////////////
2  // STE: __3449__
3  ////////////////
4  // Input enable OR gate
5  wire     __3449___EN;
6  assign __3449___EN = __3448__;
7
8  // Match logic and activation register
9  (*dont_touch = "true"*) always @(posedge Clk) // should not be optimized
10   begin
11      if (Rst_n == 1'b0)
12        __3449__ <= 1'b0;
13      else if (__3449___EN == 1'b1)
14        case (Symbol)
15          8'd47: __3449__ <= 1'b1;
16          default: __3449__ <= 1'b0;
17        endcase
18      else __3449__ <= 1'b0;
19   end
20
21    ////////////////
22       // STE: __3443__
23    ////////////////
24    // Input enable OR gate
```

```
25    wire __3443___EN;
26    assign __3443___EN = __3442__;
27
28    // Match logic and activation register
29    (*dont_touch = "true"*) always @(posedge Clk) // should not be optimized
30      begin
31        if (Rst_n == 1'b0)
32          __3443__ <= 1'b0;
33        else if (__3443___EN == 1'b1)
34          case (Symbol)
35            8'd47: __3443__ <= 1'b1;
36            default: __3443__ <= 1'b0;
37          endcase
38        else __3443__ <= 1'b0;
39      end
```

Listing 4.2: A snippet of verilog code generated by VASim representing two state transition elements (STEs).

All STEs use registers to buffer communication between each other. Each element defines a custom computation according to its input symbols. This computation is then emitted in Verilog code, per automata element. Verilog allows the designs to be portable among different FPGA vendor architectures, and allows designs to take advantage of the many optimizations available in modern FPGA compilers and place-and-route tools.

4.6 VASim Simulation Performance

While performance of automata processing was not considered a main design constraint in the original VASim prototype, VASim uses efficient data structures and algorithms to reduce the amount and expense of all automata computation. Furthermore, VASim is multi-threaded making it trivial to scale automata computation across multiple cores. In this section, we evaluate VASim against Micron's single-threaded automata evaluation tool, *apemulate* [55].

We consider the 12 real-world automata processing benchmarks from the ANMLZoo [22] automata processing benchmark suite for evaluating VASim against Micron's automata simulation tool *apemulate*. ANMLZoo contains a diverse set of automata benchmarks with varying static and dynamic properties, allowing for fair comparisons of automata processing engines and architectures [22]. ANMLZoo is presented in more detail in Chapter 5.

Table 4.1 shows runtimes of VASim and *apemulate* over all benchmarks in ANMLZoo. Experiments were performed on a 3.3GHz 6-core (12-thread) Intel i7-5820K, with 32GB of RAM using version 1.6.5 of the AP SDK [55]. Each automata was run in its original form, and then run after applying optimizations:

Benchmark	Family	VASim (s)	apemulate (s)	VASim Opt (s)	apemulate Opt (s)	VASim Speedup
Snort	Regex	65.86	4,584.60	3.84	838.14	218.27×
Dotstar	Regex	40.06	3,058.81	1.09	208.64	191.42×
ClamAV	Regex	4.66	607.50	0.62	77.58	125.12×
PowerEN	Regex	8.81	3,294.99	1.27	881.38	694.01×
Brill	Regex	92.46	3,281.95	2.25	132.43	58.87×
Protomata	Regex	78.78	3,431.30	28.82	1,405.21	48.77×
Hamming	Mesh	8.61	393.06	7.25	339.93	46.90×
Levenshtein	Mesh	6.26	223.86	5.19	194.92	37.57×
Entity Resolution	Widget	78.90	killed	12.82	122.85	9.59×
SPM	Widget	561.97	8,252	561.97	2,225	3.96×
Fermi	Widget	187.16	3,451.96	183.89	2,824.97	15.37×
Random Forest	Widget	99.82	4,538.11	66.90	1,045.21	15.63×

Table 4.1: Performance of VASim compared against apemulate. *Opt* refers to performance after applying both tool's redundant state elimination passes. VASim is at least 3.96× and up to 694× faster than apemulate even after optimizations are applied. VASim's performance for SPM is relatively low because Micron's compiler applies more sophisticated state reduction algorithms than our heuristic approach.

prefix merging for VASim and all standard optimizations enabled for apemulate (-O1). Output reporting was enabled for both tools and runtime was measured using the Unix *time* command.

When prefix merging is enabled in VASim, the simulation core is at least 3.96× faster than apemulate, and is greater than 125× faster than apemulate in 4/12 applications.

4.7 Conclusions

This chapter presented the Virtual Automata Simulator or VASim [78, 79]. VASim is an open-source framework for automata processing research. It combines a flexible, object-oriented automata programming framework with a high-performance simulation core. VASim is capable of building, optimizing, transforming, converting, and simulating automata graphs, and is designed to be flexible to support modifications to answer new research questions.

VASim is in active use by both academia and industry. As of the writing of this chapter, VASim had been downloaded over 210 times by 140 unique users from GitHub [80]. VASim has been used to publish papers at top-tier computer architecture conferences including the International Symposium on Computer Architecture (ISCA) [81], the International Symposium on Microarchitecture (MICRO) [82], and the International Symposium on High-Performance Computer Architecture (HPCA) [13, 83].

VASim continues to evolve to support the needs users, and is under active development. Current projects include re-factoring and stability improvements, and implementation of more sophisticated NFA reduction algorithms.

Chapter 5

ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures

Because acceleration of automata processing has traditionally been motivated by network intrusion detection, new automata-processing engines on different architectures are evaluated using a small, relatively homogeneous set of existing representative regular expression rulesets [4, 7, 45]. Synthetic benchmark suites such as IBM's PowerEN suite [50] allow scientists to do more controlled studies of regular expression processing. Becchi et al. [84] created a synthetic regular expression rule and stimulus generator to help researchers do even more accurate sensitivity studies on regular expression processing engines.

While these regular expressions applications and benchmarks are valid and important real-world use cases for automata processing, they represent a very narrow range of all useful automata. Regular expressions, as written by humans, tend to be converted by classic algorithms into non-deterministic finite automata with very similar average structure, dynamic behavior, and matching complexity, and thus do not represent a wide range of possible useful automata structures or behavior.

Micron's Automata Processor and accompanying software development kit have made prototyping and development of automata-based (rather than regular-expression-based) pattern matching engines much easier. No good quantitative metric exists to measure the relative merits of either approach, but in our experience, directly constructing finite automata is an easier and more intuitive way for defining complex regular languages and pattern mining tasks. The availability of this software and hardware has led to the

development of a large number of new, non-obvious automata-based applications in domains such as big data analysis [18], data-mining [16, 15, 17], bioinformatics [12, 6, 11, 13], high-energy particle physics [21], machine learning [8, 9, 10], pseudo-random number generation and simulation [36], and natural language processing [19, 20] that can differ significantly in static structure and dynamic behavior from existing regular expression benchmarks [4, 7, 45, 50].

This new diversity in automata-based applications, development tools, software recognition engines, and hardware architectures for automata processing, motivates a standard but flexible application and engine repository for fair evaluation of new automata-processing algorithms, architectures, and automata applications.

This chapter presents ANMLZoo, a benchmark suite of automata-based applications for prototyping and evaluation of both software and hardware automata-processing engines. ANMLZoo contains 14 different automata-based applications that represent four major classes of automata: regular expression rulesets, string scoring meshes, programmable widgets, and synthetic automata, all of which may stress automata processing algorithms and architectures in different ways. Furthermore, ANMLZoo contains source code for high-performance automata processing engines that we have deployed for CPUs, Intel's XeonPhi, and GPUs, and can accomodate new state-of-the-art high-performance algorithms and architectures as they are developed.

The main contributions of this chapter are the following:

- The creation of a public repository for standardized finite automata benchmarks and input stimuli, allowing for easy and fair comparisons of von-Neumann and reconfigurable fabric-based automata processing engines.

- The inclusion of parameterizable automata-generation scripts for some benchmarks, allowing for sensitivity studies of different static and dynamic automata properties of both real applications and synthetic automata.

- The analysis and categorization of automata using both qualitative and quantitative metrics. We consider both well-known metrics (node count, edge count, active set, etc...) and novel metrics (activity compressability, character set complexity) to categorize automata benchmarks. We show how some of these metrics impact performance on different architectures.

- The creation of a public repository for automata processing engines for several architectures for easy and fair comparisons of new algorithms and implementations to prior work. We include high-performance

state-of-the-art automata-processing engines that we developed for CPUs (including Intel's XeonPhi), and for GPUs.

- The designation of a common tool for ANMLZoo automata manipulation, optimization, transformation, and analysis. Thus, new automata representations and optimizations can be easily shared among researchers and compared to prior work.

To demonstrate the usefulness of ANMLZoo, we investigate performance of four automata processing engines on four different macro architectures (Intel i7-5820K, Intel's XeonPhi 3120, a Maxwell-based NVidia Titan X GPU, and Micron's first-generation Automata Processor) and identify relative bottlenecks in each software engine and architecture pair.

5.1 Problems with Existing Rulesets and Generators

Research into fast regular expression processing engines has traditionally been motivated by deep packet inspection, which includes applications in the network intrusion detection system (NIDS) community. NIDS tools attempt to match regular expression patterns that identify possible malicious network behavior like attacks or probes against streaming network packet traffic. An example NIDS regex pattern (and an equivalent NFA) from the Snort ruleset searching for an click-to-launch executable within a PDF file is shown below.

$$/\backslash x2fF\backslash s*(<<|)\backslash s*\backslash x2fDOS\backslash s*\backslash x28/smi$$

NIDS rulesets such as Snort [4], as well as virus detection rulesets such as ClamAV [7], and synthetic rulesets such as PowerEN [50] have been popular rulesets for benchmarking existing regular expression engines [51, 30]. However, it is desirable to have a common and flexible methodology for benchmarking and conducting sensitivity analysis on regular expression engines with parameterizable rulesets and input stimuli. Becchi et al. [84] constructed a synthetic regular expression generation tool that parameterized regular expression features that make DFA conversion expensive. This tool also includes an automatic trace generation tool, which can tune input streams to induce various levels of activity in any automaton. However, this tool was motivated and designed to generate regular expressions and inputs to better evaluate deep packet inspection engines and architectures, and not for arbitrary automata processing.

Similarly, new architectures for automata evaluation [51] are designed for and evaluated using the above mentioned patterns, or with simple exact match strings, highly compressable binary trees [85], and/or finite automata with a very small number of states.

As our characterization will show, existing regular expression and automata benchmarks are either very similar in static and dynamic properties, or not publicly available, easily accessible, or in a common format.

Benchmark	Family	States*	Compressability	Node Degree*	Charset Complexity*	Active Set*	Activity Compressability
Snort	Regex	34,480	50.04%	1.13	8.74	98.45	75.71%
Dotstar	Regex	38,951	59.6%	1.01	8.28	3.25	92.79%
ClamAV	Regex	42,543	14.12%	1.02	7.86	4.30	94.78%
PowerEN	Regex	34,495	14.85%	1.06	8.11	31.15	66.20%
Brill†	Regex	26,364	38.20%	1.49	8.75	14.28	99.14%
Protomata†	Regex	38,251	8.95%	1.04	19.44	554.281	63.15%
Hamming	Mesh	11,254	0.81%	1.71	9.89	240.1	15.78%
Levenshtein	Mesh	2,660	4.45%	3.36	8.0	88.02	22.93%
Entity Resolution	Widget	5,689	94.02%	6.38	8.60	10.62	99.11 %
SPM	Widget	100,500	0%	2.1	6.58	6,331.32	0%
Fermi	Widget	39,033	4.29%	1.48	8.18	3,854.45	∼0%
Random Forest	Widget	71,574	5.00%	1.053	14.26	968.64	1.26%
BlockRings	Synthetic	44,352	NA	1	8	192	NA
CoreRings	Synthetic	48,002	NA	1	8	2	NA

Table 5.1: ANMLZoo benchmark suite. † Newly published automata-inspired regex-like rulesets. Results are gathered using representative input streams should be considered baseline results, and may change with new algorithms, implementations, and architectures.

This makes it extremely difficult or even impossible to evaluate improvements over existing state-of-the-art publications. Many state-of-the-art software engines and infrastructures for automata processing and transformation are also not publicly available, again making it difficult or impossible to do fair evaluations of existing automata-processing algorithms and implementations on different architectures.

5.2 ANMLZoo: an Automata Processing Benchmark Suite

To address the above drawbacks with the current methodology for benchmarking of automata processing engines we present ANMLZoo, a repository for automata benchmarks, input stimuli, and software engines and infrastructures for fair benchmarking of new automata processing engines [1]. Each ANMLZoo benchmark is shown in Table 5.1.

Below we list each application in ANMLZoo, including both existing popular regular expression benchmarks, as well as a new set of recently published automata-based applications that together form a much more diverse starting point for benchmarking of automata-processing engines.

Snort[4] are regular expressions extracted from a snapshot of the snort ruleset, commonly used to benchmark regular expression processing engines.

Dotstar [24] is a combined set of synthetic regular expressions from Becchi et al. [24] containing all variations of the synthetic dotstar rules created from the backdoor Snort rules and the spyware rules used in that evaluation.

ClamAV [7] is a set of regular expression signatures for identifying virus signatures in files. Our benchmark includes ClamAV rules with small (< 64) quantifiers and no ranges.

[1] Deyuan Guo, Elaheh Sadredini, Tommy Tracy, Chunkun Bo, Nathan Brunelle, and Matt Grimm all contributed to either generation or consultation on generation of these benchmarks.

PowerEN [50] is a combination of over 2000 regular expressions from the PowerEN "complex" regex rule set.

Brill [19, 20] is a set of over 2000 Brill tagging rules.

Protomata [6] is a set of 2340 real and randomly generated protein motif signatures.

Hamming [12] is a set of 93 Hamming distance automata used to calculate the number of mismatches between a randomly generated encoded string and random input sequence.

Levenshtein [11] is a set of 24 Levenshtein automata designed to calculate the edit distance between an encoded DNA sequence and an input DNA sequence.

Entity Resolution [18] is a set of automata designed to identify whether input name sequences match a certain encoded pattern.

SPM [16] or Sequential Pattern Mining, is an automata-based application to identify groups of related items in baskets.

Fermi [21] is a path recognition algorithm that looks for sequential series of ordered coordinates defining a particle path.

Random Forest [8] is an encoded and compressed implementation of a random forest ensemble classifier for handwriting recognition.

BlockRings are synthetic automaton rings with deterministic behavior meant to occupy each block (192) on an AP chip.

CoreRings are synthetic automaton rings with deterministic behavior meant to occupy each core (2) on an AP chip.

The difference between automata constructed by regular expressions and other modern automata-based applications in ANMLZoo can be easily quantified by looking at both static and dynamic properties.

Before we perform static or dynamic analysis, all automata are compressed using *common-prefix merging* or CPM as described in Chapter 4. CPM merges redundant states from the automata in a breadth-first manner, from start states to end states, while preserving automata correctness. This optimization can greatly reduce the size of, and redundant traversals for, automata, and is thus used for baseline static and dynamic evaluation of automata. However, we do not claim that these are optimally minimized automata, and thus they may be compressed further. Dynamic properties can vary greatly depending on the corresponding input stimulus, and so metrics like *active set* should also not be considered inherent properties of the benchmarks or applications, but rather based on the particular automata evaluated and behavior induced by a representative input.

We considered five metrics to quantify differences in automata applications. Each metric is described below:

- **States:** The total number of states (STEs) in the common prefix-merged (CPM) automata graph. The capacity of an AP chip is 49,152 STEs. State counts lower than this number indicate lower utilization of on-chip resources and a harder routing task for the AP compiler and fabric. State counts higher than this number indicate the Micron compiler was able to identify compression opportunities other than CPM.

- **State Compressability:** The percentage of redundant states removed by CPM. High compressability reduces pressure on reconfigurable resources in FPGAs and the AP, but also may improve cache behavior in von Neumann engines.

- **Node degree:** The average output degree of each node. Higher node degrees and more connectivity indicate a harder place and route task for spatial automata processing engines like FPGAs and the AP.

- **Character set complexity:** We use the Quine-McCluskey algorithm [86] to calculate the minimum number of boolean terms required to compute the boolean match function corresponding to the character set of an automaton transition rule (STE). This metric reflects the average difficulty in building a circuit to compute the match function of a particular STE.

- **Active Set:** The average number of active states. Larger numbers of active states require more transition rule fetches. Thus, this is a proxy metric inverse to performance in von Neumann architectures. Spatial architectures like the AP are unaffected by active set because all transitions are accomplished in parallel in a single cycle if the design can successfully place and route.

- **Activity Compressability:** The average amount of redundant activity removed by CPM. This metric roughly indicates how much performance is gained on von Neumann engines from the CPM optimization.

Table 5.1 shows that many benchmarks derived from regular expressions have similar static properties. Each regex benchmark has an average node-degree of about 1, reflecting the long strings of automata states that are often emitted from typical regular-expression-to-automaton conversion algorithms. In contrast, applications such as mesh automata and Entity Resolution (which uses Hamming automata as a sub-kernel) have many more output edges and represent a much more complex structure and routing task for spatial automata-processing fabrics. Regular-expression-like automata tend to have a high number of common prefixes, reflected in high state and activity compressability factors. In contrast, automata widgets such as Fermi are designed to compute non-obvious recognition tasks and generally have much less redundancy by design. We explore automata compressability via CPM and its effect on performance in Section 5.3.

ANMLZoo provides the following features:

Diverse Automata Structure and Behavior: ANMLZoo is originally divided into four major automata families: regular-expression-derived automata (Snort, ClamAV, PowerEN, Brill, Protomata), automata meshes for string scoring (Hamming, Levenshtein), structured processing elements or "widgets" (SPM, Random Forest, ER, Fermi), and synthetic automata with exact known properties (BlockRings, CoreRings). All applications are quantitatively diverse in both static and dynamic properties, and reflect real-world uses of automata.

Standard Candles: Each ANMLZoo benchmark provides at least one file that defines a standard set of automata that max out the resources of a single first-generation Micron D480 AP chip. While there is no one "correct" way to standardize trade-offs among both automata state-size, activity, and connectivity, we chose this metric as a compromise to allow easy and fair comparisons between different reconfigurable data-flow architectures and von Neumann automata processing engines. We call these standardized automata the ANMLZoo *standard candles*. Because standard candle automata max out the resources of an AP chip, we can easily and fairly compare application performance on other automata processing architectures against different deployment scenarios of small 4W AP D480 chips. For example, performance of 1 AP rank (8 parallel AP chips consuming 8 parallel input streams), is trivial to deduce via multiplying the performance of one AP chip by 8. This feature is exemplified in Section 5.8. Each *standard candle* benchmark is also accompanied by a corresponding stimulus of 1MB and 10MB, for testing and evaluation respectively. Versions of standard candles and input stimulus may evolve to adapt to the needs of the community, but automata and input stimuli will never be removed from the suite allowing for easy and fair comparisons to prior work.

Written in ANML: Each application is defined using Micron's Automata Network markup language [55] or ANML. ANML allows a standard but flexible method for defining automata networks. ANML is an XML-like language that is used to define automata computation graphs. Applications that are defined as regular expressions can be converted to ANML automata using Micron's SDK [55]. If a benchmark is derived from regular expression rulesets, these rules are also included in the suite.

Parametric Automata Generation Scripts: Where possible, automata generation scripts have been provided to facilitate sensitivity analyses of different automata-processing applications and architectures. For example, Section 5.4 shows how performance of von Neumann architectures is impacted by varying fixed properties of synthetic automata and Section 5.7 shows how AP chip utilization is affected by varying dimensions of mesh automata (Hamming, Levenshtein). These sensitivity analysis cannot be done using the fixed benchmarks like the standard candle automata.

To demonstrate the usefulness of ANMLZoo, the following sections present five different experiments exploring the sensitivity of different engines to different types of automata, exposing bottlenecks in automata processing engines on von Neumann and spatial architectures, and relative advantages of automata processing

Figure 5.1: Sensitivity of VASim performance in response to additional automata rules. Performance of automata with many common prefixes and high activity compressability (ER) are less sensitive to additional rules. This indicates average automata activity after common optimizations, rather than total rule-count, is a better predictor of performance.

on each available architecture. For each CPU experiment, we use a 6-core (12-thread) Intel i7-5870K clocked at 3.3GHz with 32GB of RAM clocked at 2166MHz. This server also acts as the host CPU for the following accelerators. For each many-core CPU accelerator experiment, we use a 57-core (228-thread) Intel XeonPhi 3120p clocked at 1.1GHz. For each GPU accelerator experiment, we use an NVIDIA Maxwell-based GTX Titan X clocked at 1GHz. For each AP fabric utilization experiment, we use Micron's AP SDK version 1.6.5.

5.3 Parallel Automata Rule Scaling

Many regular expression processing applications are concerned with the number of parallel "rules" or automata that a given engine or architecture can process. For von Neumann architectures, more automata computed in parallel may mean more transitions to compute, and more pressure on the memory hierarchy. For data-flow, reconfigurable fabric automata-processing architectures, more parallel automata lead to a higher number of states, and thus more pressure on the underlying reconfigurable fabric capacity and routing resources. Thus, the more rules an engine is capable of quickly processing, the more desirable the engine.

However, "number of rules" is a poor metric to measure the amount of work being done by an automata engine. As an example, we consider two applications from ANMLZoo (Entity Resolution and Fermi) and vary the number of rules processed by a single thread on the CPU, measuring the sensitivity of performance of VASim to the number of automata being computed. Figure 5.1 shows the results of our experiment.

We plot normalized performance of common-prefix-merged versions of the automata rules (compressed) and the original (uncompressed) versions of the automata rules for both Entity Resolution and Fermi. The common-prefix-merged version of Entity Resolution, while initially incurring a severe penalty for additional rules, quickly reaches a point where additional rules have little impact on performance. This is due to the high activity compressability of Entity Resolution, as the redundant states removed by common-prefix merging were also responsible for a high amount of redundant activity in the original automata.

Figure 5.1 also plots the performance cost of adding rules without the benefit of common-prefix merging. The performance penalty of additional rules is much more severe, and does not plateau, indicating new rules require significant additional activity and computation when uncompressed.

For the Fermi application, both the CPM and non-CPM versions have near-identical performance characteristics. This is because Fermi has an extremely small activity-compressability factor. Specially-designed automata like Fermi are therefore extremely important to consider when characterizing new automata engines and optimizations, as they are harder to compress, and therefore pay a larger penalty when computing additional rules.

5.4 Visited Set and Active Set Sensitivity

Because automata processing on von-Neumann architectures requires many sequential accesses to memory, performance of automata processing on these architectures has been shown to be limited by access latency in the memory hierarchy [51, 85]. However, this bottleneck and its impact can greatly depend on the underlying automata engine algorithm and implementation. In the previous section, we saw that automata activity, rather than "number of rules," was the main factor hurting performance, but this activity was not measured or controlled for.

In general, it can be difficult or impossible to guarantee certain properties of automata for controlled experiments. It is therefore important to have a set of automata benchmarks (or generation tools) in the benchmark suite that can precisely vary metrics such as the *visited set* (the set of states consistently visited during computation) and the *active set* (the number of active states which need to perform memory accesses per cycle). These synthetic automata and synthetic automata generation tools allow for controlled experiments measuring the specific impact of memory hierarchy latency or throughput on total performance.

We present a parametric synthetic automata design to control for the ratio of active set to visited set. The synthetic automata design is shown in Figure 5.2 [2].

[2]Prof. Nathan Brunelle originated the idea for the synthetic automata design, and built the scripts to generate them.

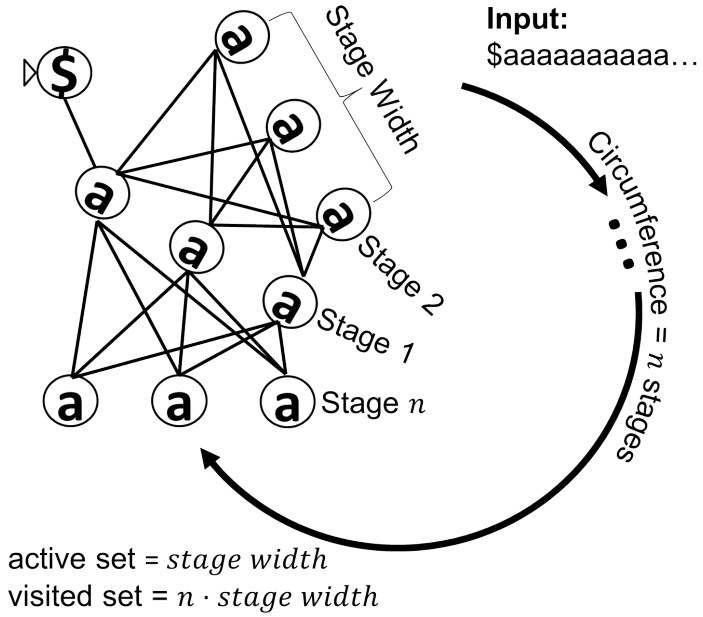

Figure 5.2: Parameterizable synthetic automata design. Each ring is guaranteed to have a constant active set and visited set, and is driven by an easy-to-generate input string. This instance has width 3, thus active set 3. Each stage is fully connected with its succeeding stage to form a continuous ring. The circumference, n, is derived using the equation $n = \lceil \frac{visited}{width} \rceil$.

Each automaton is organized as a ring of stages. Each stage in the ring has a fixed number of states that is always activated by the previous stage. This property guarantees that at any one cycle, the *active set* in any one ring is equal to the width of the stage. Each ring is also of a fixed circumference (i.e. the number of stages in the ring). Therefore, the total *visited set* of the automaton is the width of the stage times the number of stages in the ring. This design allows us to individually control for both active set and visited set, and isolate the impact of each on performance of different automata-processing engines. Below shows the results of VASim performance while varying active set and visited set independently. We vary the visited set of a single ring by multiples of 10 states from 100 to 100,000 states, and vary the active set of each ring by 1 from 1 to 20.

The more average states visited, the larger the pressure on the caches in the memory hierarchy of a von Neumann architecture. Therefore, guaranteeing that the visited set fits into L1 or L2 caches of a CPU can be extremely important for high-performance.

When the number of states is between 100 and 10,000, increasing the size of the visited set has little impact on performance. This indicates that the entire visited set fits within a single level of the memory hierarchy, and so a larger number of states does not impact the performance of computing transitions for the active set. However, there is a relatively large impact when increasing the size of the visited set from 10,000

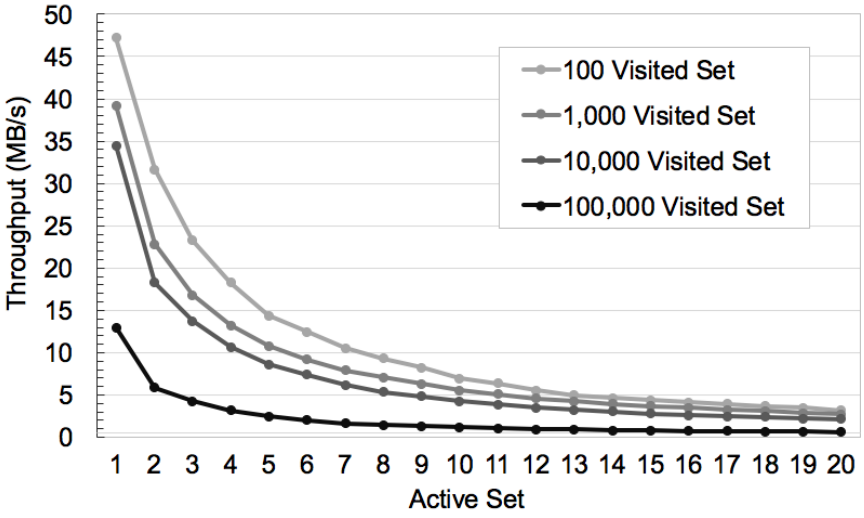

Figure 5.3: Sensitivity of automata simulation performance to changes in the active set (number of states considered per cycle) and the visited set (number of states consistently visited). Performance is much more sensitive to increases in active set. The visited set impacts performance when its size grows larger than the size of an available level of cache.

to 100,000.

While increasing the size of the visited set does impact performance, slight increases in active set can have extremely large impacts on performance. Because automata structure can be irregular and behavior is often unpredictable, it is difficult to guarantee locality of access. Therefore, to improve automata processing performance on the CPU, VASim must work to reduce the size of the average active set via automata optimizations and transformations, but also maintain an automata visited set size that optimizes performance.

5.5 Automata vs Input-level Parallelism Scaling

Because it can be difficult or impossible to reduce the active set of automata, and improving memory latency at the architectural level can be extremely expensive, automata engines often attempt to exploit parallelism among independent automata and among automata input streams to hide the latency of individual transitions and increase throughput of automata engines. This section explores sensitivity of automata engines and architectures to these two dimensions of parallelism–parallel automata and parallel input streams. Distinct automata can be divided into an equal number of groups (G), and the input stream can be divided into an equal number of sections (S). Thus, we can launch $G \times S$ number of CPU threads or GPU thread-blocks to compute in parallel.

We pick three applications: Protomata, Hamming, and Random Forest to illustrate how different families of applications (regex, mesh, and widget) respond to varying automata group and stream parallelism. For

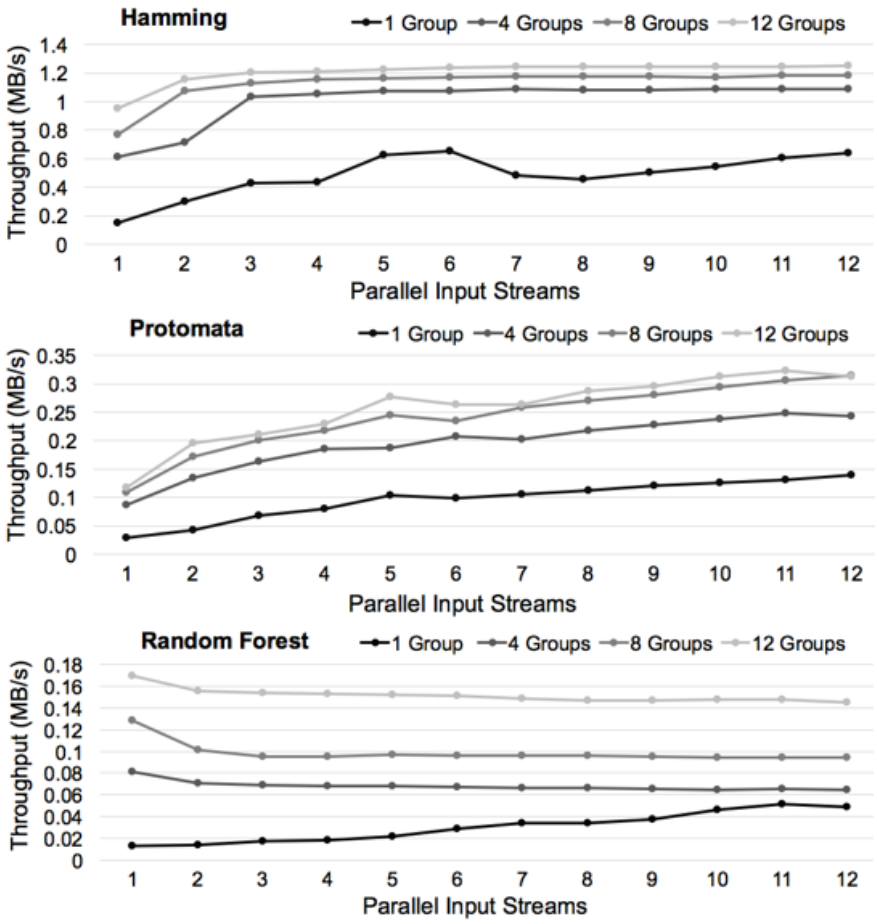

Figure 5.4: Hamming automata benefit most from automata-level parallelism. Protomata benefits from parallelism in both dimensions. Random Forest only benefits from automata-level parallelism.

each application, we pick 4 sets of groups and vary the number of parallel streams on both the VAsim (CPU) and iNFAnt2 (GPU) baseline engines. Results are presented below.

5.5.1 CPU Parallel Scaling

Results from varying automata groups and parallel input streams on VASim, our baseline CPU automata engine, are shown in Figure 5.4.

Hamming automata seem to favor more parallel automata groups and are not accelerated by increasing the number of parallel packet streams. This is because Hamming automata have relatively little activity compression and so parallel threads computing parallel automata are more likely to be doing distinct, non-redundant work. Thus it is better to have single threads operate on smaller, distinct automata that may have good behavior in an individual CPU's L1 cache.

Protomata is much more responsive to both automata-level and input-level parallelism. This is because

Protomata has a small number of automata that have a much greater level of activity than others. Because VASim is not equipped to parallelize work within individual automata, the threads that are responsible for these "problem" automata run much slower and bottleneck performance. While automata-level parallelism cannot accelerate problem automata, stream-level parallelism can. Thus Protomata performs best with eight parallel automata groups (8), but a larger number of packet streams (12).

Random Forest has an extremely low level of activity compressability and so benefits most from distributing automata across many threads. Random Forest benefits so much from parallel automata computation, that any additional thread contexts for computing parallel input streams hurts performance, even when cores are underutilized. This indicates that shared per-chip (as opposed to per-core) resources like L2 and L3 cache are over-utilized, and important for performance where active set is high.

These experiments shows that parallelization strategies for CPU-based automata processing depend highly on the automata topography, compressability, and dynamic behavior.

5.5.2 GPU Parallel Scaling

Results from varying automata groups and parallel input streams on iNFAnt2, our baseline GPU automata engine, are shown in Figure 5.5 [3].

Unlike the CPU-based engine, Hamming automata on the GPU overwhelmingly favor more parallel input streams. Hamming performs best when each CUDA thread block operates on all meshes simultaneously and there are more than 560 parallel blocks operating on different sections of the input stream. This highlights the ability of the GPU to hide the latency of any individual memory access by executing an extremely large number of parallel tasks. Because there may be a relatively small number of parallel accesses in any one benchmark (e.g. Hamming has an average active set of 240 over 49 distinct automata when prefix merged) it is generally better to exploit input stream-level parallelism for latency hiding on the GPU.

Protomata shows similar performance characteristics to Hamming. However, a single group does not universally perform best. Dividing the automata into eight groups performs better as the number of parallel streams is increased. This reflects sensitivity to utilization of per-GPU stream-processor resources such as shared memory and L1 cache. The total performance of the GPU engine relies on a balance of NFA transition table size and stream-level parallelism that is highly application specific.

Random Forest, as discussed earlier, has a small amount of activity compressability, and therefore favors computation by more parallel groups, with smaller, more efficient NFA transition tables. Random Forest performs best on the GPU when split into 48 distinct groups. However, it is still the case that too many

[3] Vinh Dang gathered results for these scaling experiments

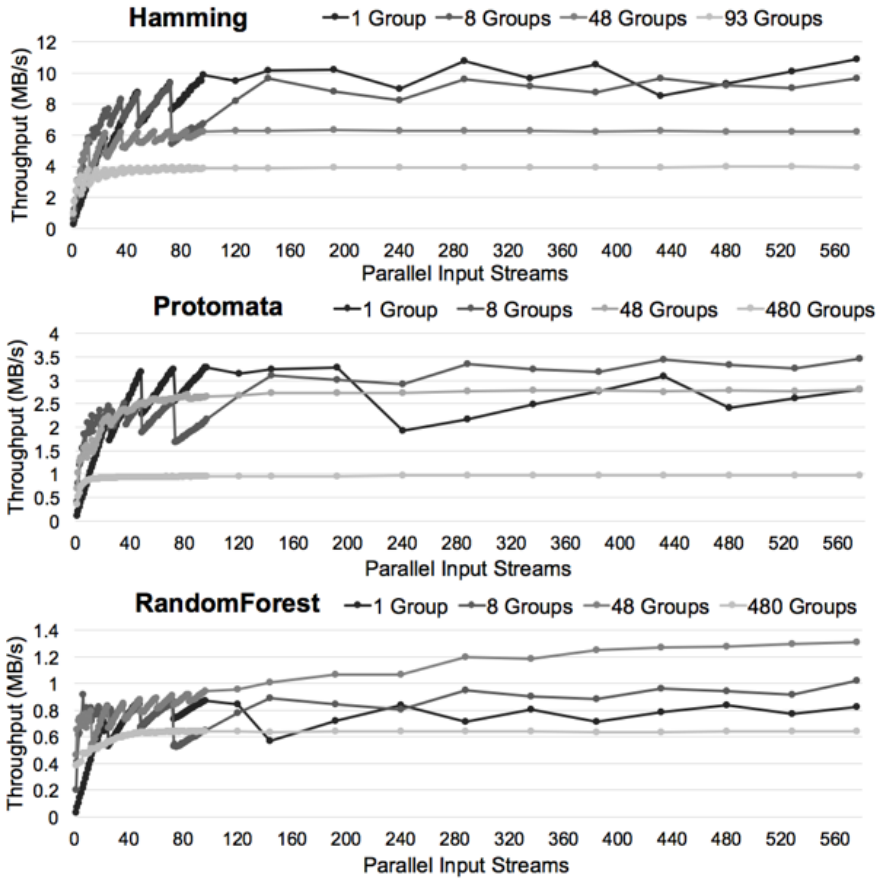

Figure 5.5: Protomata, Hamming, and Random Forest all benefit from a massive amount of stream level parallelism, however appropriate care must be taken to tune automata groups to match GPU core resources.

automata groups will limit stream level parallelism, and reduce the ability of the GPU to hide the latency of transition table lookups.

These experiments show that GPU automata processing engines mostly favor parallelization via parallel input streams. If an application allows its input stream to be divided among parallel threads, the GPU can better hide the long latencies associated with SIMD control-flow and memory divergence.

5.6 NFA vs. DFA Engines on the GPU

The variable topography and dynamic parallelism of NFAs can be especially difficult to efficiently map to the GPU's SIMD architecture. Thus, deterministic finite automata (DFA) have been explored as a possible alternative method of automata processing to better exploit the GPU's available resources [87, 25]. DFAs are equivalent automata that are constructed so that only one state can be occupied at any one time. A DFA

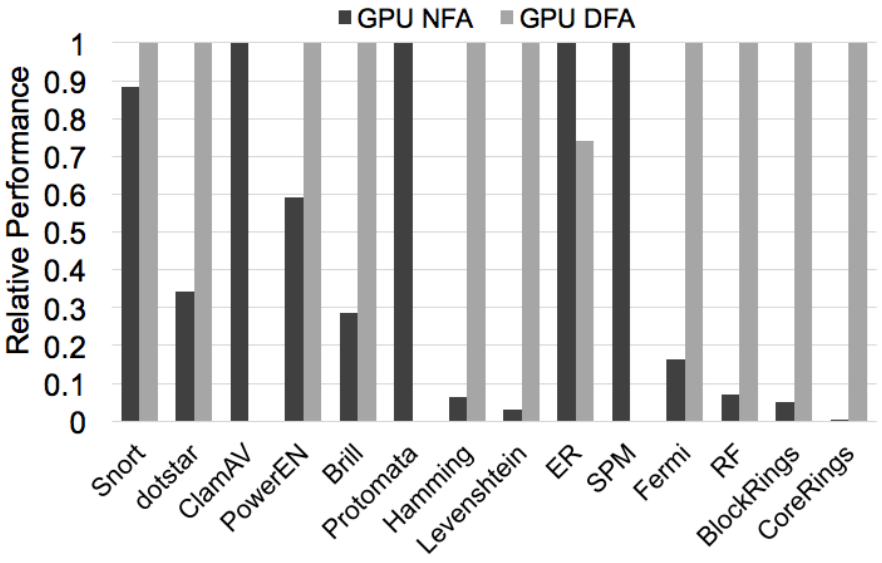

Figure 5.6: Relative performance of NFA and DFA engines over all benchmarks in ANMLZoo. DFAs for ClamAV, Protomata, and SPM were too expensive to construct due to space or time costs.

state therefore represents a particular configuration of NFA states. Because of this relationship, DFAs can be potentially exponentially larger than their equivalent NFAs, and exponentially expensive in time to construct.

We use the DFA generation tool developed by Becchi [45] to convert as many ANMLZoo NFAs to DFAs as was possible. Some ANMLZoo applications took too long, or required too much memory to be converted to a reasonable number of DFAs and were ignored. The GPU DFA engine in iNFAnt2 assigns individual CUDA threads within a thread-block to processes a particular DFA. In contrast, the iNFAnt2 NFA engine maps the computation of entire NFAs to CUDA thread-blocks. Because each DFA only ever requires one, deterministic transition per cycle, the number of control-flow and memory operations per cycle is much lower. Furthermore, divergence among threads is much less likely, leading to higher utilization of SIMD units.

Figure 5.6 shows the relative performance between our baseline NFA and DFA engines achieved using the optimal block and grid size, and thread and stream configuration for each application [4]. The DFA-based engine traverses exactly one state per symbol, independent of the automaton and input stream, while the NFA-based processing engine follows a number of state transitions.

Unsurprisingly, Figure 5.6 shows that the DFA engine–when DFAs are able to be created–is the best solution for every benchmark with the exception of ER. This is due to the relative simplicity of the DFA kernel, and the reduced number of total instructions required to compute the automata. We compared profiling information gathered by NVIDIA's profiling tool *nvprof* on the Levenshtein automata. The NFA kernel executed over $5,700$ times more control flow instructions than the equivalent DFA kernel, and 43 times more memory instructions per input symbol.

[4] Vinh Dang gathered results for these experiments

In some applications (Snort), DFAs do not give significantly better performance compared to NFAs in iNFAnt2. This is because very few NFAs can be combined into single DFAs. Specifically, for the ER benchmark, the large number of required DFAs causes the iNFAnt2 DFA engine to perform worse than the iNFAnt2 NFA engine.

5.7 Mesh Scaling and AP Fabric Utilization

Mesh automata, such as the Hamming and Levenshtein automata, score input strings by positionally keeping track of input mismatches with an encoded string.

Hamming-distance automata have been shown to help accelerate both DNA and protein [12] motif search algorithms. These automata use a simple kernel–match or mismatch–to positionally keep track of the number of mismatches between the input and encoded string using automata states. A mismatch will force a transition to a new row of states that represent mismatches one greater than the previous row. In practice, it is usually only important to keep track of mismatches up to a particular score threshold, and so rows can be pruned from Hamming distance automata to decrease unnecessary states and computation. Because Hamming distance automata only ever considers the match or mismatch kernel, the fan-out and fan-in of any individual state is always less than or equal to 2, no matter the length of the input string, or the number of mismatches the automaton is programmed to compute.

Levenshtein automata use a more complex kernel to keep track of differences between an encoded string and an input string. While Hamming distance only considered matches and mismatches, Levenshtein automata additionally keep track of possible insertions in the input string and deletions from the encoded string, ultimately scoring the number of "edits" (edit distance) required to transform one string to the other up to a certain edit threshold. Because the Levenshtein automata must account for any number of deletions up to the threshold, the maximum fan-out and fan-in of any individual state grows linearly with the size of the threshold.

This increase in the connectivity is not problematic for von Neumann-based automata processing engines, where arbitrary automata networks can be easily stored in memory. However, high connectivity can be problematic for spatial architectures that rely on a reconfigurable routing matrix to lay out all possible datapaths in the automata networks. To show impacts of connectivity in mesh automata on spatial architectures, we vary both encoded string length and a score threshold for Hamming and Levenshtein automata. We then compile the designs for the AP and measure their on-chip routing utilization. Figure 5.7 plots the resulting routing complexity vs. encoded string length for ten different automata.

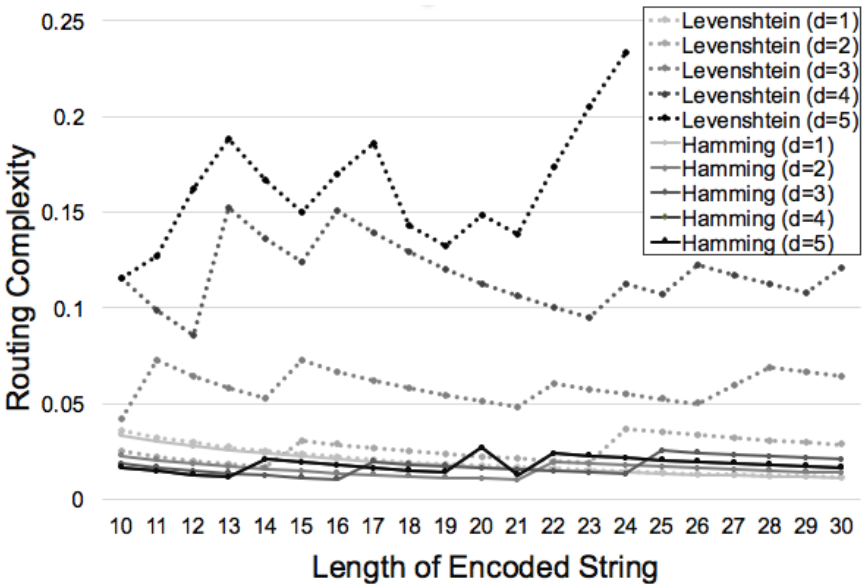

Figure 5.7: Hamming automata have a constant fan-in/fan-out per STE and therefore have relatively low routing complexity that is not impacted by the dimension of the mesh. The node degree of Levenshtein automata grows linearly in the size of the encoded edit distance threshold, therefore routing complexity is very sensitive to this dimension. Levenshtein automata with edit distance threshold 5 (d=5) fail to route on the current AP hardware past encoded string length 24.

Hamming automata are relatively insensitive to increases in both dimensions–the encoded string length and the score threshold–of the automata. This reflects the constant fan-in/fan-out per match/mismatch kernel. While the number of these kernels increases, their routing complexity remains relatively flat.

In contrast, the routing complexity of Levenshtein automata is highly sensitive to changes in the score threshold. This is due to the linear scaling of fan-in/fan-out to account for a number of deletions up to the score threshold. Figure 5.7 shows that a Levenshtein automaton with length 24 and score threshold 5 takes about 4 times more routing resources than a Levenshtein automaton with a score threshold of 3, and 10 times more routing resources than a Hamming automaton with a score threshold of 5 and encoded string length of 24. Levenshtein automata with a score threshold of 5, and length greater than 24 fail to route on the current generation of the AP architecture and place-and-route tools.

5.8 Cross-Architecture Application Evaluation

We evaluate the performance of each baseline NFA automata-processing engine over all standard-candle ANMLZoo benchmarks. Results are shown in Figure 5.8. While this does not represent an absolute ranking of the performance of each architecture, it does represent the current state of the baseline evaluation engines included in ANMLZoo as compared to the AP. We present the estimated performance of the first generation

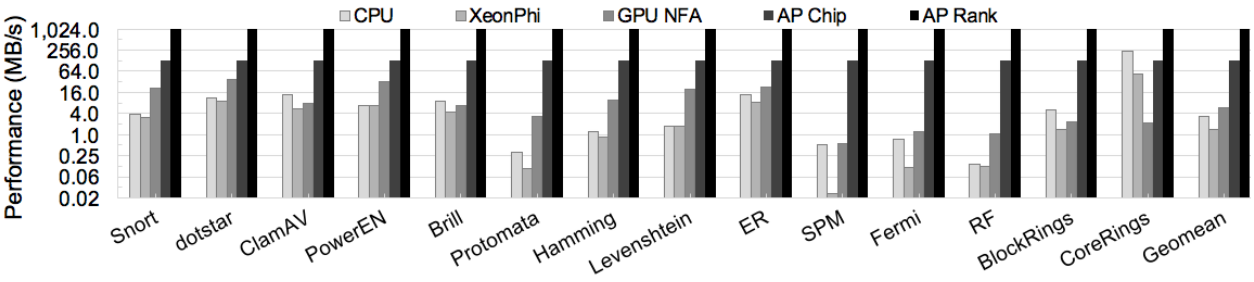

Figure 5.8: Performance of all standard candle benchmarks on each available architecture. AP performance is estimated to be 133MB/s, however, we expect to see performance degradations due to output reporting constraints when using the real hardware. Because each ANMLZoo *standard candle* automata maxes out an AP chip, it is easy and fair to estimate the performance of an AP Rank (8 chips) as 8 times the performance of an individual AP chip.

AP hardware [34].

VASim tends to perform worse on the XeonPhi than the i7 CPU. While the XeonPhi has many more individual cores (57 rather than 6), the CPU's large L3 cache is more important than parallel cores for accelerating the VASim algorithm. A XeonPhi-specific baseline automata-engine to take advantage of its vector units, similar to iNFAnt2, is therefore desirable for a more fair evaluation of these architectures. The XeonPhi performs better than the CPU on the PowerEN and Levenshtein benchmarks, indicating that the VASim algorithm is more bottlenecked by parallelism, and less bottlenecked by rule-transition latency, for these benchmarks.

The GPU NFA engine performs better than our baseline CPU engine in 10 out of 14 benchmarks, indicating that the GPU's massively parallel resources are important for parallel automata-processing. However, the CPU engine outperforms the GPU engine on Brill, ClamAV, BlockRings, and CoreRings. This is most likely due to both a small active set and visited set in these applications, allowing for more ideal cache behavior on the CPU.

The simulated AP's chip's data-flow style architecture generally outperforms all von-Neumann-style NFA automata engines. One notable exception is the synthetic CoreRings benchmark, where VASim is capable of achieving upwards of $230MB/s$. CoreRings has an extremely low activity relative to the number of states. Section 5.4 showed that the CPU can achieve $50MB/s$ per core per active state. Thus, the impressive performance for the full multi-threaded version of CoreRings is unsurprising. This motivates heterogeneous automata-processing engines and architectures with both von Neumann and data-flow engines operating on portions of automata that best suit them.

Because each standard-candle automata benchmark maximizes the resources of an individual AP chip, we can easily, and fairly, estimate performance of other deployment scenarios of AP chips. Because each AP chip can operate on a separate, parallel portion of the input stream, AP performance is expected to scale

perfectly linearly in the real hardware. As an example of this feature of the benchmark suite, we also include estimated AP Rank performance in Figure 5.8.

For each architecture, we explored both dimensions of automata parallelism to attempt find the best-performing configuration. However, exploring every possible configuration was not feasible and so optimal performance is not guaranteed. Data-flow architectures such as the AP do not have this dimension of complexity, and thus guarantee deterministic performance with *no* dynamic performance tuning. This property is extremely desirable for real-time applications such as deep-packet inspection and on-line machine learning.

Moving forward, new algorithms, automata-representations, automata-processing engines and new automata-processing architectures can be easily evaluated and compared using ANMLZoo. We encourage researchers to contribute any of these components as they are developed, so that new research can fairly and easily compare to prior work.

5.9 Towards ANMLZoo 2.0: A Retrospective and Future Benchmarking Template

ANMLZoo was released in the Summer of 2016. Since its release, it has been used by researchers to benchmark automata processing approaches in major architecture [81, 82, 83], and FPGA [53, 88] conferences. While ANMLZoo has had immediate, high impact in the automata processing domain, since its publication, some valid criticisms have arisen. This section highlights valid criticisms of the ANMLZoo benchmarks, and then suggests modifications to the benchmark suite, and a new automata processing benchmarking methodology.

5.9.1 Critiques of ANMLZoo Benchmarking Methodology

Below is a list of common critiques that have been levied at ANMLZoo, and responses:

Automata are too small [89]: Nourian et al. claim that automata in the ANMLZoo benchmark suite are "not meant for large scale analysis."

Response: It is unclear what the authors meant by this, and they did not elaborate. We interpret this criticism to mean that automata are "too small." ANMLZoo applications are standardized to the Micron D480. Some important, real-world automata applications were actually not big enough to fit into one AP chip. Some applications (especially in the domain of bio-informatics [11]) can be orders of magnitudes larger than the capacity of an AP chip, and have complex routing that causes high

utilization, or unroutability. Nourian et al. [89] picked small, unrealistic bio-informatics automata structures but replicated them many times in order to justify a "large" benchmark. It is thus unclear how these automata represent a better or fairer evaluation than ANMLZoo benchmarks and why they chose not to also evaluate each system's performance with the ANMLZoo automata.

Documentation is lacking: This criticism has been levied by many users. Users find it frustrating to have to read cited papers in order to have an intuition behind how the automata work. In many cases there are tens of hours involved in understanding each individual benchmark, and how it computes the underlying function.

Response: This criticism is valid. We have even seen examples of researchers mis-using the benchmarks due to misunderstandings, applying optimizations and claiming speedups that are spurious [81]. These misunderstandings could have been avoided if documentation of both automata graphs and input streams was available or clearer. In response, we believe every benchmark should have an easy to understand landing page that describes in an intuitive way, with visuals, how the automata process the application's input streams. We have made an effort to improve documentation. In particular, Hamming, Levenshtein, RandomForest, EntityResolution, and SPM have updated documentation [90] meant to help teach users the intuition behind these automata, without requiring researchers to read the original papers.

Inputs are too small (size)/not enough inputs (number)/not diverse enough (source): Many users and peer reviewers observe that different inputs may cause different behavior in automata, and change the conclusions of evaluations that use ANMLZoo. The availability of only one, 10MB input per benchmark also limits the ability of novel techniques to properly train/test models.

Response (size): We semi-arbitrarily chose inputs of 10MB. We thought that 10MB would give users enough data to assess streaming performance of automata applications, offer enough of a snapshot of an application to capture diverse behavior, and also be small enough to distribute efficiently. For some benchmarks 10MB is much less than the inputs used in the real world. For example, bio-informatics alignment (Hamming, Levenshtein) often deals with analysis of the human genome, which is a few gigabytes of data. In hindsight, we have no reason to believe that 10MB inputs caused issues with conclusions from studies that used the benchmarks. However, we do agree that the ability to generate longer inputs, to capture the overheads of streaming computation, or to measure end-to-end performance on real-world, large scale evaluations would be useful. This motivates the inclusion of scripts to generate

arbitrarily long inputs, or pointers to repositories that include larger, more realistic, but still compatible data sets.

We also recognize that it is not necessary to fix the evaluation input byte streams to a particular size for all benchmarks. Future benchmarks should include input sizes that capture and reflect real-world behavior of the application, and perform a full instance of the measured algorithm. There does not seem to be any worthwhile benefit to fixing the input size across all benchmarks.

Response (number): ANMLZoo chose to couple single evaluation inputs with single instances of automata graphs in order to constitute one benchmark. This single data point might miss behavior caused by malicious inputs, or not represent average-case behavior.

In hindsight, we have no reason to believe that the inputs we chose did not represent average-case behavior, or caused issues with conclusions drawn from the benchmarks in papers. However, we do agree that more inputs, and inputs that intentionally cause certain behaviors, will only add to the usefulness of the benchmark suite. In particular, having multiple inputs for training and testing of statistical techniques would be especially helpful. We therefore suggest that future benchmarks include one "standard" input for performance evaluations and testing, and at least nine other "training" or other evaluation inputs of the same size.

In the design of ANMLZoo, we chose inputs that seemed representative of average-case behavior. In situations where input stream choice seemed genuinely arbitrary, we decided to make the input source choice arbitrarily. In hindsight, we have no reason to believe that this caused issues with conclusions using the benchmarks. However, we do agree that more inputs, and inputs that intentionally cause certain behaviors, will only add to the usefulness of the benchmark suite. We therefore suggest that future benchmarks include larger and more

Response (source): We think that evaluated inputs should represent average-case behavior. However, average-case behavior might not capture worst-case behavior that are important to design around for some use-cases. For example, in network intrusion detection (Snort), systems are designed with malicious attackers in mind. Thus corner case behavior that causes poor performance in a system is a likely attack vector. These corner case stimuli are less useful for architects, who search for broad trends in performance, however, we do agree that this behavior is important for some application domains, and should be included for application specific evaluations of systems.

Benchmarks are not full applications: In order to spatially normalize automata, some applications were artificially cut down, or enhanced. This makes apples-to-apples comparisons of the performance of the automata benchmarks to other, non-automata-based algorithms on other architectures impossible.

Response: This is a valid, important criticism that should be addressed when designing future benchmarks. We argue that cut-down applications are acceptable benchmarks when comparing automata processing performance across different execution engines. However, automata-processing may not be the most efficient algorithm for computing a particular application on a given architecture [76]. Thus, cut down benchmarks are not useful for benchmarking kernel performance of a particular architecture. For example, we cannot say that a GPU performs worse than an FPGA for Random Forest inference given an automata benchmark that does not capture an entire instance of a computation. We can only say that the GPU performs worse at Random Forest inference *when using the automata processing algorithm to compute the kernel*. Thus, benchmarks should strive to be comparable to other algorithms implementing the same kernel where possible.

Benchmarks pegged to unavailable/obsolete processor: The Micron D480 will most likely never come to market, its tools are difficult to access, and FPGAs have "leap-frogged" the AP in performance and capacity. Why peg benchmarks to an obsolete processor?

Response: When ANMLZoo was conceived, Micron's Automata Processor was still expected to reach commercial availability. It was also still considered the state-of-the-art as a real-hardware automata processing accelerator. Since the inception of the AP, commercial FPGAs benefited greatly from process manufacturing advances. Recent work has shown that FPGAs have a much larger capacity than the D480 AP, and potentially higher operating frequencies [53], and more flexible routing resources [37]. Thus, it is no longer prudent to consider the AP as the state-of-the-art performance and capacity target, and we do not see a reason to continue to spatially normalize applications to this architecture.

However, when designing new benchmarks, we should still develop reasonable justifications for regularizing automata when the choice of size and number of automata is arbitrary. Arbitrary choices should be avoided when possible.

The above criticisms, filed after years of use and experience were instrumental in helping us understand drawbacks to current approaches and to define new goals for good automata processing benchmarks. Below we summarize the lessons learned and properties that we think benchmarks should require before addition to a future benchmark suite.

Full Applications: Where possible, automata benchmarks should compute a full, useful, end-to-end kernel computation. In this way, automata processing on various accelerators can be fairly compared to the best known algorithms implementing the same kernel on other architectures.

Full documentation: Each application should be documented in a concise, intuitive, easy to understand manner. This is so researchers can understand how inputs are co-designed with the automata so that they properly understand acceleration opportunities.

Avoiding purely arbitrary design choices: The choice of size and number of some automata benchmarks can sometimes seem completely arbitrary. For example, mesh automata used for string scoring (Hamming, Levenshtein) have three arbitrary dimensions of design: 1) the length of the encoded string, 2) the hamming or edit-distance the widget calculates, and 3) the number of widgets in the benchmark. Arbitrary choices should be avoided where possible when designing benchmarks with many design parameters.

Realistic, diverse inputs: Inputs should not be arbitrarily sized, and should represent the actual inputs of an automata application. There should be a single, default testing input that represents "average-case" behavior. There should also be at least nine other accompanying training inputs of the same general behavior. If the application is motivated by possibly malicious inputs, a separate set of inputs should be generated and combined with the automata to form a similar, but distinct benchmark. Where possible, scripts to generate inputs should be given.

In the next section, we use the above guidelines to build a new benchmark for automata processing–Snort network intrusion detection rules–as a possible member of a next-generation benchmark suite.

5.9.2 Snort Network Intrusion Detection Benchmark

Snort is an open-source set of network packet inspection tools and is an especially important motivator for regular expression and finite automata processing acceleration. Due to increasing network traffic and increasing numbers of packet inspection rules over time [91], the processing requirements for network analysis systems have also increased.

The original ANMLZoo benchmark suite included a Snort benchmark [22]. To generate the benchmark, regular expressions were extracted from the *pcre* tags of snort rules. PCRE regular expressions can have *modifiers* that guide how the regular expression should be compiled or guide what part of the input stream the regular expression should be applied to. For ANMLZoo, we stripped Snort-specific modifiers [92] and compiled regular expression patters without regards to the semantics of these modifiers.

Each rule was then added to a common rule-set and then the rule set was compiled to automata using Micron's regular expression to automata compiler [55]. If the ruleset failed to compile, the rule is skipped, and the process continues. As discussed in previous sections, the benchmark is complete when the compiler indicates that any additional rules would require two AP chips to implement [22].

After the release of the benchmark suite, a number of issues were identified with this methodology. The first issue was the closed source nature of the Micron PCRE compiler [55]. Some rules were not able to be compiled for various reasons, and could not be diagnosed due to the closed-source nature of the tool. Thus, these rules could not be considered in the benchmark. In contrast, open-source PCRE compilers, such as Intel's Hyperscan [43], offer a much more robust compiler framework, that can consider more rules.

The second issue was that we were ignoring the semantics of the Snort-specific modifiers [92]. Some of these modifiers guide the Snort system to search for regular expression patterns in a particular section of a packet payload, rather than the entire packet. Thus, our methodology of ignoring these modifiers, compiling the rules, and running these automata on the whole packet stream, produced spurious reports, and an extremely high matching rate.

To solve the above problems, we make two changes to our methodology for converting regular expressions rules to automata benchmarks:

1. We use Hyperscan [43], a publicly available, open-source regular expression processing library to convert regular expressions to homogeneous finite automata format. This tool is published as *hscompile*, as a part of the MNCaRT automata processing ecosystem [79] developed at the University of Virginia. Hyperscan is an industry standard regular expression engine and offers users a high-level of confidence that regular expressions are being compiled and converted correctly.

2. We segregate rules with Snort specific modifiers. Instead of removing the modifiers and considering PCREs that are context specific, we instead only consider PCREs from Snort rules that do not have Snort specific modifiers, and ignore rules with Snort specific modifiers. In the future, rules with these modifiers could be segregated into groups, and form distinct benchmarks, with inputs that correspond to their expected use-case.

3. We consider all PCRE regular expressions from all community Snort rules. Previously, we stopped adding new rules when the Micron AP chip was full. Our new methodology considers all rules.

The result of this new methodology is a new Snort benchmark that behaves in a way that is much closer to the actual application, and larger in size due to the better support in the open-source compiler,

	States	Compressed States	Compression Factor	Average Activity	Reports/symbol
ANMLZoo Snort	69,029	34,366	50.7%	30.13	2.29
New Snort	258,750	216,315	16.5%	432.55	0.132
Change	3.75x	6.3x	-3.07x	14.36x	-17.34x

Table 5.2: Comparison of static and dynamic metrics of the ANMLZoo Snort benchmark and the new Snort benchmark built with the new methodology. The new benchmark has more states, more activity, is less compressable, and has many fewer reports.

and consideration of many more PCRE rules. Table 5.2 shows summary statistics of this new automata benchmark.

The first major difference is the size of the benchmark. The new Snort benchmark has 3.75x the number of total states, and 6.29x the total number of non-redundant states. Interestingly, the old Snort benchmark was compiled from 2,585 PCRE rules, while the new Snort benchmark was compiled from 2,670 PCRE rules. This large increase in states corresponding to a small increase in total rules means 1) that larger rules were added to the benchmark and 2) more unique rules were added to the benchmark.

The second major difference is the activity. We profiled the original Snort benchmark and the new Snort benchmark using ANMLZoo's 1MB inputs and measured the average activity in the automata. Table 5.2 shows the results of our experiment. The original benchmark has, on average, 30.13 states compute on any given cycle. In contrast, the new Snort benchmark has 432.55 states compute on any given cycle, a 14.4x increase. This is a substantial increase in activity, and further motivates spatial approaches for this application to handle this increase in activity.

The third major difference is reporting behavior. The original Snort application reports on average 2.29 times for every input symbol. This frequent reporting behavior is caused by indiscriminately applying context sensitive rules to the entire input stream as discussed above. The new snort benchmark reports far less–on average only once every 7.6 input symbols. Note that these results differ from those in Chapter 7 due to bug-fixes to VASim during the period between the evaluations.

5.10 Conclusions and Future Work

This chapter presented ANMLZoo, a diverse benchmark suite of finite automata for easy and fair evaluation of automata processing engines. ANMLZoo benchmarks are quantitatively diverse in both static structure and dynamic behavior and represent a wide range of well known and new applications for automata processing.

Using ANMLZoo, we were able to show bottlenecks in von Neumann computer architectures for automata processing. CPUs perform well when the average activity in an automata is small, and the average number of visited automata states is small. GPUs and Intel's XeonPhi can perform well, but exploiting the computational

power of SIMD units to compute the irregular parallelism of automata is difficult. Thus, these architectures generally benefit from input stream parallelism, rather than automata-level parallelism. The AP is the fastest automata-processing hardware but its capacity is very sensitive to automata topography and cannot place-and-route automata states with complex connectivity.

In retrospect, after years of use and experience, we outline guidelines to build future benchmarks that better serve the research community. These benchmarks should represent full applications, be well documented, avoid arbitrary design, and include input stimuli that correspond to real problems. As a step towards this goal, we present a new Snort benchmark that more closely represents the Snort application computation requirements. We suggest that future work should apply this methodology to build a next generation benchmark suite for automata processing.

Chapter 6

Automata-to-Routing: An Open-Source Toolchain for Design-Space Exploration of Spatial Automata Processing Architectures

6.1 Introduction

Prior spatial automata acceleration has investigated using FPGA fabrics to place-and-route automata states [28, 29, 30]. New commercially available hardware such as Micron's Automata Processor [34] uses a finite-automata-specific reconfigurable fabric to improve state density over techniques on more general-purpose fabrics. In order to achieve the full potential of automata processing, it is necessary to further research automata-specific spatial architectures that can identify tradeoffs among different design decisions. However, of the new automata processing architectures published since Micron published details of the Automata Processor [51, 50, 49], all are either von Neumann-based techniques or implemented in existing FPGA fabrics.

The reason for this lack of spatial architecture research is three-fold:

1) A benchmark suite of diverse, real-world automata processing benchmarks was not available for fair apples-to-apples evaluations of automata processing architectures.

2) Suitable open-source tools for the optimization of automata (tantamount to good performance on both von Neumann and spatial architectures) did not exist. Thus, important standard optimizations must be

reproduced, making research in automata processing high effort.

3) Furthermore, no open-source tools for spatial automata architecture research (including high-quality place-and-route algorithms and flexible, parametric automata-processing fabric description languages) existed for fair evaluation of spatial automata-processing architecture design choices.

To solve the above problems, this chapter presents a new open-source toolchain–Automata-to-Routing (ATR)–that can place-and-route any AP application on a parameterizable spatial automata processing architecture. ATR builds upon three open-source tools: a newly available automata processing benchmark suite, ANMLZoo [22], an open-source framework for automata optimization and transformation, VASim [78], and a well-known existing FPGA architecture research framework, VPR [38]. Integration of these tools was not straightforward, and required additional capabilities to enforce spatial architecture design rules on abstract automata, and emit automata in a VPR-readable format. The ATR toolchain enables research on novel spatial automata processing architectures, and can be used to evaluate design choices in existing, commercially available processors.

To show the usefulness of the ATR toolchain we use it to explore bottlenecks in a commercially available automata processor. We first create a baseline model of Micron's Automata Processor (AP) [34] to see if ATR is capable of faithfully modeling the AP's logic-tile architecture. We compare placement performance of this baseline AP model architecture to the real AP, using the ANMLZoo benchmark suite. In many cases, the ATR toolchain can closely model the AP's logic-tile architecture, matching placement statistics to within an average of 7.9% for 9/14 benchmarks. This result indicates that we are able to accurately model the AP's tile architecture for most applications, and motivates future high-level design-space exploration of automata-processing logic-tile architectures using the toolchain.

ATR cannot closely model five benchmarks. We demonstrate that this is due to differences between the AP and ATR's routing matrix designs. Micron's AP uses a deep hierarchical routing matrix, while VPR is only able to model 2D-mesh routing fabrics. Our results indicate that ATR's shallower, 2D-mesh routing fabric–in contrast to the AP's deep hierarchical fabric–reduces resource requirements by up to $4.2x$ for difficult-to-route benchmarks. We then identify two properties of automata that cause inefficient resource usage on the AP: average automaton subgraph size and average fan-out. We show that highly connected automata that span a significant portion of the chip are difficult to place-and-route efficiently in the deep hierarchical design of the AP. This result provides application specific insight into how future AP architectures should be designed.

6.2 Automata-to-Routing Toolchain

This section first describes the three tools used to enable Automata-to-Routing: ANMLZoo, VASim, and VPR. We then describe the ATR architecture and how it enables spatial automata processing architecture research.

6.2.1 ANMLZoo Automata Benchmark Suite

ANMLZoo [22] is a public repository of 14 automata processing benchmarks and corresponding input stimuli. Twelve benchmarks are from real-world use-cases for automata processing and two are synthetic. Each benchmark is roughly classified into a "family" of automata based on the topology and size of disjoint automata subgraphs. The first family are regular expression automata (Dotstar, Brill, PowerEN, ClamAV, Protomata). Regular expressions tend to correspond to long, narrow automata, with low topological complexity. "Mesh" automata (Hamming, Levenshtein) have regular, 2-dimensional properties and can grow quadratically with problem size. "Widget" automata (EntityResolution, RandomForest, SPM) are generally composed of smaller custom automaton engines with complex topology that can vary in individual size and complexity depending on application and problem size. Synthetic automata (BlockRings, CoreRings) are designed to test particular properties of execution engines while controlling for automata properties such as active set and visited set.

Because both the number of automata states and routing complexity affect fabric utilization, there is no "right" way to standardize automata benchmarks. Instead of picking one metric, each ANMLZoo benchmark is compiled to completely fill the resources of an AP chip, thus standardizing for both state and routing resources at the same time. This methodology allows easy and fair comparisons of the capacity and routing capabilities of the AP versus other spatial architectures.

6.2.2 VASim Virtual Automata Simulator

VASim [78] is an open-source platform for manipulation, optimization, and simulation of finite automata. VASim offers an easy-to-use, object oriented view of automata-processing directed graphs and is designed from the ground up to be an easy to understand research platform for automata processing application and architecture research.

VASim offers traditional, well known automata minimization passes, e.g. prefix merging. Prefix merging is an algorithm, analogous to circuit minimization, that can greatly improve processing performance in von Neumann automata processing engines, and reduce resource requirements for automata implemented on spatial architectures [22]. VASim allows automata researchers access to these standard algorithms without the need for lengthy re-implementation. Furthermore, VASim also acts as an open-source repository for new

automata optimization algorithms so that researchers can easily share their optimizations with others and use them for both application and architecture research.

VASim can also enforce architecture specific design rules. While von Neumann automata processing architectures generally do not place any restrictions on the number of states, fan-in, and fan-out allowed in an automata graph, spatial architectures are usually very sensitive to these parameters. Just as in FPGA place-and-route, a large fan-in or fan-out in an automaton might prevent a graph from being routable. For ATR, we extend VASim to automatically enforce fan-in requirements of spatial architectures.

6.2.3 Versitile Place and Route

Versatile Place and Route (VPR) [38] is a well known open-source tool for research on tiled, "island style" spatial architectures and is widely used in the FPGA design community. VPR allows researchers to define their own spatial fabrics using an XML-based parameterizable architecture description language. The language describes fabric dimensions (number of logic tiles wide and tall), switch-block architecture, connection-block architecture, channel width, and logic-tile architecture. Logic tiles can be hierarchical, and may be composed of smaller elements such as basic LUTs and registers connected using a basic, but fairly flexible interconnect library.

Once an architecture description file has been defined, VPR takes a Berkeley logic interconnect format (.blif) logic circuit netlist as input. VPR packs each logic tile with circuit elements, places and optimizes placement of tiles in the architecture, and then routes each tile within the reconfigurable fabric.

While VPR is geared towards placement and routing of gate-level logic, VPR also allows place-and-route of "black box" elements, to represent hard logic such as multipliers that may exist as non-LUT computation blocks in the FPGA fabric. VPR is so general, that it allows the definition of *any arbitrary* black box element, not just traditional FPGA logic elements. Thus, in ATR, we use VPR to model reconfigurable arrays of automata state transition elements, rather than traditional logic gates. This new use-case for VPR highlights its flexibility as a back-end architecture description and place-and-route tool for *any* application domain targeting an island style reconfigurable array of processors, as long as a benchmark suite and optimization and transformation tools exist to support this domain.

6.2.4 ATR Toolchain Architecture

Automata-to-routing (ATR) combines ANMLZoo [22], VASim [78], and VPR [38] into a new toolchain for spatial automata processing research. Figure 6.1 describes the high-level architecture of the ATR toolchain. First, automata are fed into VASim. VASim is responsible for optimizing automata (e.g. applying prefix

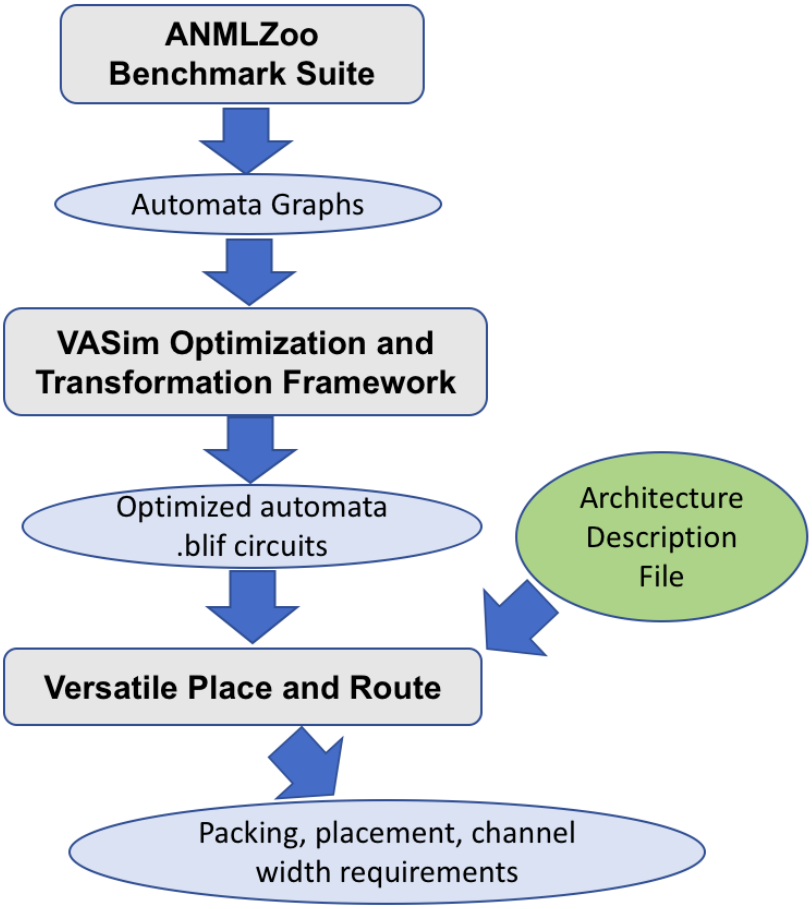

Figure 6.1: The Automata-to-Routing or ATR toolchain flow. ANMLZoo applications are used to evaluate automata architectures. These automata graphs are fed to VASim which parses and optimizes the automata. VASim can also enforce design rules on automata and automatically transform them to fit an architecture without changing the semantics of the automata. VASim emits these automata graphs as .blif circuit files for corresponding automata processing spatial architecture models. VPR takes an architecture description and places-and-routes circuits in this hypothetical architecture.

merging) and applying transformations to enforce user-specified design rules. One particular transformation required for this work, fan-in relaxation, is described below. Once automata have been optimized and transformed to fit the design rules of a particular architecture, VASim emits automata netlist in a VPR readable .blif format. ATR then feeds the .blif circuit and spatial automata processor description file defined by the researcher to VPR. VPR then packs, places, and routes the automata circuit into the architecture.

Results from place-and-route can then be used to evaluate the "goodness" of a given architecture depending on desired properties. ANMLZoo provides a large set of diverse benchmarks to fairly evaluate the relative merits of hypothetical and real automata processing architectures.

Together, ANMLZoo, VASim, and VPR form Automata-to-Routing, a powerful spatial architecture research toolchain. The following sections describe modifications to the open-source VASim tool to enable

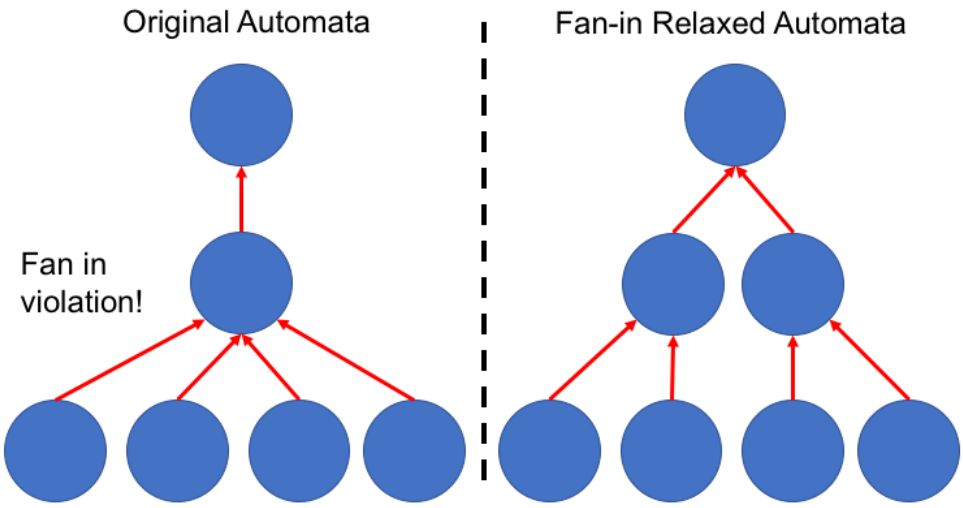

Figure 6.2: Fan-in relaxation example. The maximum fan-in is reduced from 4 to 2 by duplicating a state.

this toolchain and results and conclusions from placing and routing ANMLZoo benchmarks on both real and ATR modeled spatial automata processing architectures.

6.3 VASim Extensions

6.3.1 Design Rule Transformation: Fan-in Relaxation

We add one new transformation to VASim to better allow it to serve spatial-architecture researchers: fan-in relaxation. While abstract automata states can have arbitrarily large fan-in, spatially-routed automata can have fan-in restrictions based on the underlying spatial architecture's routing matrix and tile architecture. Fan-in relaxation duplicates a state that has a fan-in that violates a maximum defined by the architect.

When duplicating a state, inputs to the original state are divided among the new duplicate states, while output edges are copied. A picture illustrating a simple example of input duplication is show in Figure 6.2. If the fan-in was N before relaxation, fan-in is guaranteed to be at most $ceil(N/2)$ after relaxation. Note that this technique doubles the fan-in of the child states of the duplicated state. This algorithm proceeds in a breadth-first manner to ensure that the required fan-in is enforced for all nodes in a single pass.

Three applications from the ANMLZoo automata benchmark suite have high fan-ins: ClamAV, EntityResolution, and Snort. All three required the fan-in relaxation transformation before successful place-and-route on the architectures modelled below.

6.3.2 Design Rule Transformation: Group-of-Two Grouping

In order to save logic and routing resources on-chip, the AP D480 couples every two STEs into a *Group-of-Two* or GoT. A Group-of-Two allows configurable connectivity between a pair of STEs without consuming global routing resources [2]. A figure describing the GoT functionality is shown in Figure 2.4. VPR is currently unable to model the logical OR of signals, and so, we implement a pass in VASim to identify pairs of STEs that might use this OR-gate and pre-group them into a GoT element.

6.3.3 .blif Emission Algorithm

VASim is also extended to convert abstract automata to .blif circuit files readable by VPR. .blif files have three distinct sections that define a circuit. The first section defines the top level input/output signals. The second section describes all instantiations of circuit elements. The third section describes module definitions of each circuit element.

We guide VASim to emit a single clock input pin and no output pins. Because I/O is handled by specialized IP blocks in the AP and not by individual pins in the chip, we essentially ignore I/O pins and turn off dangling block removal in VPR. Output reporting signals for STEs are left unconnected to better match the AP's statically routed reporting architecture [1].

We then guide VASim to emit a circuit element for each STE and pre-grouped GoT in the automata graph. Each STE and GoT lists its input-enable ports, assigning the wire names of each STE that connects to it. It also lists a single output wire that can connect to other STEs in the netlist. We finally guide VASim to emit circuit element module definitions for STEs and GoTs. While this methodology is specific to Micron's AP, ATR is flexible to support the addition of any arbitrary processing elements and I/O architectures. Thus, allowing researchers to investigate the impact of new, hypothetical automata-processing elements.

6.4 Modelling Micron's Automata Processor

In order to demonstrate the usefulness of ATR, we first attempt to model Micron's Automata Processor in the VPR architecture description language as accurately as possible. By attempting to accurately model the AP, we can create a *baseline architecture* from which we can evaluate the advantages, disadvantages, and trade-offs of potential architectural changes. Thus, researchers will be able to explore new spatial architecture design parameters (such tile complexity or channel width), new hardware features, or entirely new spatial architectures for automata processing. Previously, no such capability was available to researchers, inhibiting advances in this area.

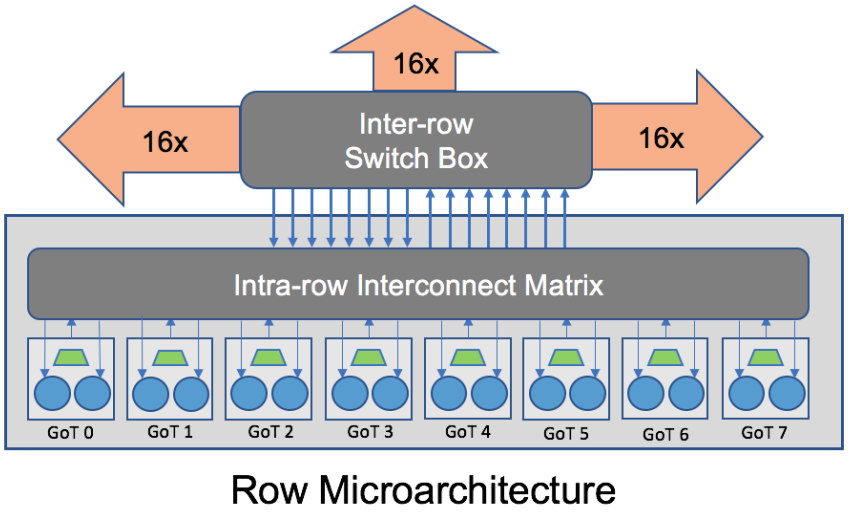

Figure 6.3: Model-AP routing architecture configuration with channel width of 16, and Row architecture with 8 Groups-of-Two (GoT). Each GoT has two inputs, but selects a single output between either STE or the OR or their outputs as detailed in Figure 2.4.

6.4.1 Defining A Baseline Tile Architecture

Because the AP's routing matrix is hierarchical in nature, and the VPR tool assumes a 2D mesh-style routing fabric, the AP architecture cannot be modeled 100% faithfully. We must choose a layer of the AP's hierarchy (described in Figure 2.5) to expose to VPR's 2D routing fabric as the base level tile to model. We choose the AP Row as the tile abstraction for our baseline AP model. The AP Row is the first AP structure to directly connect the global reconfigurable routing network [34] and thus is a natural (although admittedly imprecise) structure to implement as the base tile.

6.4.2 Defining A Baseline Routing Network

Once the baseline tile abstraction has been defined, we can design our baseline routing architecture to model the corresponding level of the AP's routing network.

Detailed architectural descriptions of the AP's routing network have not been made publicly available. However, patents filed by Micron that correspond to the AP give us a high-level idea of how tiles might connect to the routing matrix [2]. Figure 6.3 describes the assumed routing matrix organization of the AP. This organization is presumptive and not authoritative. The rest of this chapter refers to the presumptive modeled architecture as the "model AP."

Each model-AP Row has 8 input and 8 output connections to the global routing matrix. GoTs within a Row can receive input from any of these input wires via the intra-row interconnect matrix, and can send output to a single output wire. GoTs can choose output from either internal STE, or the OR of their output.

VPR is unable to model the GoT's OR gate functionality, and so these structures must be identified using VASim's GoT identification pass (discussed in Section 6.3.2) prior to VPR packing and placement. Final states report activations via two reporting ports statically placed in each row structure.

The model-AP Row routing switch block is 16 tracks wide, suggesting a channel width of 16. Because channel width and wire segment length are co-designed and unknown for the real AP hardware, we pick a segment length of one. We leave refinement of these parameters for future work.

6.5 Place-and-Route Results

We use the ATR toolchain to place-and-route each ANMLZoo automata benchmark on our baseline representative AP architecture. Three ANMLZoo benchmarks (ClamAV, EntityResolution, and Snort) were transformed using fan-in relaxation to have maximum fan-ins of 8. We measure and report the number of rows required to successfully pack, place, and route each benchmark. VPR was configured to prevent unrelated clustering. Unrelated clustering allows automata states from different disjoint subgraphs to be placed into the same tile. While this generally allows for a denser packing, it can greatly increase channel width requirements, and is turned off to satisfy the model AP channel width requirements. Place-and-route results are presented below.

6.5.1 Tile Resource Requirements

Figure 6.4 plots the tiles (AP Rows) required to place and route each ANMLZoo automata benchmark. The first bar represents the performance of ATR with VASim's automata prefix-merging optimizations turned off. The second bar represents the performance of ATR with VASim's prefix-merging optimizations turned on. VASim's optimizations attempt to simulate optimizations performed in Micron's automata compiler, and are thus always considered for fair model comparison. The third bar represents performance of ATR with optimizations and GoT grouping turned on. GoT grouping only makes a difference if automata states benefit from the internal OR gate functionality. Figure 6.4 also plots the number of tiles the Micron AP compiler requires to place-and-route each ANMLZoo benchmark.

Some model AP results closely match the results from Micron's compiler and architecture. For ClamAV, Dotstar, Fermi, Protomata, RandomForest (RF), Snort, and BlockRings, the number of model AP tiles are within $2.1\% - 9.5\%$ of the real AP compiler stack and architecture. These particular application results indicate that we are able to model the AP architecture and compiler stack.

Some applications actually require fewer tile resources than the AP when placed-and-routed in our model. For Brill, EntityResolution (ER), Hamming, Levenshtein, and SPM, the number of model AP tiles is

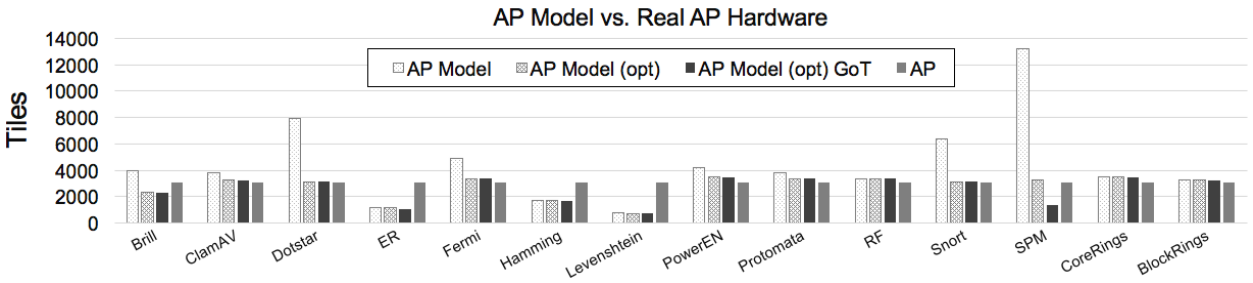

Figure 6.4: Compilation results from our AP model implemented in the ATR toolchain and compiled by Micron's AP compiler for the first generation AP D480 chip. "opt" refers to automata graphs optimized using VASim's prefix-merging optimization. "GoT" refers to graphs with pre-grouped GoTs using VASim's GoT grouping pass. ATR is capable of accurately modeling the resource usage of the AP in many cases. Large deviations are due to limitations of VPR's support for deep hierarchical routing matrices.

considerably lower than the real AP. This indicates that, for these applications, the model AP architecture performs better than the real AP hardware and compiler stack. We discuss possible reasons for improved performance over the AP architecture and suggest improvements to the AP to take advantage of these results in Section 6.6.

6.5.2 Routing Resource Requirements

VPR uses a binary search to identify the smallest number of required tracks per channel to successfully route a circuit in a reasonable amount of time. This metric can be used as a proxy for how difficult the placed circuit was to route. A higher minimum required channel width indicates a more difficult routing problem, and suggests that the architecture requires at least that many routing tracks to successfully implement that circuit. Figure 6.5 shows the minimum channel width requirements derived by VPR for each ANMLZoo benchmark for the model AP.

Most benchmarks can be routed with 6 tracks per channel. However, three applications required 14 tracks per channel. All applications have a minimum channel width requirement less than the model AP's 16-wide channels.

6.6 Evaluating the AP's Routing Matrix Using ATR Modelling

While the Automata Processor is capable of routing many applications in ANMLZoo with the same efficiency as the model AP, the previous section identified five benchmarks (Brill, EntityResolution, Hamming, Levenshtein, and SPM) where the model AP performs much better. In the case of Levenshtein, the model AP is able to route all automaton subgraphs while using $4.2x$ fewer tile resources.

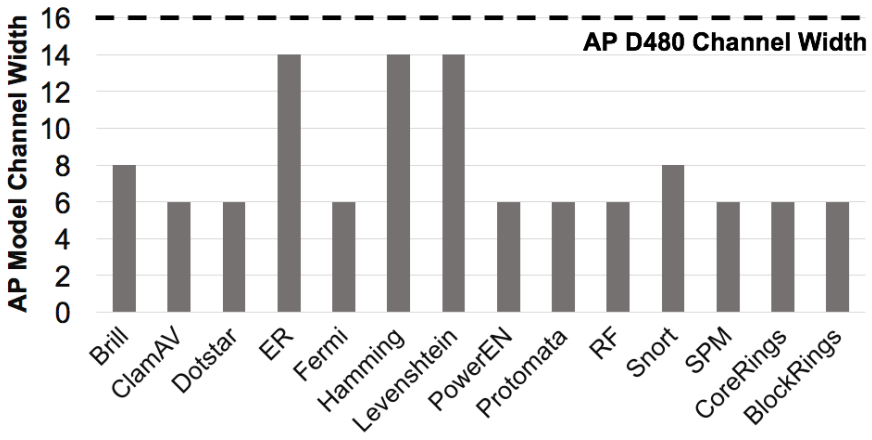

Figure 6.5: Minimum channel-width requirements determined by VPR for each ANMLZoo benchmark for the model AP. All benchmarks are able to be placed-and-routed successfully using less than 16 routing tracks per channel, the maximum channel width of the model AP routing matrix.

We suspect the reason for this improvement is the difference in the routing matrix topology. The AP's hierarchical routing matrix has four levels [34]. In contrast, VPR's underlying routing matrix assumes a 2D-mesh, island-style architecture with only one layer. Because they lack a deep hierarchy, 2D-mesh fabrics are much more flexible in the graph size and graph topologies they can place-and-route efficiently. To highlight this difference between tree-based routing fabrics and mesh-based routing fabrics, we plot each ANMLZoo benchmark as a function of its average automaton subgraph size and the average node fan-out. These results are shown in Figure 6.6.

The darker region encompassing in Figure 6.6 highlights where automata are difficult to route using a hierarchical routing matrix. Importantly, these automata have both high connectivity, and large subgraph sizes.

Brill is a large tree structure with a single, very large automaton subgraph. However, large automaton subgraphs are not enough to cause poor performance in the AP's routing matrix. For instance, CoreRings and RandomForest both have average automata subgraph sizes greater than Brill. The key difference is that Brill has a large enough average fan-out (1.49) that most likely causes congestion in the roots of the AP's hierarchical routing matrix. CoreRings and RandomForest are essentially very large loops, that can be routed fairly easily through the routing hierarchy, even though they are larger than the Brill automaton.

However, some applications like Fermi have relatively similar fan-out (1.48), but route efficiently in the AP. Fermi has relatively complex connectivity, but each automaton subgraph is so small that it is able to fit within just a few tiles. As long as the connections between tiles are relatively few, the AP is able fit these automaton into the leaves of the hierarchical routing matrix without causing congestion in the roots. Fermi automaton only need four automata tiles and thus also represent a much easier global routing problem than

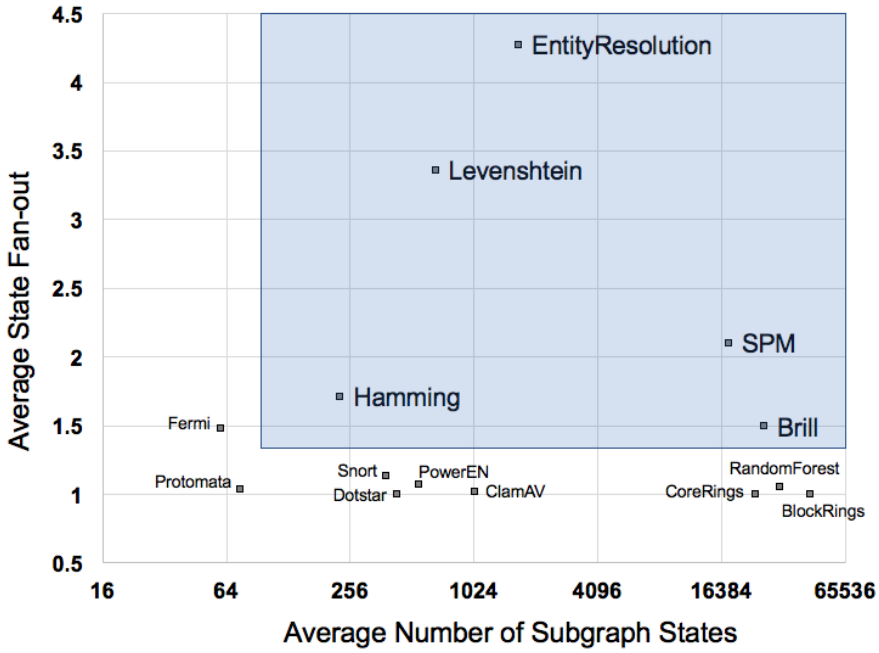

Figure 6.6: Each ANMLZoo benchmark plotted as a function of the average size of each disjoint automaton subgraph, and the average fan-out of each node. The darker region highlights an area where automata are larger, with larger fan-out. EntityResolution, Levenshtein, Hamming, Brill, and SPM are all applications where 2D-mesh, spatial-automata processors perform much better than the 4-layer hierarchical routing matrix of the AP.

Hamming, which requires at least 32.

EntityResolution and Levenshtein represent more challenging cases, where both the average automaton subgraph size and fan-out are higher. EntityResolution and Levenshtein must use $2.7x$, and $4.2x$ more tiles respectively when routed using the hierarchical routing matrix than the model AP's 2D-mesh routing fabric.

EntityResolution, Brill, Levenshtein, and Hamming (the four benchmarks that the tree hierarchy struggles to route efficiently) all have both high-connectivity and relatively large subgraph sizes. EntityResolution is a custom automata that utilizes large, highly-connected trees to match approximate names in databases. Brill is a large, single tree structure, and must be placed-and-routed to span the entire reconfigurable fabric. Both Hamming and Levenshtein are classified as *mesh* automata in ANMLZoo [22]. Mesh automata have a regular, 2D-grid structure, and are much more naturally placed-and-routed in 2D routing matrices.

While the above applications are not easily routed using the AP's hierarchical routing matrix, most ANMLZoo applications, especially those that are regular-expression based, perform very well. Because regular-expression processing is such an important motivation for automata processing acceleration, we conclude that the AP's routing matrix architecture is suitable for regular expression rule-set acceleration in most cases. However, for some applications such as Levenshtein edit-distance automata (an extremely important string comparison kernel for a wide variety of application domains), the AP's routing matrix is not

an ideal design.

6.7 Conclusions and Future Work

This chapter presented the Automata-to-Routing (ATR) toolchain. ATR is a full-stack, automata-processing toolchain that enables research of spatial automata-processing architectures. Prior to ATR, no research toolchain for spatial automata-processing architecture research toolchain existed, preventing exploration of this exciting new class of architectures. ATR combines three different existing open-source software tools–the ANMLZoo [22] benchmark suite, the VASim [78] open finite automata simulator and optimization framework, and VPR [38], a flexible reconfigurable array simulator and place-and-route tool–into one toolchain that allows researchers to experiment with spatial, automata architectures. ATR enables evaluation of new and existing automata processing-tile organizations, and routing-matrix architectures.

We show that ATR is capable of modelling the logic tiles of existing spatial, reconfigurable, automata-processing architectures such as Micron's Automata Processor with sufficient accuracy to enable research on new architectures for spatial automata processing.

We present a study–enabled by ATR–comparing and contrasting performance of the AP's hierarchical routing matrix against a proposed 2D-mesh style routing matrix. We identify four benchmarks where a 2D-mesh routing matrix performs better, using up to $4.2x$ fewer logic tiles. We also characterize two properties of automata graphs–average state fan-out and average number of subgraph states–that correlate with poor routability on the AP.

Future work is needed on further extensions to the ATR tools, design-space exploration of spatial automata processing tile microarchitectures, identifying improved tile microarchitectures. Future work may also include area and power modelling of spatial automata-processing architectures to identify additional trade-offs between logic-tile and routing resources.

Chapter 7

Characterizing and Mitigating Output Bottlenecks in Spatial Automata Processing Architectures

Spatial, reconfigurable architectures, such as field programmable gate arrays (FPGAs), usually offer much improved performance over von Neumann solutions [22]. Spatial architectures consist of reconfigurable networks of processing elements, and implement automata graphs by wiring together processing elements using a place-and-route algorithm. Spatial architectures excel at automata processing, because they allow massively parallel state-matching computation and point-to-point transition rule communication using parallel processing elements [22]. Thus, spatial architectures are *performance inelastic* and have the same performance no matter how many state transitions are computed in parallel.

However, depending on their implementation, spatial architectures can suffer performance degradation due to input and output processing. Because spatial automata processing architectures compute many state transitions in parallel, and may match many patterns within a single cycle, they can generate a massive amount of output, and become bottlenecked by I/O operations. For example, Micron's D480 Automata Processor—an automata-specific spatial architecture that supports up to 6,144 reporting states per chip—must stall for up to 255 cycles if many reports occur on a single symbol cycle [1]; a 255x overhead! Even when applications have more modest reporting (e.g. a single report every 10 cycles) the minimum performance penalty is a 6.5x slowdown over ideal computation with no reporting.

Every spatial automata processor must somehow combine or compress a possibly large number of report

signals. Careful attention is needed to design reporting architectures that can support both a large number of reporting states, and frequent reporting events. Otherwise, reporting overheads will eat into much of the performance benefits of spatial architectures over von Neumann architectures.

This chapter focuses on characterizing and mitigating the output reporting problem for spatial automata processing architectures. To the best of our knowledge, this is the first work to characterize reporting behavior across a wide variety of automata benchmarks and recognize its importance in automata-processing application and architecture design. We first characterize automata-processing output requirements using ANMLZoo, a standardized benchmark suite [22]. We use the VASim virtual automata simulator [78] to track the density and frequency of reporting events inside automata graphs. We find that usually there are only a few automata states that report on any one input symbol (sparse reporting), but that reporting events can occur extremely frequently. In rare cases, many automata states may need to report on the same input symbol (dense reporting), but fairly infrequently. This motivates the design of spatial reporting architectures that are optimized to support efficient handling of a small number of frequent reports at low-cost (the common case), but do not hurt performance when reporting is dense (the uncommon case).

To better understand the overhead associated with output reporting in real spatial architectures, we develop a novel parameterizable cycle-accurate simulator methodology for spatial architectures. We validate this simulator methodology by configuring it to model an example real-world spatial automata processor (the Micron D480 Automata Processor or AP) and show that output reporting overhead in this architecture can be extremely high, causing up to $46x$ performance degradation. This particular reporting architecture is designed to efficiently handle dense reporting, but pays a large penalty when reporting is sparse. Thus, when faced with common-case reporting behavior of automata applications, overheads can be extremely high.

Motivated by this high overhead, we consider methods to increase spatial architecture performance by reducing report-output overhead, while still supporting a large number of reporting states. The first method we consider modifies the automata graph, combining reporting state outputs that can provably be disambiguated when activated. By combining reporting states, we can reduce output port pressure on the reporting architecture. This transformation is purely a software change, and does not require added hardware.

The second method we consider is a new reporting architecture design for automata processing on FPGAs and automata-specific spatial architectures. Using reporting characteristics discovered from application profiling, we design a configurable reporting architecture that increases performance over the Micron D480 AP reporting architecture, while still supporting a large number of generic output ports. This architecture improves performance when reporting is sparse, and performs just as well as the AP when reporting is dense.

This chapter makes the following contributions:

- To the best of our knowledge, the first characterization of automata reporting frequency and density over a large set of diverse automata benchmarks. This study motivates direct changes to existing automata-processing architectures, and influences design of future architectures.

- A novel methodology for cycle-accurate simulation of spatial automata processors, validated against real hardware. The simulator combines public descriptions of the cycle costs of certain operations with placement information from spatial place-and-route tools to generate highly-accurate performance metrics. We simulate a commercial automata processor and identify that automata reporting can cause severe overheads – up to 46x over ideal performance with no reporting.

- A software-based automata transformation to reduce the cost of output reporting on existing spatial architectures. This automata transformation requires no changes to underlying hardware and increases performance by up to 40%.

- A new, configurable reporting architecture design for spatial automata-processing architectures that can be configured for both sparse and dense reporting. When compared to existing reporting architectures tailored for dense reporting, our configurable architecture shows speedups of up to $5.1x$ when reporting is sparse (the common case), and never hurts performance when reporting is dense (the uncommon case).

These studies not only motivate architecture changes in future automata-specific spatial architectures, but also reporting architectures implemented in automata processing engines on general purpose spatial architectures, such as FPGAs. Our simulator is flexible to account for any spatial architecture solution that relies on compressing and buffering output reports. Our work identifies that reporting, which has been ignored thus far, is a first-class design constraint and should be one of the main focus areas of spatial automata processing architecture research.

7.1 Characterizing Automata Reporting Behavior

To understand typical reporting behavior in various automata use-cases, we first profile benchmarks from the ANMLZoo automata benchmark suite [22]. ANMLZoo is a diverse set of finite automata and associated input streams adapted from real-world applications. Characterizing the behavior of these benchmark applications will help motivate architectures that better satisfy real-world requirements. To the best of our knowledge, this is the first study of reporting behavior in a wide range of diverse automata applications.

Table 7.1: Summary statistics for ANMLZoo reporting behavior

Benchmark	Family	Reports	Report Cycles	Reports/Cycle	Reports/RCycle	Max/RCycle	Std.Dev/RCycle	Index of Disp.
Snort	Regex	1,710,495	995,011	1.710495	1.719	6	0.567	0.197
Dotstar	Regex	0	0	0	0	0	0	0
ClamAV	Regex	0	0	0	0	0	0	0
PowerEn	Regex	4,304	4,303	0.004	1.000	2	0.015	0.996
Brill	Regex	429,386	118,005	0.429	3.640	11	1.585	3.900
Protomata	Regex	111,239	105,722	0.111	1.052	4	0.230	0.991
Hamming	Mesh	2	2	2e-06	1.0	1	0	0.999
Levenshtein	Mesh	4	4	4e-06	1.0	1	0	0.999
ER	Widget	37,628	28,612	0.0380	1.315	3	0.523	1.490
SPM	Widget	47,304,453	33,933	47.304	1394.055	1792	283.980	1,404.599
Fermi	Widget	96,127	13,444	0.096	7.150	20	4.503	9.890
RF	Widget	21,310	3,322	0.021	6.415	9	0.710	6.472

7.1.1 Experimental Methodology

We use the Virtual Automata Simulator (VASim) [78] to simulate the 12 non-synthetic ANMLZoo applications on the 1MB ANMLZoo standard inputs. We use non-synthetic applications, because the synthetic benchmarks are designed for micro-benchmarking von Neumann automata processors and do not attempt to give insight into real-world application behavior.

Before we simulate automata, we first run VASim's standard redundancy-elimination optimization passes. In some instances, automata benchmarks have fully redundant automata that can be identified and merged. Usually, VASim preserves redundant reporting states so that the functionality of the automaton is indistinguishable from the original graph. For this study, we modify the standard VASim redundancy-elimination pass so that these automata and their reporting states are fully merged. However, we map a single report state to multiple virtual state IDs. Thus, automata functionality is not affected, but reporting overheads are reduced. This mimics the behavior of the Micron AP compiler [55].

We then simulate the automata on the standard 1MB ANMLZoo inputs provided with the benchmark suite and track every report over the course of automata execution. The total number of reports (Reports) is distinguished from the total number of cycles in which any number of reports occurred (Report Cycles or RCycles). Because we used ANMLZoo's 1MB inputs, the total number of cycles for the entire application is 1,000,000, as we assume 1 symbol is processed per cycle in spatial systems. Because there are 12 applications, and report traces are large, we present varying summary statistics. These summary statistics will guide our bottleneck analysis and motivate efficient reporting architecture designs. Results are shown in Table 7.1.

7.1.2 Profiling Results

Reporting behavior in the ANMLZoo benchmark suite varies highly from application to application. Some applications do not report at all (ClamAV, Dotstar). While it might seem strange for a benchmark to not use all of its states, this is not bad behavior. ClamAV is a set of virus scanning signatures. As input,

ANMLZoo chose a semi-arbitrary file to represent common case input. This happens to not be a virus, and so no reports should be expected. Some applications, such as Hamming and Levenshtein, report very infrequently. Hamming and Levenshtein automata identify strings that approximately match encoded strings in the automata. Their input was generated randomly, and only very few strings within the scoring metrics were identified. While these particular workloads do not bottleneck spatial processors, this does not mean that these applications could not be bottlenecked by reporting using different automata and/or input streams.

Eight applications (Snort, PowerEN, Brill, Protomata, ER, SPM, Fermi, RF) report more than a trivial amount. Reporting behavior in these applications is highly varying. Some applications report on almost every cycle (Snort), while others report with varying frequency up to about once every 300 cycles (RF).

Furthermore, when applications report on a given cycle, there are varying numbers of reports. For instance, on one hand, PowerEN mostly only has a single report per cycle, and never has more than two reports. On the other hand, SPM has an average of almost 1,400 distinct reports for each reporting cycle. SPM is an outlier, with most applications having low, single-digit numbers of reports per reporting cycle.

To get a sense for the distribution and volume of reports in ANMLZoo, we use a metric called the *index of dispersion* (IoD). The IoD is the ratio of the variance of a data stream to the mean of a data stream and, informally, measures how "bursty" data streams are. We calculate the IoD for each ANMLZoo benchmark using the number of reports per symbol cycle to get sense for the average reporting behavior of each benchmark. IoD's equal to zero (Dotstar, ClamAV) mean that the number of reports generated per cycle has a variance equal to zero, and thus the same number of reports were generated for every cycle. Applications that never report, and therefore always report '0' times are perfectly regular and therefore have a IoD of 0. IoD's less than one (Snort) indicate very regularly-spaced reporting events of regular size. Snort reports on almost every cycle and usually has one or two reports. Thus its IoD is very low (0.197). IoD's approximately equal to one (PowerEN, Protomata) are what we would expect from a Poisson distribution, indicating there is no regular pattern in reporting. IoD's greater than one (Brill, ER, SPM, Fermi, RF) indicate clumped reporting, where reporting events are more likely to be large and clustered in time, e.g. when a signature or event is recognized. SPM in particular is extremely "bursty," reporting many times per report cycle, but intermittently.

The above characterization shows that, while reporting behavior is diverse, most benchmarks with non-trivial reporting behavior report fairly frequently, but do not create many simultaneous reports, i.e. are not very "bursty". Average reports per Report Cycle usually falls between 1 and 7. Maximum values for every application but SPM never exceed 20. This result should be intuitive. Automata are usually designed for parallel matching of various diverse patterns, thus recognizing multiple patterns at once should be a rare event. These observations motivate reporting architectures that can handle these common cases with very

low overhead. In the next section, we present a parameterizable reporting architecture simulator to explore performance bottlenecks in real spatial automata processors, and potential solutions.

7.2 Simulating Spatial Automata Processors

When considering automata processing on spatial, reconfigurable architectures such as FPGAs and Micron's Automata Processor, performance of the automata engine is ostensibly equal to the operating frequency of the placed-and-routed design. Because automata matching computations and communication happens within a single cycle, the time it takes to run automata on the input symbol stream is equal to the symbol cycle time of the device multiplied by the number of symbols in the input symbol stream. While prior work often reports this nominal, "kernel" performance [20, 18], real-hardware performance also depends heavily on how the architecture handles exporting reports. Some prior work uses equation-based models to more accurately estimate output reporting costs [93, 6]. However, this technique is not validated against real hardware and fails to account for complexities of dynamic behavior.

FPGA-based automata acceleration usually only reports nominal operating frequency of the design. To the best of our knowledge, only one FPGA-based system reports real-hardware, end-to-end performance numbers on large automata [53]. However, this work only considers a single application (Random Forest [22]) and compresses reporting states using custom, application specific hardware, and is thus not a general purpose solution or analysis [53].

To solve these problems, we present a flexible methodology for both accurate performance modeling of existing spatial architectures (to identify performance bottlenecks) and architecture research (to evaluate the impacts of changes to performance sensitive parts of the micro-architecture). We first present a parameterizable automata processing simulator. Report traces generated by the Virtual Automata Simulator tool VASim [78] can be fed to this simulator to generate cycle-accurate run-time estimates. We then validate this methodology against real spatial automata processing hardware and show it is highly accurate.

7.2.1 Spatial Automata Processor System

We first present an abstract, parameterizable automata-processing system architecture that can be used to investigate high-level performance impacts of spatial architecture reporting architectures. A figure showing the abstract automata processing system is shown in Figure 7.1. Each major structure is described below.

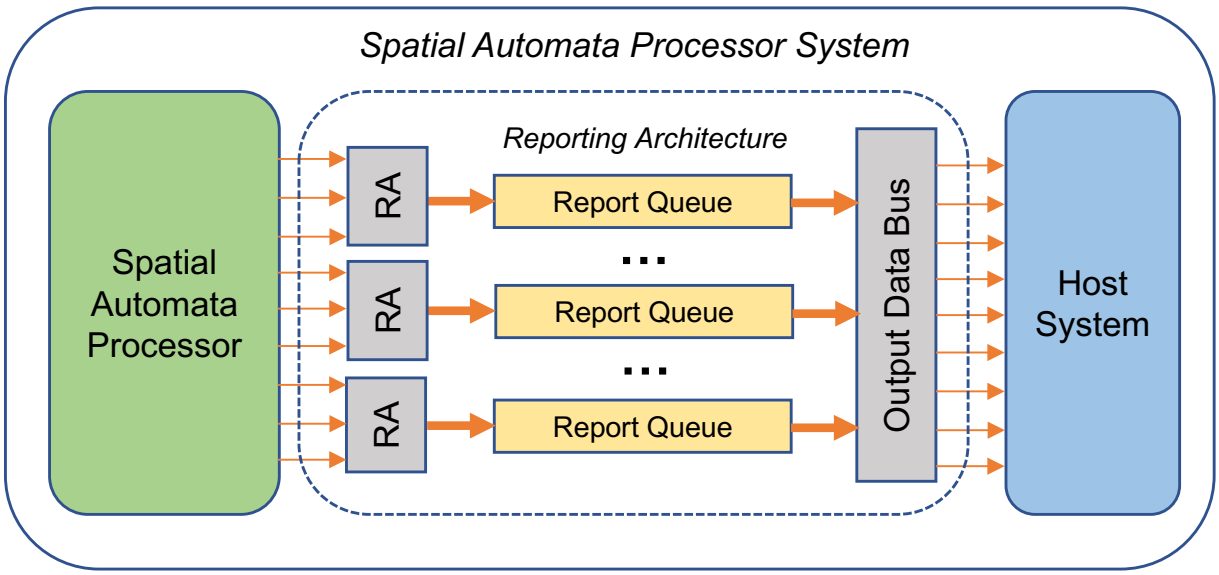

Figure 7.1: Abstract spatial automata processor system. The spatial automata processor consumes inputs at once symbol per cycle. Each reporting state is mapped to a Report Aggregator (RA). The RA takes report signals and pushes them to Report Queues. If a Report Queue fills, the system stalls and exports the Report Queue over the Output Data Bus.

Automata Processor

For spatial automata processing, computation is streaming. The inputs are the symbols that drive state transitions in parallel within the spatial fabric, and the outputs are the reports from each reporting state that activates during computation. Because the input symbol stream requires a single byte per symbol cycle, and has perfectly predictable spatial locality, we assume the input system can easily be designed to support this requirement. The automata processing architecture then consumes one symbol, computing matches, and communicating state transitions point-to-point, all within a single cycle. If a report state activates, a special signal is routed to the report aggregator circuit.

Report Aggregation

The Report Aggregator (RA) is responsible for turning reporting events into data packets that can be exported off-chip for further processing. In this system we abstract report aggregation and offer three configurable parameters: 1) the number of input signals the RA is responsible for converting into packets, and 2) the number of RA circuits assigned to the automata. We assume each RA can consume and aggregate a single report event in a single cycle in a pipelined manner.

Report Queues

Once an RA converts reporting events into data packets, the RA pushes these packets to a Report Queue (RQ). The RA can send a certain number of packets to the RQ in a single cycle. Thus, if the RA creates more packets than can be pushed to the RQ, the entire system must stall. Queues enable output to be batched for more efficient transfer off chip.

Output Data Bus

Once an RQ fills, the automata processing system stalls, and the queue is offloaded via the Output Data Bus. We assume the hypothetical system is capable of exporting the entire RQ in a single transaction with a certain cycle cost. Multiple RQs may share the same output bus, and thus must arbitrate for the bus on a reporting event.

7.2.2 Simulation Methodology

The above system can be simulated by assigning automata reporting states to ports in RAs, and tracking reporting events during automata simulation. Because we assume each symbol is executed by the automata processor on a single cycle, we set the base cost of consuming a symbol as one cycle in our simulator. We also assume that report aggregation is pipelined, and only requires one cycle to generate a report. We currently ignore pipeline startup costs as they are implementation dependent, and most likely small in comparison to the costs of megabyte input automata processing. We assume that the only event that can cause stalls in the system are the filling, and subsequent export of a Report Queues over the Output Data Bus. When a report queue fills, its export transaction cost is calculated and added to the total cycle count.

The simulation steps are formalized in Algorithm 3. For clarity, we consider reporting costs for a single report buffer and report the cycle cost for export as a fixed value. This algorithm can be directly extended to support both additional queues and export costs relative to chunks of data.

7.3 Case Study: the Micron D480 AP

To demonstrate how our parameterizable simulator can be used to identify bottlenecks in real architectures, we configure it to model performance of a the Micron D480 AP [34]. The next sections describe the architecture of the Micron D480, the parameters used to configure our spatial architecture simulator to match the D480 reporting architecture, simulator validation, and simulated performance results for the ANMLZoo benchmark

```
input  : Number of report aggregators
input  : Number of entries, q, in report queue
input  : Export cost, k, in cycles
input  : function RA returns the RA of a given state
input  : function ST returns ST state associated with report event
input  : ordered map R of reporting cycles to list of report events
output : Total number of cycles needed to process the reporting events
total_cycles ← 0;
queue_entries ← 0;
foreach c ⇒ E ∈ R do
    total_cycles ← total_cycles + 1;
    set P;
    foreach report event e ∈ E do
        add RA(ST(e)) to set P;
    end
    for i ← 1 to —P— do
        if i > 1 then
            total_cycles ← total_cycles + 1;
        end
        queue_entries ← queue_entries + 1;
        if queue_entries = q then
            total_cycles ← total_cycles + (k ∗ q);
            queue_entries ← 0;
        end
    end
end
total_cycles ← total_cycles + (k ∗ queue_entries);
return total_cycles
```

Algorithm 3: Cycle-Accurate Reporting Overhead Simulation for a Single Report Queue

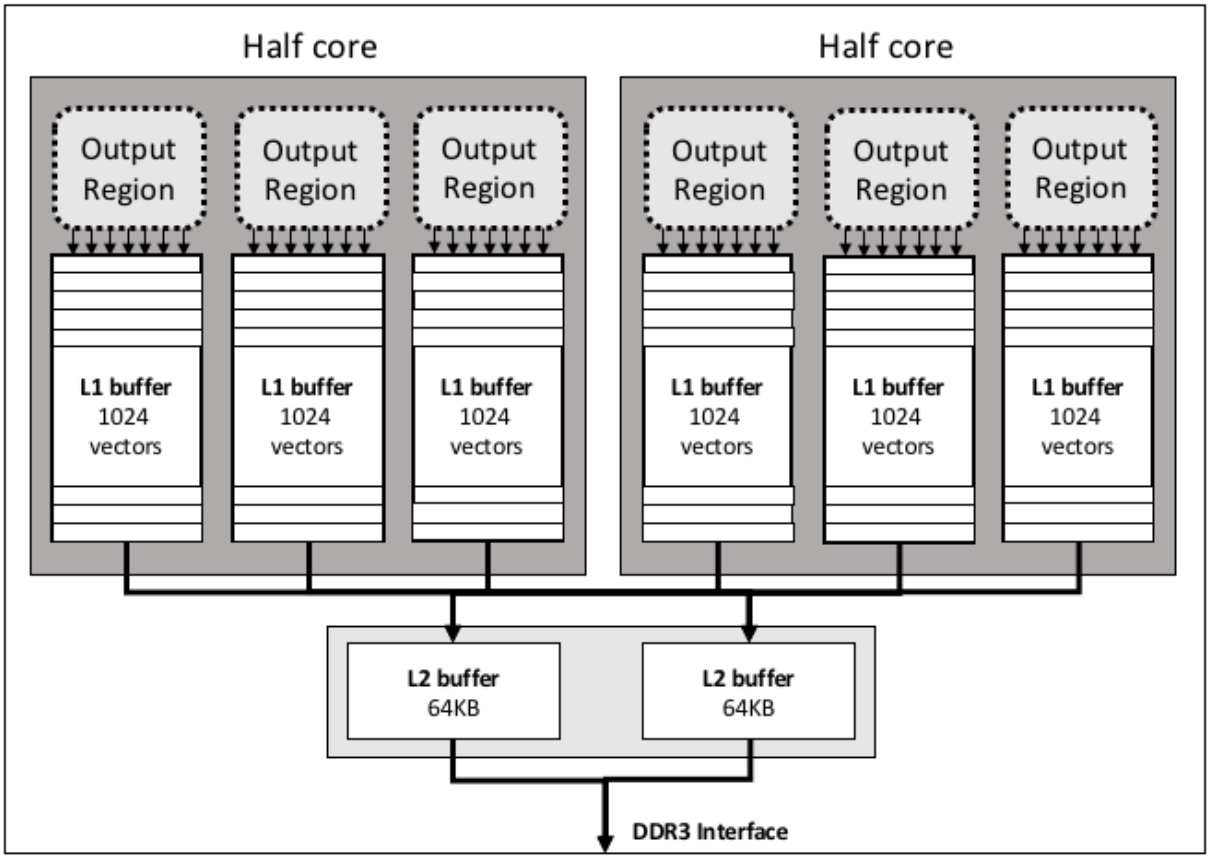

Figure 7.2: The Micron D480 reporting architecture.

suite. The lessons learned in the case study will help guide the design of future spatial automata processors, such as those developed on FPGAs.

7.3.1 The AP D480 Reporting Architecture

Figure 7.2 shows an overview of the reporting architecture of the Micron D480 AP. Each D480 chip is organized into two half-cores. Each half-core has three reporting regions, and each reporting region is responsible for recording up to 1,024 single-bit reports from 1,024 different states into a report vector on any given cycle. Reports are generated by routing the outputs from reporting states to ports in each reporting region. If any one state reports in a region, the region generates a bit-mapped report vector with 1,024 report bits (where 0's represent no report, and 1's represent a report), and a 64-bit metadata tag containing the region and cycle information of that report. These report vectors are then pushed to a first-level (L1) storage buffer.

When full, L1 buffers are exported into one of two global, second-level storage buffers for eventual export off-chip [1]. The AP must stall when an L1 buffer transfers its contents to an L2, because a report vector generated in subsequent cycles cannot be pushed to the L1 buffer while it is exporting vectors. However, the

D480 Structure	Default
Half-cores per chip	2
Reporting regions per half core	3
L1 report vector buffers per region	1
Report vector width	1024 bits
Vector metadata size	64 bits
L1 buffer entries	481
L1 empty check cost	2.5 cycles
L1 export initiation cost	15 cycles
L1 export cost per 8B chunk	2.5 cycles
L1 vector export cost	40 cycles
L2 buffers per chip	2
L2 buffer size	64kB
L2 vector export cost	*N/A*

Table 7.2: Model parameters corresponding to the first generation Micron D480 Automata Processor core architecture.

AP does not stall when an L2 buffer transfers its contents off-chip because this structure is double-buffered (i.e. when one L2 buffer is being exported off-chip, the other, sibling buffer can be used simultaneously to import report vectors). Currently, when an L1 buffer fills, the AP must check every region for reports. If a region is empty, this check costs 2.5 cycles. Table 2 shows the cycle costs of each of these operations reported by Micron [1]. Table 3 shows the simulator configuration parameters to match the Micron D480.

Report Vector Division

Because exporting large reporting vectors can be expensive, the Micron D480 allows report vectors in reporting regions to be statically reduced in size in the case that all of the ports in the full region are not required. Dubbed Report Vector Division (RVD) [1], this technique attempts to statically route reporting states into consecutive reporting ports in an output region. If the reporting region can use 512, 256, 128, or 64 ports, rather than the available 1,024, the D480 can be configured to export the divided, rather than the full, vector. While alpha D480 hardware does not have this feature enabled, we configure the simulator to evaluate its performance impact.

Simulator Configuration	Default
Report Aggregators	6
Report Queues	6
RA/RQ width	1024 bits
Vector metadata size	64 bits
Queue Entries	481
Queue empty check cost	2.5 cycles
Queue export initiation cost	15 cycles
Bus cost per 8B chunk	2.5 cycles

Table 7.3: Spatial architecture simulator configuration corresponding to the Micron D480 AP [1].

7.3.2 Cycle-Accurate Simulation

The Micron D480 AP can be simulated using our parameterizable spatial architecture simulator, by setting parameters to be as close as possible to real hardware. Each reporting region can be simulated as a separate RA. Because there are 3 reporting regions in each of 2 half-cores, we configure the simulator to have 6 RAs. Reporting regions are 1,024 bits wide, and also include a 64-bit metadata tag. Therefore, we set each RQ entry to be 1,088 bits wide. The AP can transfer a report vector per cycle to each corresponding L1 buffer, thus we set the queue push throughput to be one packet per cycle. Each L1 buffer can hold 1,024 132-byte vectors, but cannot hold more than 64kB of data (the size of the OEB). Thus, we set the number of entries in the RQs to be 481 (64kB/1,088 bits). We also augment the simulator to account for other dynamic costs such as report export initialization costs, and buffer "empty" checks [1].

Because performance depends on both when and where reports occur on chip, we use placement information emitted by the Micron spatial compiler to better identify which RA input ports reporting states are assigned to. We first place-and-route automata using Micron's compiler. We then extract placement information embedded in this representation, and create a new hardware accurate automata graph such that every state is properly tagged with the coarse-grained AP hardware region it was assigned by the compiler [1]. In this way, we can approximate which reporting regions reporting states are assigned to, improving the accuracy of the simulator.

Once states are assigned to input ports of the RAs, we run the automata on an input using VASim and create a report trace. This trace is then fed to the cycle-accurate simulator for processing as described in Section 7.2.2. The simulator processes the report vector, assigning cycle costs based on the reports generated on any given cycle, and the associated query, transfer, and stall costs. The total number of cycles can then be multiplied by the cycle time of the device to estimate total runtime.

7.3.3 Simulator Validation

We validate our cycle accurate simulation methodology against real hardware by comparing the actual wall-clock runtimes of automata on alpha release D480 AP boards with runtimes generated by our spatial compiler. We first estimate driver overhead by running the automata application with no input stimulus. This measured time encompasses all CPU, PCIe, firmware, and miscellaneous overheads associated with initiating computation. Any runtime on top of this is due to automata processing and reporting overheads. This value can then be subtracted from runtimes collected from real hardware runs for comparison with simulation.

[1] Kevin Angstadt designed and built much of the placement extraction tool.

We consider a synthetic application to validate the performance model. The synthetic application is made up of automata that are both a start state and a reporting state, and matches on a single stimulus character. Thus, any time a stimulus character appears in the input stream, a report is generated for every state. The size of this report vector, and the number of report regions used can be controlled by adding or subtracting additional automata.

We compile enough synthetic automata to occupy at least one reporting port in every report region in both AP half-cores. Because report vector division is not enabled in alpha hardware, we guarantee full report vectors will be exported from every output region region whenever a stimulus character is seen in the input stream. We vary the frequency of stimulus characters in the input by a constant amount and record the performance of varying *cycles per report* or CpR. A lower CpR means more frequent reports, higher pressure on the reporting architecture, and a higher performance penalty. We evaluate three different CpR values, two, three, and four through 481 input symbols. Alpha D480 firmware currently does not currently support contiguous inputs longer than 481 input symbols with high reporting rates. Larger inputs can still be supported by breaking the stream into chunks, but this introduces additional, unrelated overheads that we do not wish to measure. We therefore leave verification on release hardware over the entire ANMLZoo benchmark suite to future work. Figure 7.3 shows normalized runtimes of real AP hardware and runtimes predicted by our spatial architecture simulator using the three different input stimulus files. Predicted performance matches real performance to within 2.3%-4.6%.

7.3.4 ANMLZoo Reporting Overheads

We use the simulation methodology described above to simulate performance of ANMLZoo applications on the Micron D480. Figure 7.4 shows the overhead associated with output reporting for all 12 non-synthetic ANMLZoo benchmarks [22]. Some benchmarks incur extremely large reporting overheads. For example, Snort incurs a 46× slowdown over ideal performance, and 6 out of 12 benchmarks spend more time processing reporting overheads than processing automata transitions! Some benchmarks have little or not reporting overheads. This is simply because these benchmarks reports infrequently or not at all at all.

Report Vector Division (RVD) is simulated by counting the report ports per region and configuring the report vector to be the appropriate size. RVD makes a large impact on performance when there are relatively few reporting ports required, but frequent reporting. For example, RVD decreases reporting overhead by approximately 50% for Snort, Brill, Protomata, ER, Fermi, and RF.

While RVD does help improve performance, reporting overhead is still extremely high for many benchmarks. These high reporting overheads can cancel out much of the benefit of spatial acceleration. The rest of this

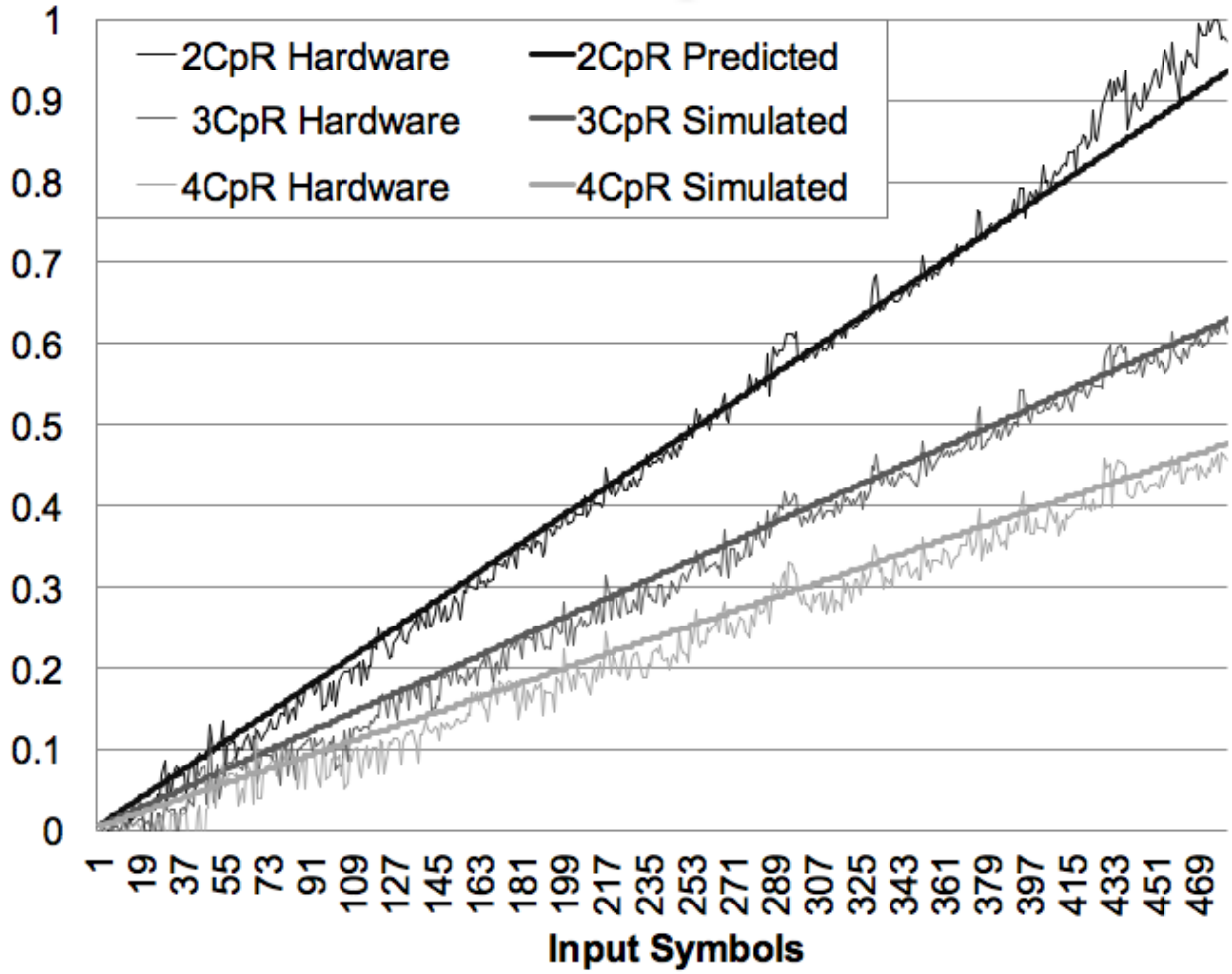

Figure 7.3: Normalized performance of alpha release AP D480 hardware compared to performance predicted by our trace-based, cycle accurate simulator. Predicted performance matches real performance to within 2.3%-4.6%.

chapter attempts to understand what causes high reporting overheads and propose solutions to mitigate them.

7.4 Automata Transformations to Reduce Reporting Overhead

Automata transformations are extremely important for improving performance on von Neumann architectures, and reducing capacity requirements on spatial architectures [22]. Automata compression does not require any changes to hardware, and thus can easily be implemented on existing systems. This section presents an automata transformation to reduce the number of required reporting ports called "disjoint report merging" or DRM.

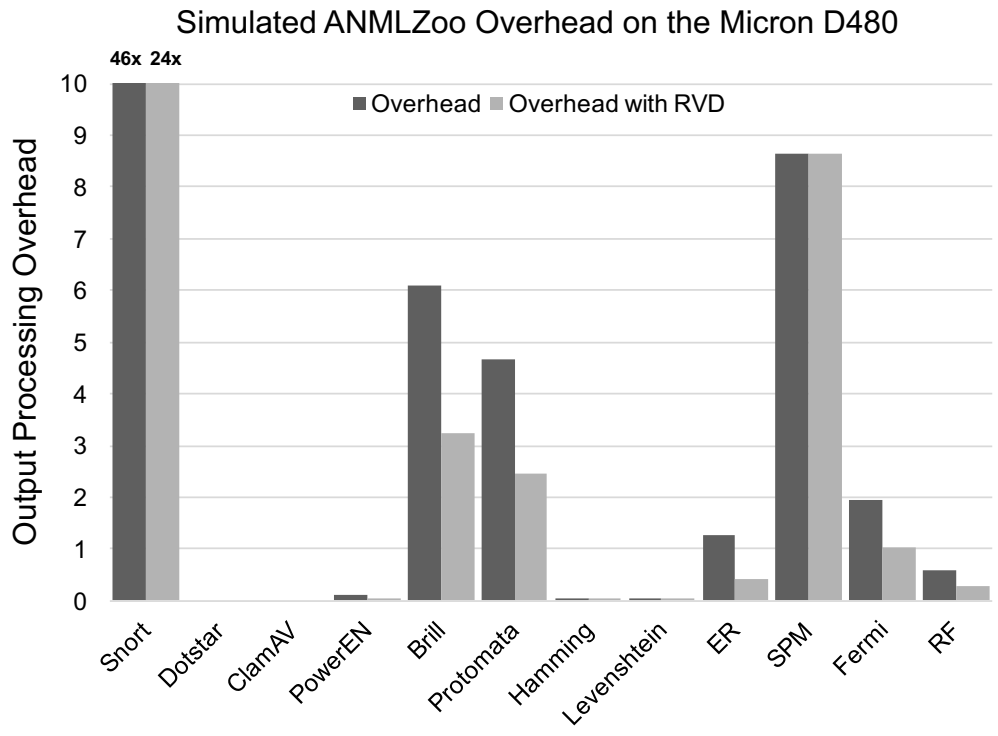

Figure 7.4: Simulated Micron D480 output processing overhead for each non-synthetic application in the ANMLZoo benchmark suite. Snort is 46× slower than ideal because it is so bottlenecked by the D480's reporting architecture. 6/12 applications spend more time exporting reports than actually processing the finite automata.

7.4.1 Disjoint Report Merging

Because reporting ports are a scarce resource, and generally increase report vector sparsity, it is desirable to decrease the total number of required reporting ports. Reporting ports can be reduced by having multiple states share a single port. However, when states share a port, multiple reports on the same cycle might be masked as a single report, or we might not be able to discern which state generated the report. Thus, it is only safe to share a reporting port if a set of reporting states provably never use the port on the same cycle. This is possible if the character sets of the reporting elements have no characters in common (disjoint), and thus can never match on the same cycle. We propose to merge the ports of reporting states with disjoint character sets to reduce the output port requirements. We call this technique "Disjoint Report Merging" or DRM.

DRM identifies sets of reporting states with disjoint character sets that provably cannot match on the same input. DRM then assigns all members of a set to the same reporting port. Unioned reports cause ambiguity when they occur (i.e. we do not know which original state reported). However, because their character sets are disjoint, it is trivial to disambiguate which reporting state was responsible for the report

by examining the symbol that caused the report on the host system.

As an example, if reporting state **(1)** has character set **[a]** and reporting state **(2)** has character set **[b]**, a report from their union **(1 or 2)** may represent **(1)** or **(2)**. However, the character sets in **(1)** and **(2)** are disjoint and we can therefore recover the original reporting state by examining the triggering input symbol. In the example above, if the symbol that caused the unioned report **(1,2)** to match was **b**, the report came from **(2)**.

By reducing the number of required reporting ports, we reduce RA input port requirements, and may induce report vector division (described in Section 7.3.1) possibly reducing reporting overheads.

input : **set** R of reporting state **state** objects
input : **function** *Children* returns output connections from given **state**
input : **function** *Matches* returns **char set** of matching input stimuli for an STE **state**
output : **mapping** from reporting **port** to **set** of merged reporting **state** objects

mapping *ports*;
foreach state $r \in R$ **do**
  if $|Children(r)| > 0$ **then**
    continue
  end
  port *sink*;
  $ports(sink) \leftarrow \{r\}$;
  $R \leftarrow R \setminus \{r\}$;
  char set $match \leftarrow Matches(r)$;
  foreach state $r' \in R$ **do**
    if $|Children(r')| > 0$ **then**
      continue
    end
    if $match \cap Matches(r') == \emptyset$ **then**
      $ports(sink) \leftarrow ports(sink) \cup \{r'\}$;
      $R \leftarrow R \setminus \{r'\}$;
    end
  end
end
return *ports*

Algorithm 4: Disjoint Report Merging

7.4.2 DRM Algorithm

Pseudocode for identifying disjoint character sets in reporting states is provided in Algorithm 4. We implement DRM as a VASim pass over the reporting states in each ANMLZoo benchmark. DRM is accomplished by examining reporting states and grouping states that have disjoint character sets.

Because routing a large number of outputs from merged states to a single reporting port might create congestion in the reconfigurable routing matrix, we only consider merging reporting states that have no outgoing connections. We also restrict DRM to merge disjoint reporting state ports in the same connected component subgraph. These restrictions make it more likely that only states that are close together in the

Benchmark	Orig.	Comp.	Factor	Speedup
Snort	1,955	1,364	30.2%	40.4%
Dotstar	1,290	343	73.4%	0%
ClamAV	515	164	67.9%	0%
PowerEN	2,920	1,054	63.9%	2.6%
Brill	1,886	1,886	0%	NA
Protomata	2,338	2,338	0%	NA
Hamming	279	156	44.0%	0%
Levenshtein	178	84	52.8%	0%
ER	1,406	1,406	0%	NA
SPM	5,025	5,025	0%	NA
Fermi	2,399	1,030	57.1%	26.2%
RF	1,661	1,661	0%	NA

Table 7.4: Number of required reporting ports in the compiled ANMLZoo benchmarks before and after disjoint report merging. Some applications cannot be compressed using this technique. Speedup measured performance improvement due to DRM when compared to the simulated Micron D480 with RVD enabled.

architecture will be merged and not increase reconfigurable fabric resource requirements, and a more realistic measure of potential benefit.

7.4.3 DRM Potential Study

We apply the DRM algorithm described above to every automata benchmark in the ANMLZoo benchmark suite [22], and compare the original number of required reporting ports to the final number after DRM to identify how much opportunity for report port compression exists. Table 7.4 shows the original and compressed number of reporting states.

The reporting ports of many benchmarks can be compressed by large amounts. For instance, reporting ports in 5 out of 12 of the applications can be compressed by more than 50%. 73% of Dotstar's reporting ports can be compressed from $1,290$ to just 343. However, reporting ports in 5 out of 12 applications cannot be compressed using DRM at all. For example, Brill has $1,886$ reporting states, but each has an identical character set, and are thus are never disjoint. Brill can be report-compressed if the algorithm considers the second-to-last level of matching states, and disambiguates reports using a symbol lookup into the second to last symbol that caused a report. Ideally, DRM merges the ports of automata with identical suffixes until it encounters parent states with disjoint character sets. We leave development of this algorithm for future work.

7.4.4 DRM Performance Impact

We simulate the performance of each ANMLZoo benchmark using ANMLZoo's 1MB input files before and after DRM is applied. The simulator first re-maps outputs from disjoint sets to a single report vector port in the architecture. RVD is then applied. DRM only increases system performance if it induces RVD. The

last column in Table 4 shows the simulated speedup of the DRM version of the automata over the original application with RVD enabled.

While many applications are compressible, if reports are infrequent, or if RVD is not induced, there is no performance benefit for DRM. However some applications benefit greatly from DRM. Snort's end-to-end performance was improved by $\sim 40.8\%$, reflecting a $\sim 42\%$ reduction in report processing overhead. Fermi's end-to-end performance was improved by $\sim 26\%$, reflecting a $\sim 57\%$ reduction in in report processing overhead.

7.5 Identifying Architectural Bottlenecks in Reporting

While DRM can reduce output port requirements and induce report vector division, DRM is not a general technique, and does not help decrease reporting overheads for most of the ANMLZoo benchmarks. For example, DRM cannot be applied to the SPM benchmark, which has a reporting overhead of $\sim 8x$. Snort, which enjoys the largest benefit from DRM, still has a reporting overhead of $\sim 13x$. The following sections first characterize the reporting bottleneck in the Micron D480 architecture. We identify that a large percentage of reporting bottlenecks are caused by extremely sparse reporting vectors and explore a new reporting architecture design that reduce the sparsity of these vectors.

7.5.1 Characterizing Report Vector Sparsity

Report Aggregators (RAs) are configured to export reporting events as bit vectors where set bits correspond to states that reported on that particular cycle. Whenever a report occurs, a sparse vector is generated and pushed to the corresponding Report Queues (RQ). Because each report vector is tagged with 64-bits of metadata, wider RAs can amortize the cost of this metadata over a larger number of reports. However, if there are few reports per report cycle, RAs that are too wide introduce large levels of sparsity, and can clog the Output Data Bus with a large amount of unnecessary data.

Section 7.1 identified that there were usually between 1-7 reports per cycle, and very rarely a large number. Thus, we hypothesize that RAs that are 1,024 bits wide introduce a large amount of sparsity. We measure the density of each reporting vector by recording the ratio of 1's in a report vector to total bits, not counting the metadata. A larger density (lower sparsity) means that more meaningful data is being recorded. Results are shown in Figure 7.5.

In general, density is extremely low. Most applications, even with RVD applied, do not use more than 0.5% of the available vector space. SPM is an obvious outlier as its average density is 22.7%. For each reporting cycle, SPM must account for an average of $\sim 1,394$ reports, where the rest of ANMLZoo averages

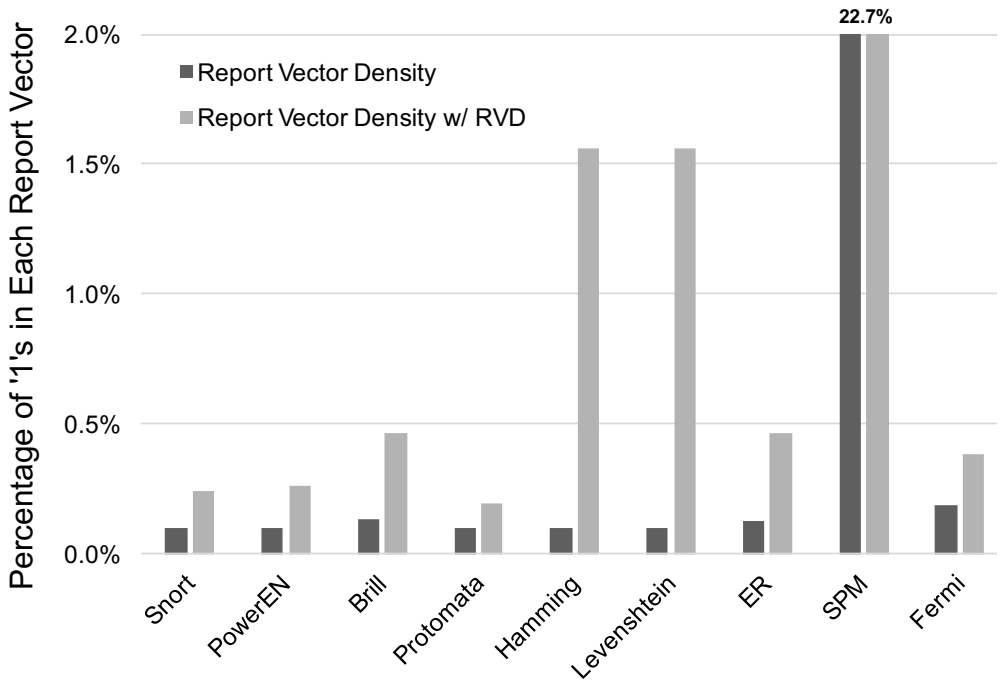

Figure 7.5: Report vector density (ratio of '1's to total bits) for all applications in ANMLZoo. Most applications have extremely sparse reporting vectors. Report Vector Division (RVD) statically re-sizes report vectors to reduce vector sparsity known at compile time.

between 1 and 7. While the common case is sparse reporting, it is also important to make sure we do not hurt performance of applications with denser reporting.

Report Vector Division more than doubles report vector density in all applications but SPM. This is because RVD is able to statically reduce RA size, removing ports that provably will always be '0's.

Because our spatial automata processing system must pay a cycle penalty for every exported bit, this sparsity is a huge source of inefficiency. The next section explores modifications to the spatial architecture model to reduce this sparsity, and decrease reporting overheads.

7.5.2 Reducing Output Sparsity

The previous section showed that report vectors, even when statically divided using RVD, were extremely sparse, causing large and unnecessary overheads. To solve this problem, we modify the architecture, splitting RAs into finer grained structures or sub-RAs. These finer grained structures can be configured to push smaller packets to the output queue when reporting is sparse, or be chain-ganged together into sub-groups to push larger packets when reporting is dense. We call this technique Report Aggregator Division (RAD). Similar to RVD described in Section 7.3.1, RAD generates smaller packets, reducing the sparsity of output.

However, unlike RVD, RAD does not require the automata to use a small number of report ports and can support very large numbers of ports without paying a penalty for sparse output.

Report Aggregator Division

We implement RAD by dividing each 1,024-bit wide RA into 64, 16-bit-wide sub-RAs. Each sub-RA is statically responsible for 16 reports from the automata fabric. When reports occur in the automata fabric, sub-RAs generate small, 16-bit report packets. Sub-RAs can be chain-ganged together into equal-sized sub-groups to generate larger packets if reporting is dense.

To keep track of when and where reports are generated in an RA/RQ pair, we add a metadata generator block (MGB). The MGB is responsible for tagging data packets generated by sub-RAs with the symbol cycle that generated the packet, the ID of the sub-RA that generated the packet, and the RAD configuration (the size of the sub-groups) of the RA. Each metadata tag is 64-bits and consists of a 32-bit field to hold the index of the cycle that generated the packet, a 16-bit field to hold the ID of the sub-RA that generated the packet, and a 16-bit field to identify how many sequential sub-RAs are currently chain-ganged together into a sub-group. In order to support the possibility that more than one sub-RA or group of sub-RAs generates a packet on a given cycle, we add a hardware structure to control how packets are pushed to the RQ called the arbitration unit (AU). The AU multiplexes packets from sub-RA groups and pushes them to the RQ. If more than one packet is generated by a sub-RA group on the same cycle, the AU stalls automata processing and pushes each packet to the RQ until automata processing can resume. Sub-RA groups are configured by setting appropriate configuration bits in the MGB and AU.

Because RAD configuration for each RA/RQ pair is carried via the metadata packet, the number of sub-RAs chain-ganged together in a sub-group can be configured at any time by stalling processing and re-setting the appropriate bits in the MGB and AU. RAD reconfiguration is designed to be light-weight, and does not require a recompilation or a separate place-and-route step. The augmented system supporting RAD is shown in Figure 7.6.

Sensitivity Analysis

We explore the potential benefits of RAD by adding RAD capabilities to the spatial architecture simulator and simulating system performance on the Snort and SPM ANMLZoo benchmarks.

We increase the RAD division factor from 1 (64 sub-RAs chain-ganged together) to 64 (16 input ports for each of 64 independent sub-RAs) and measure reporting overheads. Every other parameter in the simulator is set to the default Micron D480 setting. Results are shown in Figure 7.7.

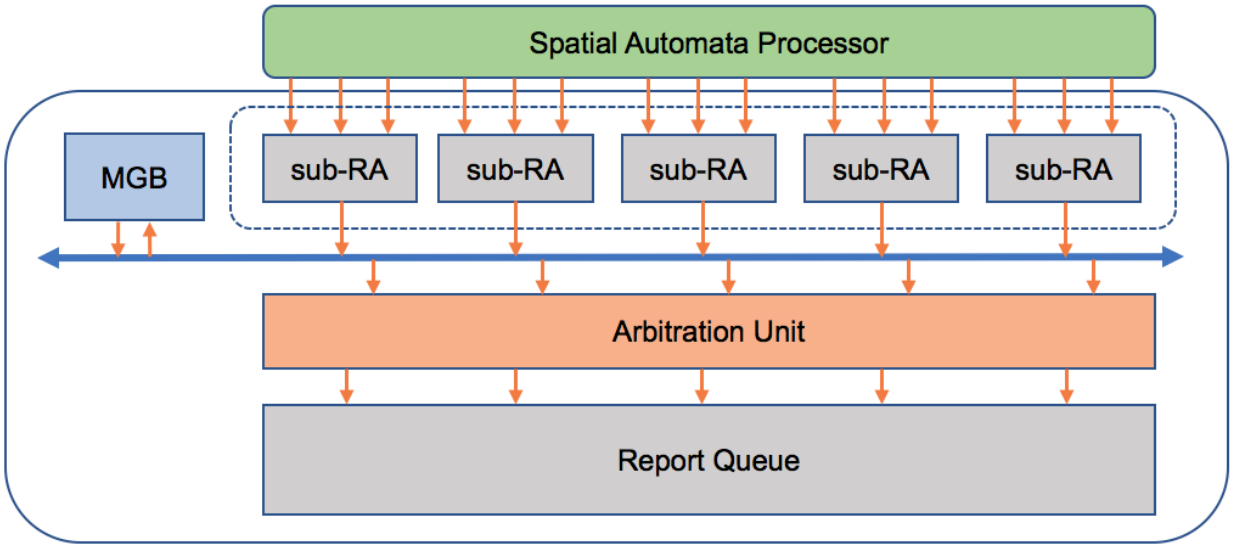

Figure 7.6: Spatial Reporting Architecture with report aggregation split into sub-modules. The Metadata Generator Block tags report packets with RAD configuration information, the sub-RA ID where the packet was generated in this configuration, and the cycle index the packet was generated. The Arbitration Unit combines and arbitrates packets from sub-RAs to be pushed to the report queue.

A RAD factor of 1 represents the original configuration of the Micron D480 AP. Because Snort reports are frequent and sparse (low IoD), Snort benefits greatly from a high RAD factor. A RAD factor of 64 (64 sub-RAs with 16-bit packets) reduces reporting overheads to $3.8x$ versus $46.3x$ when the RAD factor is 1 and configured to match the Micron D480 AP. On the other hand, because SPM's reports are infrequent and dense (high IoD), SPM shows better performance from a low RAD factor, and performs best when RAD is configured to match the Micron D480 AP. This result highlights the benefits of a flexible reporting architecture: the RAD architecture allows us to tune the RAD factor to best match the reporting behavior of an application.

Results

We simulate all ANMLZoo benchmarks using the original architecture with RVD enabled, and compare the results to our RAD-enabled architecture with the highest performing RAD factor. This corresponds to 64 for all benchmarks but SPM, which does not benefit from RAD. Results are shown in Figure 7.8.

When compared to RVD, RAD is able to reduce reporting overheads by 66% to 84% for applications with sparse reporting behavior. Unlike RVD, RAD is able to reduce sparsity while also allowing a large number of input ports.

For benchmarks with large reporting overheads, RAD greatly improves performance in almost all cases. For example, RAD improves the performance of Snort, which had a $24x$ reporting overhead with RVD enabled,

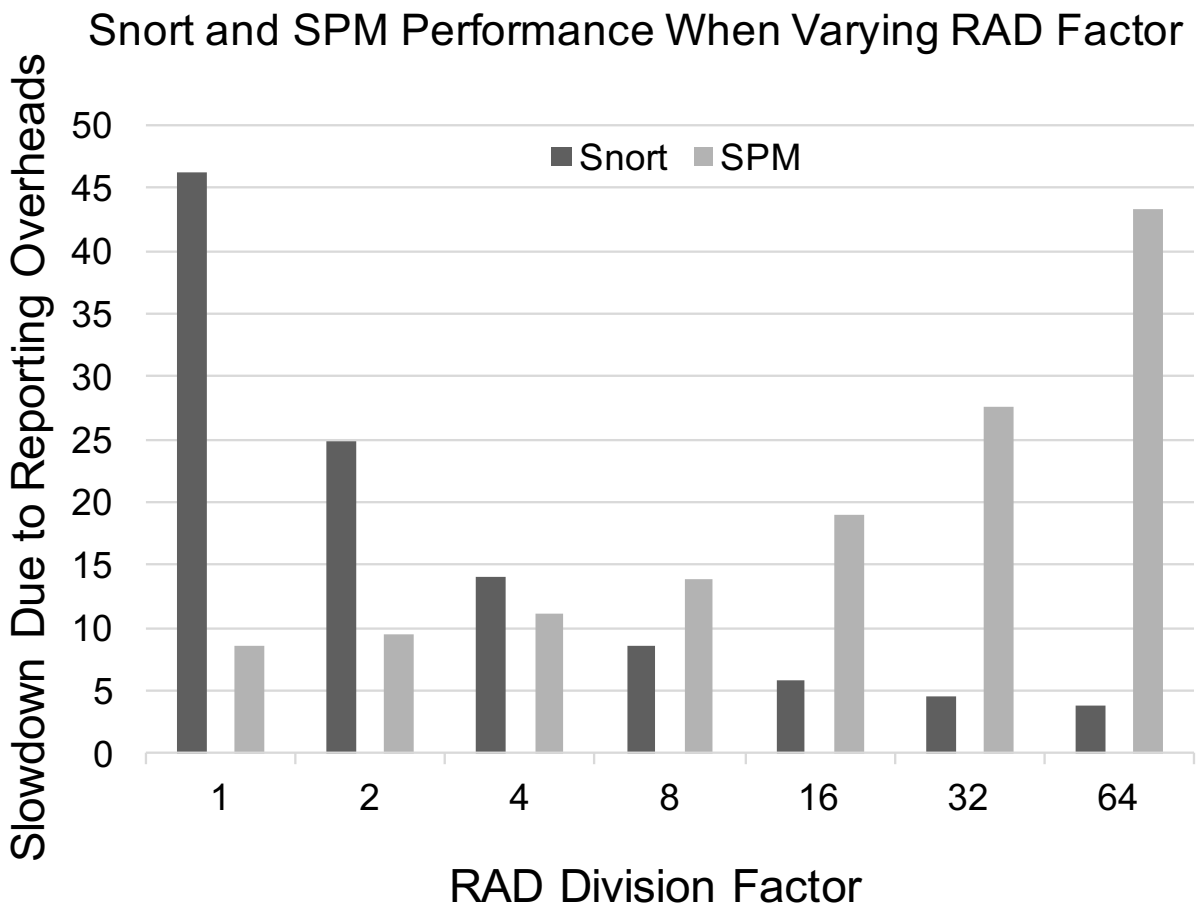

Figure 7.7: Reporting overheads as a function of increasing RAD factor for Snort and SPM. Snort has sparse reporting behavior, and thus benefits from smaller packets. SPM has dense reporting behavior, and benefits from larger packets.

by 5.1x.

When reporting is dense, such as in SPM, RAD has no positive benefit. However, importantly, RAD does not hurt performance, as it is configurable to account for dense reporting behavior.

While these results are impressive, other techniques can be employed to further reduce overheads. Future work will investigate the cost of implementation of the most promising solutions discovered by our spatial architecture simulator.

7.6 Discussion and Future Work

This chapter motivates automata reporting as a critical bottleneck in practical automata processing architecture design. We first characterized automata reporting to identify common-case behavior and showed that reporting behavior for most benchmarks is sparse, but can be dense. We then show that architectures that over-design

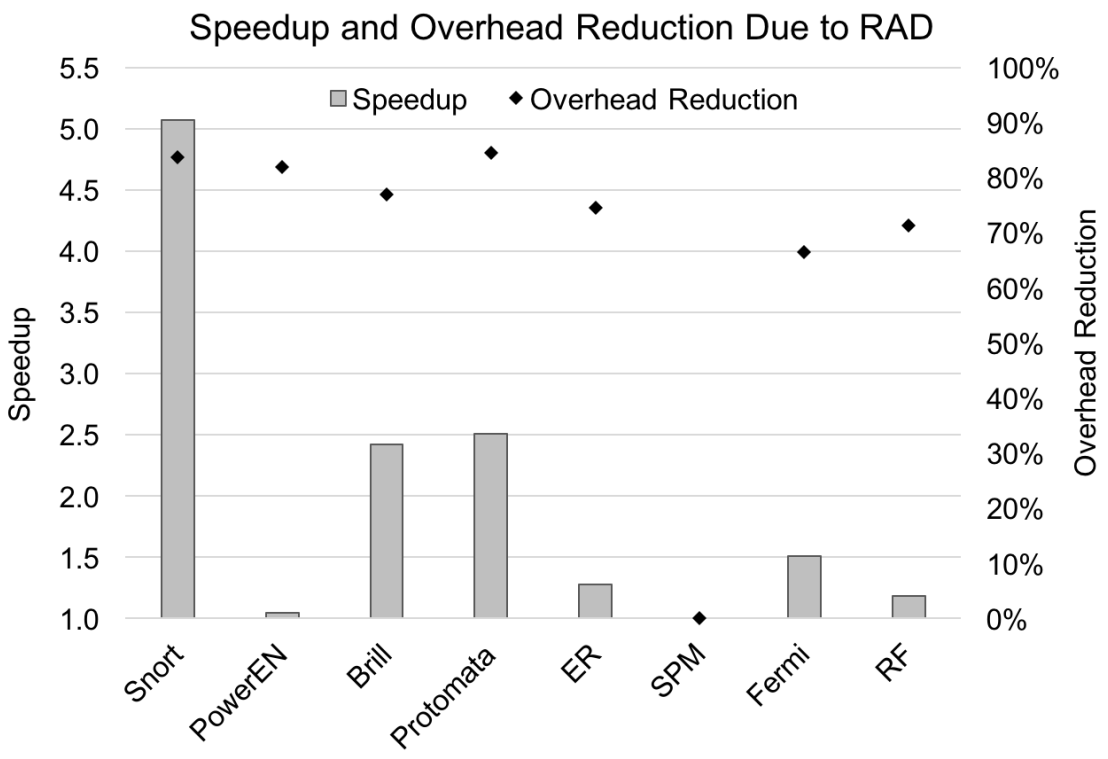

Figure 7.8: Speedups and reduction in reporting overhead due to RAD. SPM did not benefit from RAD because it generates dense reporting vectors.

for dense reporting perform poorly for applications with sparse reporting. We then present a configurable reporting architecture that is able to efficiently handle common-case, sparse behavior better than existing systems, while also not hurting performance when reporting is dense.

Could different inputs hurt performance of a RAD configuration? Inputs that differ from the characterization inputs might cause different reporting behavior, and might hurt performance of a chosen RAD configuration. Because reporting behavior can be diverse, we designed the RAD reporting architecture to be configurable to account for such changes in behavior. If average case behavior is not captured by the profiling inputs, and reporting overhead is high, the architecture can be quickly reconfigured with a new RAD factor by updating the settings of the MGB unit in all, or some of the RA/RQ pairs. In future work, more applications and more inputs could be used to generate more complete application characterizations that may motivate improvements to this architecture. Future work could also monitor reporting behavior at runtime to guide dynamic reconfiguration of the RAD architectures, in order to respond to variations in reporting behavior.

Could compression circuits further reduce sparsity? Yes. Future work could explore the trade-off

space in compression circuit area and power costs versus performance. Careful attention should be paid such that frequent or dense reporting does not overwhelm these compression circuits. Application specific compression schemes are an extremely promising path for future work. Architects could build in certain popular compression kernels such as report counting and thresholding [16], or classed report voting [8, 53].

Why is there still a gap between the potential and achieved speedups? While RAD enables much more efficient reporting for benchmarks with sparse but frequent reporting, the architecture still must stall for every reporting event. Completely eliminating these stalls might be accomplished by double buffering the report queues so that reports from one queue could be exported off-chip while the automata continues to run and pushes report packets to a second sibling queue. We leave evaluation of the impacts of double buffering, as well as other techniques to reduce reporting overhead to zero, as future work.

Could report signal to RA routing cause congestion in the reconfigurable routing matrix? Yes. All spatial architectures must somehow route reporting signals to ports in an RA or similar structure, and these signals might cause congestion in the reconfigurable routing matrix. DRM might exacerbate routing congestion by wiring many input signals from automata states placed far away to the same reporting port. Our DRM algorithm attempts to minimize this potential impact by only grouping states with no other output signals, and within the same a connected component. The RAD architecture operates independently of the routing fabric, and does not affect routing constraints.

Do results extend to other spatial architectures? The lessons of these studies not only apply to current spatial automata processor architectures like the D480, but also motivate designs of reporting architectures for FPGA-based automata processing engines, and reporting architectures for next generation spatial automata processing ASICs. Most prior work in spatial automata processing overlooked reporting architecture design and performance impact. Our work shows that reporting cannot be ignored when considering a wide variety of real-world applications, and should be carefully designed to consider a wide variety of reporting behavior.

7.7 Conclusions

Spatial automata processing architectures offer an exciting acceleration opportunity for the widening array of automata-based applications. However, because of their massively parallel nature, spatial architectures can suffer from output reporting constraints. This chapter first characterizes reporting behavior of automata applications. To the best of our knowledge, this work is the first to characterize reporting behavior over a large set of diverse automata benchmarks. We identify that reporting can be frequent, but is usually sparse in nature.

To identify performance impacts of reporting, we design a parameterizable spatial automata processing simulator. This simulator can be configured to behave like a wide range of real and hypothetical spatial automata processing systems. The simulator uses report traces generated by offline automata processing, and measures the costs associated with exporting these reports off chip. We use this spatial automata processing simulator to measure the overhead due to reporting in a commercial spatial automata processing architecture–Micron's D480 Automata Processor. Reporting overheads for many applications are extremely large. For example, Snort has a projected reporting overhead of $\sim 46x$, and 6 out of 12 applications spend more time exporting reports than processing automata symbols.

We explore two novel methods to reduce reporting overheads in spatial architecture systems. One software only method, and one hardware architecture modification. The software method transforms the automata, merging reporting outputs that can provably be disambiguated after computation. The hardware method modularizes report aggregation units so that they can be divided into finer-grained, independent structures. We show that modularizing report aggregators can reduce reporting overheads by up to 84% and increase performance by up to $5.1x$.

Chapter 8

Hybrid Spatial/von Neumann Automata Processing

8.1 Introduction

Power limitations brought on by the breakdown in Dennard scaling, and ever increasing transistor counts brought on by Moore's law, have led to underutilized or "dark" silicon. Adding specialized accelerators to architectures has emerged as a way to utilize dark silicon in order to extend traditional performance scaling trends [94]. Specialized accelerators allow for greatly reduced power budgets and increased performance for specific types of computation when compared to general purpose architectures.

One area of specialization that has garnered increased attention over the last decade is automata processing, also known as finite state machine or finite state automata simulation. Automata processing is one of the 13 Berkeley Parallel Motifs [95], and is widely used for regular-expression-based streaming pattern matching. Automata processing is a central kernel in deep packet inspection (for network traffic analysis and intrusion detection [4]), data mining, database queries [96], and virus detection [7]. Other uses for accelerated automata processing have been discovered in recent years. Automata have been found to be useful in bioinformatics [12, 11, 13], machine-learning [8, 10], pattern mining [15, 16, 17], natural language processing [19, 20], and other important application domains [22].

Informally, finite automata are directed graphs of state nodes. Each directed edge in the graph has a transition rule associated with it. Computation begins in "start" states, and transitions in the automata are guided by a global input string of symbols. If the automata ever enters into a "final state", it has recognized some pattern in the input, and this event is recorded as an output. Large-scale automata

processing is generally difficult on traditional von Neumann or "temporally programmed" architectures [97] because automata simulation involves a large number of parallel, unpredictable memory accesses, with little computational intensity (aka "pointer-chasing"). Thus, while temporal processors, such as CPUs, can fit extremely large automata graphs in main memory, their performance depends greatly on how active automata are, and how accesses utilize the memory hierarchy.

Reconfigurable "spatial" architectures, such as FPGAs, are ideal target architectures to accelerate large-scale automata processing. Spatial architectures can "lay out" automata states as processing elements in a reconfigurable fabric. Transitions between states can be implemented using point-to-point connections wired via an on-chip routing matrix. Once placed-and-routed, all automata states can compute in parallel, within a single cycle, no matter how active the automata. The downside to spatial architectures is that they can be greatly limited in processing element capacity and routing resources. If an automaton is too large to fit into the spatial architecture or if the graph topology is too complex to route using the on-chip routing resources, the automaton graph cannot be executed, or the graph must be partitioned and executed with multiple passes [16]. Larger fabrics, or more chips increase available resources to compute larger automata, but these large fabrics usually also have much higher costs, and can greatly increase system complexity.

Hybrid Automata Processing Intuition: *Note that the pros and cons of each architecture are complementary.* On one hand, temporal architectures can hold extremely large automata graphs in main memory, but perform poorly when computing large amounts of parallel activity. On the other hand, spatial architectures have highly-limited capacity and routing constraints, limiting the size of the automata they can implement, but can efficiently compute arbitrary amounts of parallel activity.

This chapter proposes processing automata on *hybrid spatial/temporal architectures.* If automata graphs can be partitioned so that the spatial processor computes small, densely-active portions of the automata, and the temporal processor computes large, sparsely-active portions of automata on the temporal processor, a majority of the benefits of both architectures can be realized. Intuition tells us that many automata should have this lopsided behavior. Automata are usually designed as *filters.* In filter-style automata, a vast majority of candidate matches are filtered out in the first few states (high activity), and states deeper in the filter are rarely needed (low activity).

If an automaton transition crosses from a hot region to a cold region, a signal must be sent from the spatial processor to the temporal processor to resume computation at the correct location. Figure 8.1 shows a high level overview of how filter-style automata could be profiled and partitioned across cores in a hybrid spatial/temporal system.

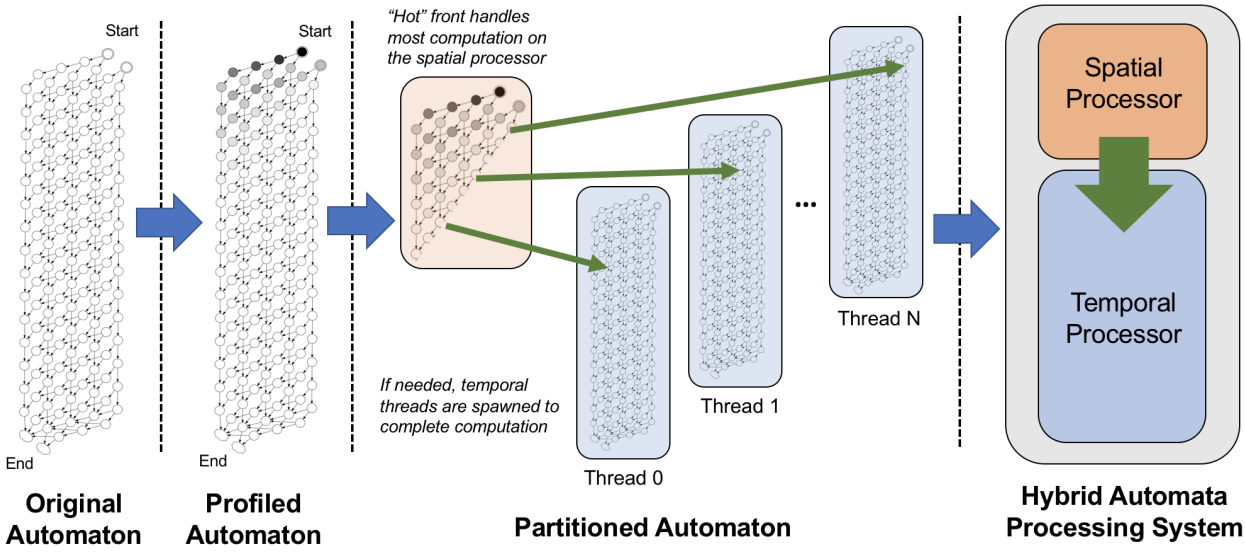

Figure 8.1: Hybrid spatial/temporal architectures can efficiently process filter-style automata. Dynamic profiling identifies highly-active, "hot" regions of automata. Hot regions are placed-and-routed on the spatial processor, and handle a majority of the computation. On occasion, if the automata transitions into the cold region, a thread is spawned on the temporal processor to complete computation.

Benefits of Hybrid Automata Processing: Depending on automata topology, dynamic behavior, and available spatial resources, the benefits of hybrid automata processing are the following:

- **A reduction of spatial architecture requirements:** If 5% of automata states are required to compute 99.9999% of total work, a spatial core 20x smaller might be able to efficiently compute the same problem. This reduction in spatial resources might enable small, low-cost integrated spatial cores to compute problems that typically would only fit on massive, expensive off-chip co-processors.
- **Increased capacity:** If a large spatial architecture is available, offloading 95% of automata states could make room to process $20x$ more automata states, allowing for parallel instances of the same automata or the computation of larger problem sizes. For many large automata applications, spatial architecture capacity is directly tied to performance [16, 11, 8].
- **Shorter place-and-route times:** Large graphs that utilize a significant percentage of spatial resources can have extremely long place-and-route times. Offloading large proportions of the automaton graph might lead to drastically shorter compile times. Place-and-route can be a large bottleneck for automata processing in network security, where zero day exploits need to be patched as quickly as possible.

To evaluate the potential of hybrid automata processing, this chapter presents the following contributions:

- A profile-driven partitioning algorithm that identifies regions of automata graphs that are responsible for a large proportion of computation.

- A characterization of automata behavior in the ANMLZoo benchmark suite. We show that most automata in the benchmark suite are filter-style automata, and contain large proportions of activity in a small proportion of states.
- A realization of a hybrid automata processing system targeting a commercially available hybrid CPU/FPGA architecture. We leverage existing, open-source spatial and von Neumann automata processing engines to compose our hybrid system.
- Experiments showing hybrid automata processing enables large reductions in spatial resource requirements (up to 97%) and compile times (up to 3.6x), with low performance overheads (¡6.1%).
- A feasibility study showing how hybrid architectures can use different algorithms to cooperate on one application kernel. The spatial architecture can use small, easily routable automata to pre-filter problem spaces and pass smaller problem sizes to non-automata-based CPU or GPU algorithms.

8.2 Background

8.2.1 Automata Processing

Informally, a finite automaton is defined as a directed graph of state nodes with transition rules between states guided by a globally visible input symbol. Each automaton has one or more start states that initiate computation. States that are currently performing computation are said to be *enabled*. For each input symbol, enabled states compare the current symbol on the input tape with their transition rule. If the symbol triggers the transition rule, the state *activates*, enabling all of its children, so they compute on the next cycle. Each automaton also has one or more *report* states. If a report state activates, the ID of the report state and the current position in the input symbol tape are recorded. Automata are usually designed to report when a specified pattern is seen on the input tape after following a series of state transitions.

8.2.2 Temporal Automata Processing

On von Neumann or "temporal" architectures [97], automata are usually computed using some form of transition table stored in main memory. Each enabled state is considered in a loop. If an enabled state matches the current input symbol, child states in the graph are fetched via the transition table and enabled. Because of the possibly unpredictable and large number of memory accesses involved in this algorithm, automata processing is usually bottlenecked by the memory system. Thus, the runtime of automata processing on these systems is highly correlated with the average number of active states–which determines required

memory bandwidth–and the size of the set of frequently visited states–which determines the size of the cache or scratchpad required to efficiently serve all memory requests [22]. Therefore, most high-performance, temporal processing systems rely heavily on optimizations, shortcuts, or new architectures that reduce the total number of memory accesses, or increase the efficiency of memory accesses [43, 44, 51].

8.2.3 Spatial Automata Processing

On spatial architectures (i.e. architectures with arrays of reconfigurable processing elements such as FPGAs) automata graphs are placed-and-routed in a reconfigurable fabric—similar to how logic gates in a circuit graph are placed-and-routed on an FPGA. Input symbols are broadcast simultaneously to all states in the graph, and all states compute and communicate in parallel within a single cycle. For highly active automata, spatial architectures can be several orders of magnitude faster than temporal architectures [22].

However, spatial architectures have limited capacity and routing resources. If an automaton has more states than are supported by the architecture, or if the automaton graph topology is too complex to be routable, the graph cannot be computed, or the graph must be partitioned and executed by reconfiguring the architecture to process each partition, and execute multiple passes of the input stimulus [16].

Prior work has explored automata on general purpose reconfigurable hardware [32, 33, 45, 29, 53]. Micron's Automata Processor [34] is an "automata specific" spatial architecture that uses a DRAM-based reconfigurable fabric to gain increased state density over more general-purpose reconfigurable fabrics [34, 35].

8.2.4 Hybrid Spatial/Temporal Architectures

While processors coupled with reconfigurable logic are now commonplace, high-performance, single-package hybrid FPGA/CPU systems have only recently become available. Intel has developed the Xeon+FPGA platform: a hybrid CPU/FPGA platform that marries a server-class, Xeon CPU with an Arria 10 FPGA on the same package [3]. This system is the first to combine high-performance CPUs with large-capacity FPGA fabrics. Closely-coupled spatial architectures allow lower latency and lower power communication, and the potential for a variety of on-package specialized accelerators. This system is an ideal platform to prototype hybrid automata processing as it provides high-performance CPU cores and a high-capacity FPGA fabric with an easy to program, high-throughput inter-processor communication architecture. In this chapter, we run temporal automata processing on real hardware, and functionally simulate the spatial architecture in order to estimate overheads of automata partitioning in a real system.

8.3 Hybrid Processing Potential Study

We first attempt to determine the potential benefit of hybrid automata processing by profiling automata behavior and identifying whether a large number of automata states can be offloaded to a temporal processor with low overhead. If we offload too many states, the temporal processor will be responsible for too much computation, and bottleneck the system. This section describes the benchmark workloads, profiling methodology, our partitioning algorithm, and the results of our potential study.

8.3.1 Benchmark Workloads

We run all evaluations on automata processing workloads from the ANMLZoo automata benchmark suite [22]. ANMLZoo is made up of 14 automata benchmarks. Each benchmark is an automata graph coupled with an input stimulus. ANMLZoo is a mix of real application workloads, synthetic workloads, and purely synthetic micro-benchmarks. We only consider the 12 real and synthetic benchmarks for this evaluation as the synthetic micro-benchmarks do not attempt to mimic real behavior of automata applications in the wild.

8.3.2 Profiling Methodology

We use the Virtual Automata Simulator (VASim) [78] to simulate automata. Our profiling methodology tracks activity in the automata and records the number of times each automata state attempts a matching computation on a particular input. Automata states only need to perform a matching computation when they are enabled by a parent state. Thus, the number of cycles an automata state is *enabled* directly corresponds to how much useful work each state does. Once the work levels of each automata state are recorded, we store this information in a metadata file. Note that we can easily identify the total amount of work done in the entire automata graph by summing work done by each state. Furthermore, the proportion of work done by a state, or by a group of states, can then be easily calculated by comparing the ratio of work done by a state/states to the work done by all states.

8.3.3 Partitioning Algorithm

Our partitioning algorithm uses profiling metadata gathered according to the process outlined in Section 8.3.2 to partition automata into two parts (a spatial and temporal part) according to a threshold of desired work. As input, the algorithm takes an automata graph, profiling metadata, and a percent threshold of total work, and outputs a spatial partition that contains at most that proportion of total work.

The algorithm begins with the automata start states and encodes each candidate in a priority queue according to the amount of work done by each state. At each iteration, the state in the queue that did the

Input: A profiled automata A
Input: Work threshold $workThreshold$
Output: A set of states P that do $workThreshold$ work
$priorityQueue \leftarrow A.startStates$
$workTotal \leftarrow 0$
$P \leftarrow \emptyset$
`// Grow partition until work threshold is reached`
while $workTotal \leq workThreshold$ **do**
  $s \leftarrow priorityQueue.pop$
  $workTotal += s.workDone$
  if $workTotal \leq workThreshold$ **then**
    $P \leftarrow P \bigcup s$
    for $child \in s.children$ **do**
      $priorityQueue.push(child)$
    end
  else
    break
  end
end
`// Set all boundary states to report`
for $s \in P$ **do**
  for $child \in s.children$ **do**
    if $child \notin P$ **then**
      $A.reportStates = A.reportStates \bigcup s$
    end
  end
end
return P

Algorithm 5: Profile-based Automata Partitioning

most work (the top of the queue) is considered as a candidate to add to the spatial partition. The work accomplished by that state is added to a running total. If the running total falls below the threshold of total work desired for the spatial partition, the state is added to the partition, and its children are added to the priority queue for consideration. If the running total breaks the desired work threshold, the state is not added, and the algorithm ends.

Once a spatial partition is selected, the states that have edges that cross the partition boundary are converted to reporting states. A report in the spatial partition indicates either 1) that a pattern was found or 2) that the temporal processor needs to spawn a thread to complete computation. The second case is communication overhead introduced by partitioning. Its impact is evaluated in the fully system evaluation in Section 8.5.4

Furthermore, all edges that cross from the temporal partition to the spatial partition are deleted, enforcing feed forward behavior. This greatly simplifies system design, and allows for asynchronous computation between spatial and temporal cores. To preserve correctness of the system, we conservatively implement all reachable automata states on the temporal processor. Thus, if a signal generated in the temporal processor needs to "re-enter" the spatial partition, re-entry is disallowed, and the temporal processor is responsible for this computation. Figure 8.1 shows the direction of communication always travels from the spatial partition to the temporal partition, and that a temporal thread conservatively has access to all edges and states in the original automaton. In this way, the a temporal thread can always correctly process any arbitrary injected behavior. Figure 8.1 We leave evaluation of other partitioning algorithms and implementation of support for bi-directional communication for future work.

8.3.4 Results

We profile and partition the 12 non-microbenchmark automata from the ANMLZoo benchmark suite [22] using the VASim virtual automata simulator's [78] –profile option. We profile each benchmark using the 1MB standard ANMLZoo inputs with standard optimizations turned on, and with OR gates removed (-Ox). Figure 8.2 shows the percentage of states required to capture varying thresholds of total automata work using the partitioning algorithm described in Section 8.3.3.

Many applications have the property that a small number of states are responsible for a vast amount of the computation. For example, in Dotstar, only 2.5% of states need to be considered to capture 99.999999% of automata work. In Snort, a subset of the classic network-intrusion benchmark, and real-world motivator for regular expression acceleration [22], only 8.3% of states need to be considered to capture 99.999999% of

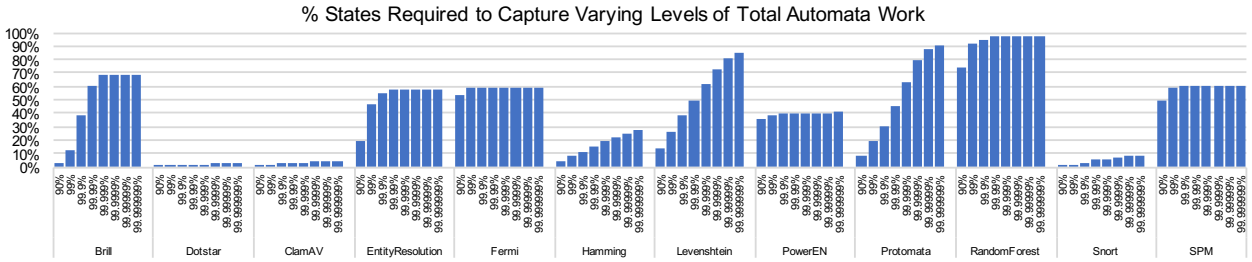

Figure 8.2: Percentage of states required to capture different levels of total work done by the automata. As we attempt to capture higher levels of total work in a partition, the number of states required increases. For some applications, very few states are required to capture a large percentage of total work; this indicates that large proportions of automata states could be offloaded to a temporal processor with low overhead.

automata work. This indicates these applications might gain huge benefits from offloading computation to a temporal processor in a hybrid system.

In some applications such as RandomForest, a large number of states (nearly 100%) must be considered to capture a large proportion of total work. In RandomForest, ensembles of decision trees are represented as a series of independent automaton loops. Loops are not amenable to partitioning because automata states are used more evenly, thus RandomForest most likely will not benefit from hybrid automata processing.

The above results indicate that impressive capacity reductions might be achieved from a hybrid-system implementation. If communication overhead is low, and CPU-based automata processing can efficiently handle offloaded computation, applications like Snort, ClamAV, and Dotstar, could reduce required automata states by 90%-99%. To measure the CPU performance on offloaded computation, and the impact of added communication overheads on the hybrid system performance, the following sections simulate hybrid system performance targeting Intel's Xeon+FPGA [3].

8.4 Hybrid Automata Processing System

This section describes our prototype implementation of a hybrid automata processor system. The high level overview of the hybrid system is shown in Figure 8.3. We chose to target our system architecture for Intel's Xeon+FPGA platform [3] described in Section 8.2.4. To the best of our knowledge, the Xeon+FPGA platform offers the highest-performance CPU and largest-capacity FPGA in a single package. We therefore target automata acceleration using an FPGA-based automata engine [53] and an Intel x86 CPU-based engine [43]. In order to efficiently communicate signals between the FPGA and CPU automata engines, we implement a version of the reporting architecture described in Wadden et al. [83]. In the remainder of this section, we describe each piece of the hybrid architecture and their implementation.

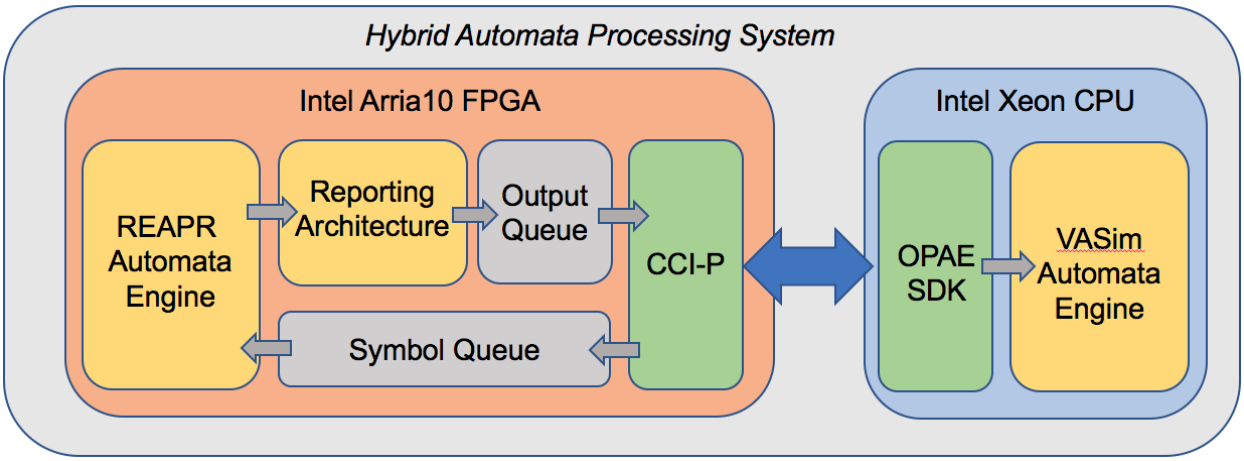

Figure 8.3: High-level overview of the proposed hybrid automata system. We target the Intel Xeon+FPGA platform [3]

While we chose the Xeon+FPGA product because of its high-performance characteristics, the benefit of hybrid automata processing should not be considered specific to this system. The partitioning algorithms presented in this chapter could be implemented on other spatial/temporal systems, or an application specific hybrid automata processor.

8.4.1 FPGA Automata Engine

For our spatial, FPGA automata engine, we choose the REAPR reconfigurable engine for automata processing [53]. While there is much prior work in FPGA-based automata processing, to our knowledge, REAPR is the first automata engine that focuses on generating both automata engines as well as a scalable I/O infrastructure for sending input symbol streams to the FPGA and collecting automata reporting results. However, REAPR's I/O infrastructure was designed using Xilinx's high-level synthesis toolchain and is specific to Xilinx systems. We therefore augment REAPR to have a standardized I/O protocol so that other platform specific I/O shims can be easily inserted.

REAPR Automata Engine

REAPR implements each automaton state as transition logic coupled with a state storage bit. This form of automaton, where transition logic is coupled with the state storage bit is called a *homogeneous* finite automaton [53]. Transition logic computes the transition rule of the state given an 8-bit input symbol. If the state storage bit is high, and the transition logic matches the current input symbol, a transition signal is propagated to connected child states.

REAPR implements transition logic using FPGA look-up-tables (LUTs), or using embedded BRAM columns. BRAM-based transition logic is usually easier to compile, but generates larger designs. We therefore choose to focus on LUT-based transition logic to accelerate design-space exploration of hybrid designs. Report states simply wire the result of their transition logic to a special reporting port. How these reports are combined and exported off-chip is discussed in Section 8.4.2

In this way, the automata implementation can be abstracted from the reporting architecture implementation, and platform specific I/O circuitry. This is extremely helpful when migrating automata engines between FPGAs of different vendors. We borrow the interface implemented in the REAPR automata engine and designate it as the Standard Automata Interface (SAI). The *symbol* input defines the 8-bit input symbol that drives computation. Computation can only occur if the *Run* signal is high. All automata transitions must occur during a single cycle of the *Clk* signal. If a reporting state matches, a corresponding signal in the *Reports* vector goes high during the cycle.

Ensuring that symbols are properly driven to the symbol bus is not included in this interface and should be implemented as a separate input queue. In this way, all symbol input and reporting output is abstracted from the automata functionality. I/O architecture will vary from vendor to vendor and from FPGA product to product and should therefore not be included in the SAI definition.

8.4.2 Reporting Architecture

While most prior spatial automata processing engines do not consider output reporting, REAPR at least evaluates a full system, including I/O for one automata benchmark in the ANMLZoo benchmark suite [53]. Input architectures for automata engines are usually fairly simple to design; no matter how many automata and states are implemented, the input requirement remains one input symbol per cycle. Output reporting architectures systems for automata processing can be much more difficult to design because of the diverse reporting needs of automata applications. For example, automata might require thousands of output ports. This requirement can greatly complicate reporting architecture design if the particular FPGA allows access to a much smaller number of wires of an output bus.

Design of the reporting architecture must also consider the reporting behavior of each automata application. If automata report frequently, but only a few automata report per cycle, small output packets are desirable. If application has many reports on a single cycle, large, bit-mapped output vectors are desired [83].

Reporting Architecture Overview

Wadden et al. [83] designed a run-time reconfigurable reporting architecture to satisfy a wide variety of automata reporting needs. We implement a version of this reporting architecture, with a 64-bit report vector width, and 64-bit metadata tag, as our communication architecture between FPGA and CPU sides of the hybrid architecture. Figure 8.4 shows an overview of this architecture.

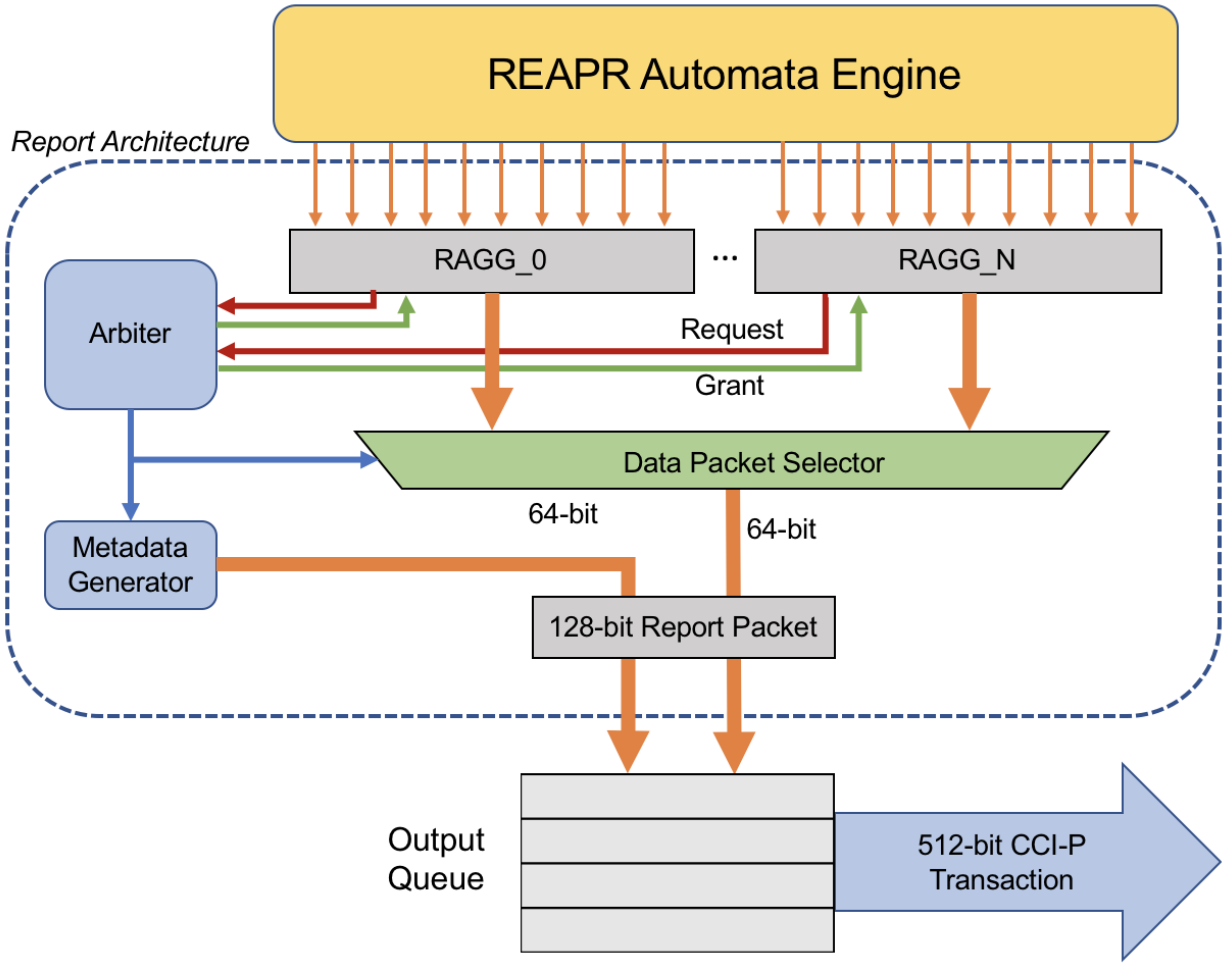

Figure 8.4: Report aggregators (RAGG) generate data packets whenever a report is generated in the automata engine. The arbiter stalls computation until all RAGGs are able to push their data packets to the report queue. A metadata tag is added to the data packet to identify when and where the packet was generated.

Our implementation requires that automata processing stall whenever a report is generated. Once the system stalls, a report packet is generated by each report aggregator (RAGG) where a report occurred. The arbiter then grants each report aggregator with a report packet access to the output queue to push its result. If the output queue becomes full, the reporting architecture is stalled, and a bus transaction is issued to send all data in the report queue to the CPU, incurring a single cycle penalty.

The architecture includes the overhead of a 64-bit metadata tag coupled with a 64-bit report packet (a 128-bit wide packet in total). The metadata provides 32 bits as a report index (i.e. what cycle the report occurred on) and 32 bits to encode the ID of the report aggregator and other dynamic information. 128 divides evenly into the 512-bit CCI-P bus discussed below. Thus, we are able to buffer 4 report packets before we need to issue a write transaction.

Other than the bus transaction cost, the reporting architecture is the only part of the architecture that can cause automata processing to stall. More reports will cause more stalls, and decrease overall system performance.

Intel CCI-P Interface

The above reporting architecture is general enough to be used on a variety of designs. However, for this implementation, we tailor all I/O to be compatible with the Intel Xeon+FPGA platform and I/O interface. The Xeon+FPGA product abstracts away much of the FPGA-to-CPU communication using a controller that implements the Cache Coherent Interface (CCI-P) protocol logic [98]. CCI-P abstracts the physical PCIe and QPI busses connecting the CPU and FPGA and offers a single interface–agnostic of the number of physical channels or protocols–to link hardware accelerators to the CPU. This improves the development time of accelerators and also ensures forward portability and inter-product portability of designs that adhere to this protocol. We implement and functionally verify a CCI-P controller for the REAPR automata engine in order to measure the hardware overhead of I/O interface hardware. We model performance of the CCI-P hardware according to the CCI-P interface documentation [98].

8.4.3 CPU Automata Engine

We use the Virtual Automata Simulator (VASim) [78] to execute automata on the CPU. VASim is an open-source automata simulation library that can optimize, manipulate, and simulate automata graphs. VASim is not the fastest CPU automata processing engine available, but it offers significant flexibility to support partitioning automata, and resuming automata processing. Furthermore, VASim performs well enough to test the potential benefits hybrid automata processing. Because VASim is not the highest performing automata engine, it should be considered a lower bound on the performance; future work might evaluate other, more sophisticated, higher-performance automata processing engines such as Intel's Hyperscan [43]. Faster automata engines would allow the system to offload of more states, or allow for smaller, lower-performance CPUs in a hybrid system, without bottlenecking system performance.

We require that the CPU-based automata engine be able to resume automata simulation in each instance where an automata transition crosses the partition boundary. We augment VASim to support this capability by allowing any number of states to be enabled on any cycle during simulation. If automata simulation reaches the boundary of the spatial partition, a report is generated and communicated from the FPGA to the CPU as a tuple consisting of 1) the location in the input stream where computation needs to be resumed, and 2) the state(s) where computation stopped. VASim resumes computation by enabling all children of the states where computation stopped, and considering the input symbol immediately following the reported index.

If VASim needs to process multiple activations that cross the partition, simulation proceeds, and new activity is injected when the proper index in the input symbol stream is reached. If, after processing all transitions in the automata for a given symbol, there is no activity, computation is fast-forwarded to the next available new activity. This fast-forwarding allows VASim to skip large parts of the input stream, greatly improving its performance over naively considering each input symbol. Performance of VASim when processing offloaded behavior ultimately depends on how often activity needs to cross the partition boundary, and how much offloaded automata computation needs to be accomplished once automata processing is resumed.

Note that CPU automata simulation is asynchronous with respect to FPGA simulation. The FPGA and CPU simulations do not need to be synchronized because we only consider automata computation that does not involve any conjunctive operations (e.g. AND gates). Thus, simulation in one part of an automaton can never impact the results of simulation in another part of the automaton. VASim conservatively has access to the entire automata, so that any arbitrary behavior can be simulated without consulting the spatial processor.

8.5 Hybrid System Evaluation

8.5.1 Profiling and Partitioning Methodology

ANMLZoo benchmarks do not provide multiple inputs. To avoid training and testing automata on the same data, we therefore use the Pareto principal and divide the 10MB ANMLZoo inputs into separate chunks for training and testing. For all evaluations in this section, we use the first 8MB of ANMLZoo input for profiling and partitioning according to various levels of automata work. We then use the last 2MB for system testing and evaluation.

When profiling automata, we also need to be careful not to over-train. If a match occurs in the input, it may bias the profiling such that entire rules are included in the partition. We assume that the benchmark inputs

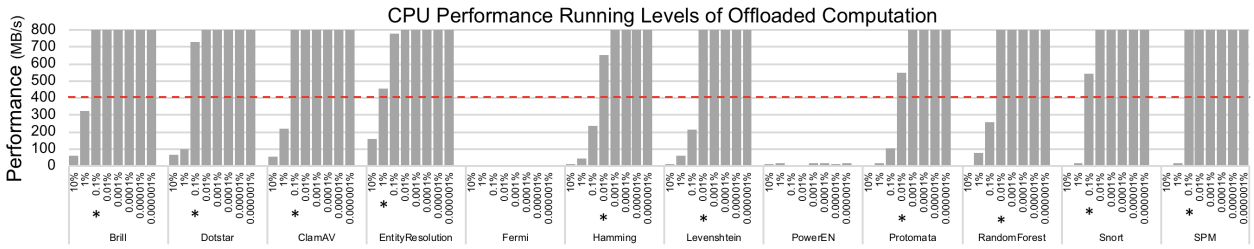

Figure 8.5: CPU performance for offloaded computation as a function of percent of total work done by the CPU. The dotted line demarcates the best possible performance of the REAPR engine on the target FPGA system. We assume if the CPU performs better than this upper bound it will not bottleneck computation. The partition that is able to offload the most states without bottlenecking computation is marked with a $*$. These "featured" partitions are used to report offloading potential in Figure 8.6.

are representative of real-world behavior; however, when training real applications rather than benchmarks, care should be taken to provide inputs such that average-case behavior for a use case is represented, perhaps using multiple, diverse inputs.

8.5.2 CPU Performance on Offloaded Computation

Offloading states from the spatial processor to the CPU co-processor might reduce the resource pressure on the spatial architecture. However, if too many states are offloaded, or if offloaded states do too much work, the CPU might bottleneck system performance. How much work is offloaded, and how efficiently the CPU processes this work, is tantamount to total system performance and will be application- and input-dependent. In order to understand how CPU performance changes as the the partition changes for different applications, we measure the performance of the CPU processing all offloaded computation.

Experimental Methodology

We first profile and partition all ANMLZoo benchmarks using the methodology described in Section 8.5.1 partitioning automata with various work thresholds (90%, 99%, 99.9% etc...). The partitioning algorithm generates a spatial partition that contains each work threshold. We then simulate each spatial partition using the 2MB testing input as if it were executed on the FPGA. We track all reports that would be emitted from the FPGA and labels them as either true reports, or as activity that needs to be offloaded the CPU. True reports are easy to distinguish from offloaded activity because true reports come from elements that would have reported in the original, unpartitioned automata. We then then divide up the transitions among multiple threads and simulate the automata while injecting this activity at the proper index in the input stream. For this evaluation, we use VASim to compute automata on the CPU in the Xeon+FPGA system [3]. This particular CPU has 28 logical cores, 36MB of L3 cache, and runs at 2.4GHz. Because the CPU has 28

logical cores, we run offloaded computation using 28 separate VASim threads; each thread is responsible for resuming automata processing and simulating the resulting work generated by activity crossing the partition boundary. As input, we use the testing input multiplied by 4 times (8MB) to insure that start-up overheads such as thread-spawn can be amortized over a longer, more realistic input.

Results

Figure 8.5 shows the performance of the CPU running different percentages of total offloaded automata work. We also demarcate the best performance attainable by the REAPR FPGA automata engine in the Intel Xeon+FPGA system (400MB/s) as a dotted line. Once CPU performance of offloaded computation goes above this line, we have high confidence that the CPU will not bottleneck the hybrid system when executing the partition. In reality, REAPR performance on the FPGA will most likely be much lower than 400MB/s due to less than optimal operating frequency and communication overheads and so this boundary should be considered a conservative threshold.

Figure 8.5 shows that as the partition grows, and the CPU is responsible for less work, CPU performance increases. For Entity Resolution, the CPU must compute 1% of total work before we are confident it will not bottleneck system performance (marked with a $*$). Most other applications require the CPU to compute less work–0.1%-0.01%–before the CPU is able to safely keep up with best-case FPGA throughput. *To show the potential of hybrid processing, we pick one "featured" partition for each benchmark where we are able to offload a large proportion of automata states, with high confidence the CPU will not bottleneck computation.* These partitions are marked with a $*$ in Figure 8.5.

Figure 8.6 shows the percentage of states offloaded for each of our featured partitions. In most cases (Brill, Dotstar, ClamAV, Entity Resolution, Hamming, Levenshtein, Protomata, Snort, SPM) we are able to offload 50% of benchmark states to the CPU before the CPU bottlenecks computation. In three cases, (Dotstar, ClamAV, and Snort) we are able to offload greater than 97% of benchmark states. This massive reduction in states might greatly decrease required FPGA resources, and compilation times. Note that different partitions might be desirable depending on the ultimate system requirements and application. More states can be offloaded, but at the risk of being bottlenecked by CPU automata processing. We leave exploration of this trade-off for future work.

For Fermi and PowerEN, we could not find a partition where the CPU did not bottleneck the FPGA. For PowerEN, this is due to the fact that the testing inputs exercise automata in places that the training inputs did not. This example motivates future work on improving the training methodology to account for worst-case behavior where applications may have diverse, and have unpredictable behavior. Either a larger training set, or a statistical method might produce more stable partitions.

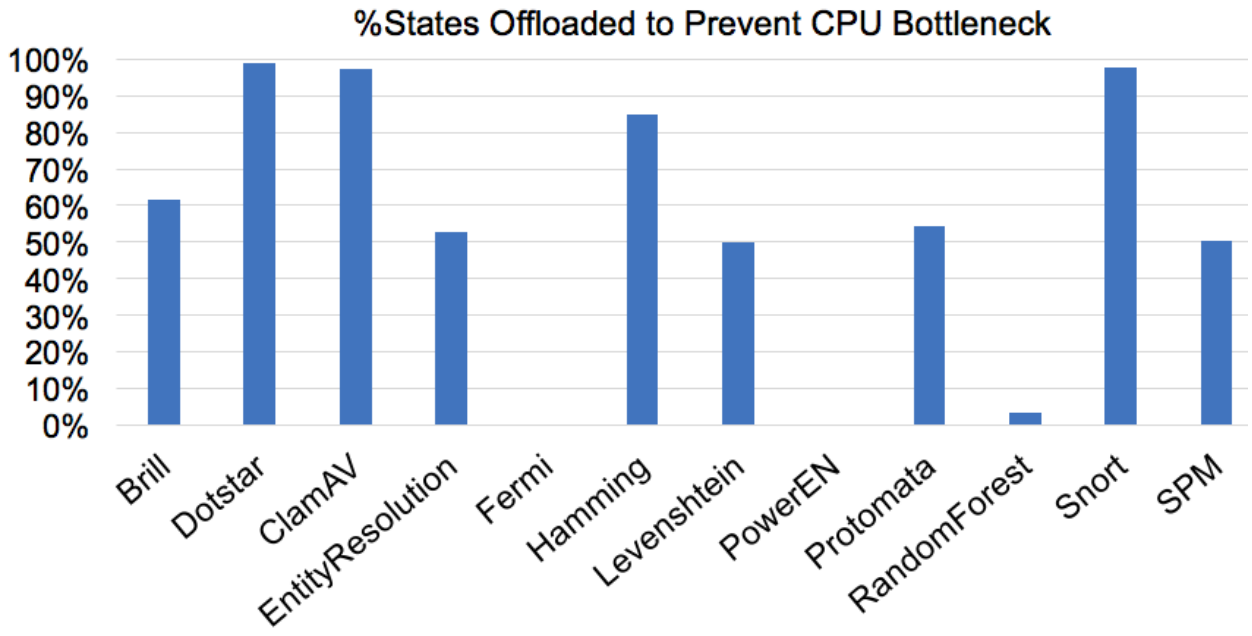

Figure 8.6: Percentage of states offloaded for each featured partition: the most states offloaded where the CPU in the Xeon+FPGA system does not bottleneck system performance. Many benchmarks can offload large proportions of states (up to 99%!) without overloading the CPU with work.

The following sections explore real-hardware space savings and compile-time reductions as a result of automata partitioning (excepting Fermi and PowerEN). We also simulate the impact of added communication overhead required to communicate offloaded computation to the CPU.

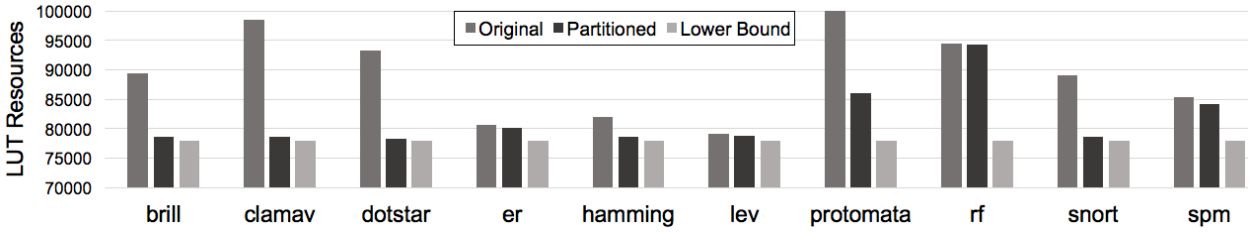

Figure 8.7: LUT resources consumed by original benchmark automata and featured partitions. "Lower bound" is a minimal kernel that approximates CCI-P interface logic overhead.

8.5.3 Spatial Resource Reduction

Place and Route Methodology

We demonstrate the significant potential benefits of automata partitioning by compiling each full ANMLZoo application as well as each corresponding featured automata partition highlighted in Figure 8.6. Automata are first converted to HDL using the REAPR toolchain [53]. Then, REAPR HDL is augmented with an

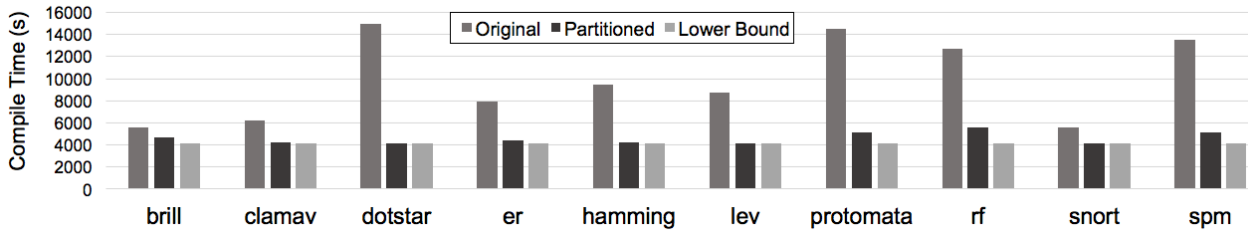

Figure 8.8: FPGA RTL compilation time of original automata and featured partitions. "Lower bound" approximates compilation time of CCI-P interface logic overhead. Some applications (ClamAV, Dotstar, ER, Hamming, Levenshtein, Snort) see massive reductions in compile times, approaching the lower bound. Even when a small number of LUT resources are offloaded, compile times can be much shorter (RF, SPM).

appropriately sized reporting architecture, and CCI-P I/O interface logic. We use Quartus Prime Pro 16.0 to compile all HDL and target the Intel Arria 10 FPGA device included in the target Xeon+FPGA system [3].

To identify the overhead of CCI-P interface logic, we compile an extremely small example accelerator with a CCI-P controller, and use this as an approximation of a non-automata, interface-logic baseline. This baseline represents the lower bound of both required LUT resources and compile time after eliminating all automata-processing related resources and is presented as *Lower Bound* in Figures 8.7 and 8.8.

Capacity Reduction

Figure 8.7 shows the FPGA LUT resources required to implement the systems before (Original) and after (Partitioned) partitioning. Most applications show a very large reduction in LUT resources. For example, our featured Snort partition was able to offload 97.8% of automata states, which corresponds to a 94.6% reduction in LUT resources when not accounting for interface logic. Brill, ClamAV, Dotstar, and Snort all show greater than 94% reduction in automata-related LUT resources. These savings are large, and the implications of these massive reductions in automata-related resources is discussed in Section 8.8.

RF is an example where very few states were able to be offloaded effectively (3.3%), resulting in a correspondingly small reduction in automata LUT resources of only 1.9%.

Compilation Time Reduction

Figure 8.8 shows the wall-clock runtime of FPGA RTL compilation for ANMLZoo benchmarks (Original) and our featured partitions (Partitioned). Our baseline (Lower Bound) is the compile time of the minimal CCI-P interface logic discussed in the previous section.

Six applications (ClamAV, Dotstar, ER, Hamming, Levenshtein, and Snort) reduce automata-related compile times by a substantial amount, over 94%. On average, automata-related compilation time is reduced

by over 90%. When partitioned, Dotstar's automata related compilation time is reduced by 99.8%, to within a fraction of the lower bound, and Dotstar's total end-to-end compilation time is reduced by 3.6x.

Interestingly, some benchmarks show a large reduction in compile times when they showed modest reductions in automata LUT requirements. For example, the Random Forest featured partition only used 1.9% fewer LUT resources than the full Random Forest benchmark, but required 73% less time to compile automata-related hardware and 56% less time to compile the full system. This is because a small reduction in automata states can generate automata partitions that are much easier to place and route. Random Forest automaton are loops of states. Upon inspection, the partitioning algorithm snipped these loops, transforming them into linear chains of states, greatly reducing the complexity of the place-and-route task, and thus greatly decreasing compile times.

Reduced compile times especially matter in the areas where "zero day" exploits occur, such as network intrusion detection (Snort) and virus detection (ClamAV). Compilation times are ideally on the order of a few minutes, rather than a few hours, and a reduction in compile times for these applications due to partitioning makes FPGA-based acceleration much more attractive.

8.5.4 Added Communication Overheads

While partitioning automata may allow us to offload a large number of "cold" states to the CPU, any added communication between the FPGA and CPU to initiate offloaded computation will cause additional report architecture stalls, and might increase total system runtime. Even if the CPU can keep up with offloaded computation, communication overhead might decrease performance, and eliminate much of the benefit of spatial acceleration. To understand this impact and potential trade-off, we simulate the communication overhead added due to partitioning for all ANMLZoo benchmarks compiled in Section 8.5.3.

Performance Model

We estimate the performance of the automata engine on the Intel Xeon+FPGA system using a cycle accurate simulation of our automata engine and reporting architecture. Automata processing processes all transitions from one symbol in parallel in one clock cycle, and is therefore trivial to simulate. However, the system must stall whenever output reporting events occur (one cycle per RAGG packet) and when a CCI-P transaction is required to empty the output report queue. We optimistically model each CCI-P transaction request with a single cycle penalty. In a real system, communication costs may be higher if there is frequent reporting and contention for the CCI-P bus, increasing average CCI-P transaction costs. Future work will explore system-specific performance costs of system I/O using this bus.

We implement automata symbol stream input on a separate CCI-P channel. Because symbol input has perfect locality, we assume that input can be easily overlapped with computation, and do not consider its cost in our simulation.

Results

We simulate the original automata and the featured partitioned automata for each benchmark and report the added communication overhead as a percentage of the performance of the original computation. Figure 8.9 shows the results of our experiment plotted with the percentage of offloaded states reported in Figure 8.6. Most applications show a small added overhead (at most 6.1%) indicating that communication overhead for these partitions does not cause significant performance degradation.

Our experiments show that communication overhead is low and automata partitioning can offer a trade-off between capacity and performance for filter-style automata. If a user can absorb a larger performance penalty, the partition could be reduced, increasing the number of offloaded states, but also increasing communication overhead. If a user requires the highest levels of performance, and can afford to compile extra states on the available FPGA, we could offload fewer states, reducing communication overhead.

Unexpectedly, ER and SPM show a *reduction* in communication overhead by 1.3% and 20.5% respectively. This is possible if the partition removes automata branches that ultimately generate multiple costly reports in the full automata simulation. As long as the CPU can keep up with the resulting branched computation, total spatial architecture performance can actually be improved by offloading computation.

8.6 Exploring Spatial Filtering for Non-Automata-Based Algorithms

When partitioning automata graphs across FPGA and CPU architectures in a hybrid system, we previously required that the CPU simulate the remainder of the automata. However, the CPU does not necessarily need to perform automata processing to identify patterns. Existing CPU algorithms can use small spatial automata filters to provide hints to where patterns might lie in the input stream and then use a different, preferable algorithm (i.e. one with higher performance, additional capabilities, or smaller memory footprint) to compute a final result.

To explore the potential for small automata filters to provide hints to non-automata-based algorithms, we present an example case study using the Levenshtein edit distance kernel [11].

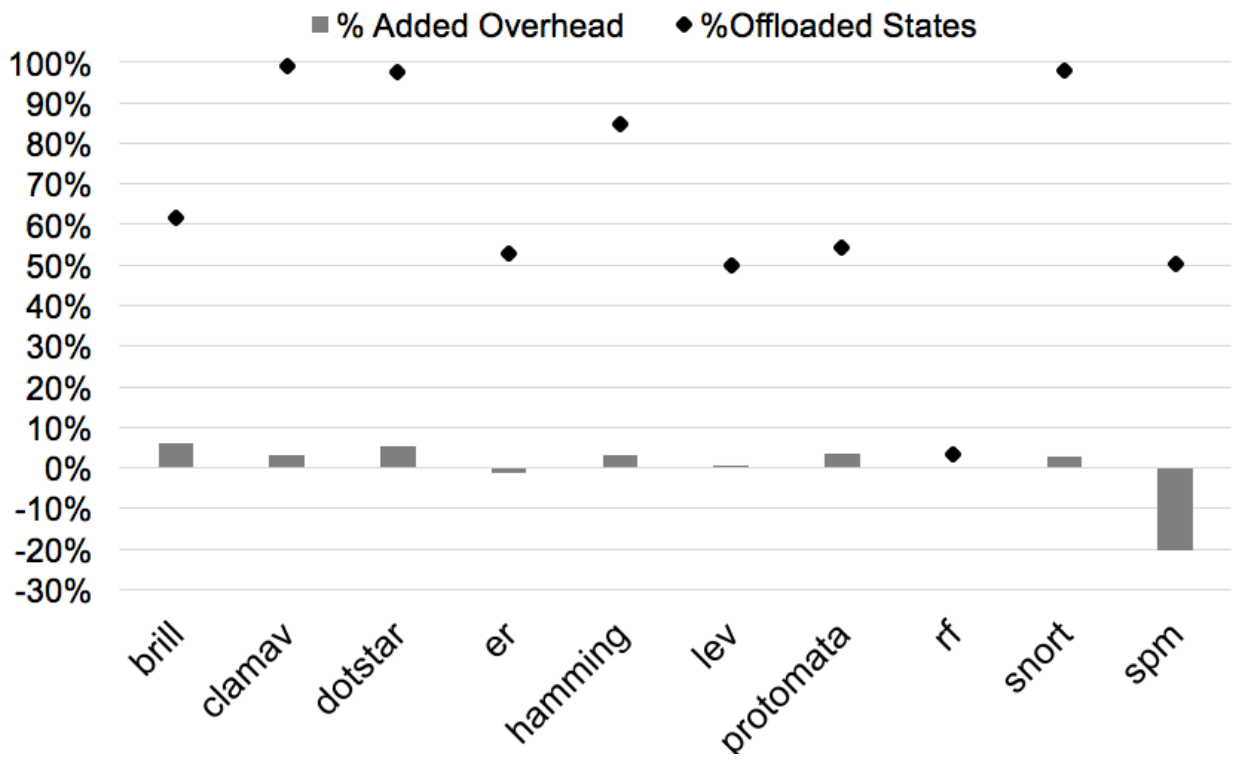

Figure 8.9: Communication overhead added by offloading computation to the CPU for each featured partition. Overhead is reported as a percentage of the original, full runtime. The percentage of states offloaded for each featured partition is also reported. Some applications (ClamAV, Dotstar, Snort) show minimal added overheads while offloading greater than 97% of states.

8.6.1 Levenshtein Edit Distance

Edit distance (or Levenshtein distance) is an approximate string matching metric used in linguistics, bioinformatics, and machine learning domains. Edit distance algorithms compare a pattern string to a reference string and indicate where the pattern string is similar to the reference string.

Edit distance can be computed using automata processing using Levenshtein automata. When implemented on spatial processors, Levenshtein automata have high performance [11], however, Levenshtein automata have a complex topology, and can suffer from routing and capacity constraints on spatial architectures when pattern strings are large. For example, Wadden et al. showed that Levenshtein automata that calculate a maximum edit distance of 5 could not be routed on Micron's Automata Processor when the pattern string length was greater than 24 [22]. Levenshtein automata often need to encode much larger pattern strings. Thus, capacity and routing constraints of spatial architectures greatly limit their usefulness when considering large, real-world problem sizes.

On CPUs, the parallel memory operations and large memory footprint of Levenshtein automata graphs can make them especially challenging to compute efficiently [22]. Because of these challenges, the edit

distance kernel is usually computed using two-dimensional dynamic programming techniques [99]. This style of algorithm performs well when small pattern strings need to be identified in relatively small reference strings, but performs poorly when many pattern strings need to be found within a large, streaming reference string.

Using the intuition gained in previous sections, we propose a spatial automata filtering technique, where small, easily routable Levenshtein automata are used to greatly filter the possible locations of pattern strings within a large reference string. Then, a dynamic programming algorithm implemented on a general purpose processor computes the final expensive pattern search without considering all possible locations in the large reference string. This technique is similar to how some long string alignment algorithms use seeds to filter large reference inputs [100] but does not require the generation of a seed index, and uses a more capable edit distance metric (rather than exact match or hamming distance metrics) to identify seed locations. Application specific integrated circuits have been developed to accelerate string scoring algorithms for bio-informatics sequence alignment [101]. Turakhia et al. developed a bio-informatics-specific accelerator capable of out-performing EDLib[99]–a high-performance CPU-based Edit Distance scorer–by 24.4x when scoring 1,000 base pair DNA sequences.

8.6.2 Spatial Automata Filtering Feasibility Study

Methodology

We construct Levenshtein automata for a set of input patterns and a fixed edit distance. Instead of constructing full Levenshtein automata for each input pattern, we only build an automata to recognize the first N characters of each pattern with at most the same similarity (1 - (edit distance/N)). At runtime, these Levensthein filters run on the input, and report positions in the input stream where potential matches for the full pattern may exist. These positions are then passed to a CPU-based edit distance scorer (edlib [99]), tailored for fast edit distance computations on long strings. Without the spatial automata pre-filtering, the EDLib algorithm must consider matching every pattern string at every position in the input string.

Evaluation

We evaluate the potential for spatial automata filtering using bio-informatics inspired data sets. For the reference string, we use the first 10MB of the first human chromosome with ambiguous DNA base pairs removed. For the pattern strings, we consider 400 DNA strings or "reads" of 1,000 base pairs (bp) in length, generated by the Mason2 [102] read simulator. We then use EDLib [99]–a state-of-the-art edit distance string scorer–to find all locations where the pattern strings occur in the reference within an edit distance of 100 to represent a desired string similarity of approximately 90% [99]. Without filtering, EDLib must compute the

edit distance of every pattern string anchored at every location in the 10MB reference for each of 400, 1,000 bp DNA patterns.

To help prune the search space for EDLib, we build 400 Levenshtein automata of length 22–corresponding to the first 22 base-pairs of each 1,000 bp read. Each Levenshtein automata is configured to match within an edit distance of 4. Compiling a full Levenshtein automaton for 1,000 length pattern strings with an edit distance of 100 would be impractical. The automata would require 72,320,000 states and would not fit into even the largest commercially available FPGAs. Using heuristics developed from profiling results, we chose 22 length, 4 edit distance filters as an appropriate size to filter out a large proportion of candidate matches, while also greatly reducing the probability of false negatives. These automata require only 55,462 states to implement–1,303x reduction in spatial requirements.

We compile our short automata filters for the Xeon+FPGA platform using the methodology outlined in Section 8.5.3. The filters required 119,479 LUT resources to compile, representing approximately 28% of the available 427,200 LUT resources in the Xeon+FPGA Arria10 device. We then simulate our filters using the methodology outlined in Section8.5.4. Simulation indicates that the FPGA is able to pre-filter the 10MB input with a throughput of about 196MB/s.

We use VASim to simulate the 22-bp Levenshtein filters on our 10MB input and guide VASim to generate report traces. The 400, 22-bp spatial filters generated 190,017 matches on 150,005 unique cycles our of 10,000,000. These matches can be thought of as "hints" to where edlib should search for a potential pattern match, and which pattern strings it should attempt to align. We then feed these hints to EDLib and measure its performance when considering only these candidates. Table 8.1 shows the results of our experiment.

EDLib Perf (KB/s)	**Filtered Perf. (KB/s)**	**Speedup**
34.3	506.4	14.78x

Table 8.1: EDLib speedup when candidate search locations are first pruned by an automata filter.

When the 10MB input is first pruned by a spatial automata filter on FPGA, EDLib is accelerated by 14.78x. This feasibility study shows that our automata partitioning intuition can be extended to accelerate applications that perform non-automata-based computation. Future work could explore other algorithms, other than Levenshtein Edit Distance, that might benefit from spatial automata filtering.

8.7 Related Work

Prior work in automata processing has used automata sub-problems to pre-filter large input streams to dramatically reduce the amount of work required to solve the problem on sequential automata runs, or on the CPU or GPU. Wang et al. [16] implemented a multi-pass, pruning approach to accelerate sequential pattern mining. By exploiting the downward closure property–that each sequential pattern of size N must include the

sequential pattern of size N-1–multiple passes can be used to make the problem tractable on capacity limited spatial automata processing architectures.

Prior work has also explored application-specific accelerators for automata processing [45, 28, 50, 51, 81, 82]. All of these architectures are constrained in some way by the number of states they can process at any one time. The work presented in this chapter offers an analysis to improve the capacity of any processor that accelerates these patterns in the presence of a CPU co-processor. Because CPUs are usually omni-present in systems that include specialized accelerators, we expect any realization of these systems to be able to take advantage of hybrid automata processing.

Prior work has also exploited automata partitioning to reduce spatial resource requirements on FPGAs in hybrid systems. Sidler et al. [96] developed a database query processor that included automata acceleration processing units (PU). Each PU is capable of processing a fully connected automata with up to four tokens, where each token can be a single character or an exact string match. For each PU, the largest number of states evaluated was 32, and the largest number of characters was 64. Because the number of states and characters are small, Sidler et al. statically partitioned larger regular expressions, executing regular expression terms that occurred after a *".*"* term on the CPU. This work is designed to accelerate small regular expressions in database queries and is not designed as a general-purpose automata accelerator. Our system can account for automata with hundreds of thousands of states, arbitrary character sets, and complex connectivity. We also dynamically profile a diverse set of automata and intelligently partition using profiled results rather than a static partition, to unlock offloading opportunities that might not be obvious to the developer.

Future work might leverage the lessons learned in this chapter to combine prior spatial accelerators (AP [34], Cache Automaton [82]) with prior temporal accelerators (UAP [51]).

8.8 Conclusions

This chapter presents a new automata processing paradigm that partitions automata across hybrid spatial/temporal architectures. We first observe that many automata-based applications behave as filters. Filter automata have a small set of "hot" automata states that do a large majority of the computation (dense computation), while a large proportion of "cold" states sit idle (sparse computation). We propose automata processing in a hybrid system where dense computation is executed in a spatial accelerator, and sparse computation is executed on a von Neumann or "temporal" processor.

We first profile automata benchmarks from the ANMLZoo benchmark suite and find that most applications do a vast majority of computation in a small fraction of automata states. For example, the Dotstar benchmark accomplishes 99.99% of total work using only 2.7% of automata states. We propose to offload the remaining

states to the CPU, reducing spatial resource requirements by up to 35x. Where prior automata applications were limited by spatial architecture capacity, this technique might enable acceleration of new problem sizes or large gains in performance.

We target the Xeon+FPGA platform [3] and develop a partitioning algorithm that profiles automata and only offloads states if the CPU can efficiently perform offloaded computation and not bottleneck computation. We then place and route these partitions to reveal large capacity and compile time reductions. Using our partitioning methodology we can realize up to 98% reduction in automata-related hardware resources and 78% improvement in end-to-end FPGA compile times with a small added performance overhead in some cases (¡6.1%).

These experiments show that automata partitioning and hybrid automata processing offer a novel trade-off between system performance, and spatial resource requirements and compile times. Users are now able to trade a small performance overhead for a large reduction in required spatial architecture resources. This reduction in spatial resources can enable more computation on a given spatial fabric, allow existing computation to fit into smaller integrated fabrics, and greatly reduce compile times. One exciting new area of research where partitioning could make a large impact is in automata processing overlays [88]. Automata overlays implement automata-specific spatial architectures on top of existing general-purpose FPGA fabrics. Overlays offer greatly reduced compile and reconfiguration times when compared to FPGAs, because of the raised abstraction of processing elements. Overlays also offer flexible dynamic reconfigurability. However, because of their "network in network" implementation, overlays suffer from a large hardware overhead per state, and reduced capacities compared to application-specific reconfigurable fabrics [88]. Hybrid automata partitioning could enable practical uses of automata-processing overlays by greatly reducing the spatial resource requirements.

Hybrid automata processing is a general technique, and leverages the best uses of spatial and temporal architectures. Future automata processing systems should should include accelerator cores designed for both sparse and dense automata processing in the same architecture to best accelerate this domain.

Chapter 9

Conclusions

9.1 Dissertation Summary

This dissertation focuses on the design and evaluation of accelerators for automata processing. Automata processing is an important kernel in networking for deep packet inspection [45, 6], but also has a wide range of use cases in bio-informatics [11, 12, 93, 13], data mining [15, 16, 17], natural language processing [19, 20], machine learning [8, 9, 10], and other application domains [21]. By building specialized accelerators for automata processing, many important kernels can be made faster, and more power efficient. Furthermore, kernels that can be implemented using automata-based algorithms can benefit from these accelerators.

This dissertation first proposes a new application domain that could benefit from accelerated automata processing: pseudo-random number generation [36]. We describe algorithms and methodologies to simulate Markov Chains using classic non-deterministic finite automata, and use Markov Chains to generate high-quality, high-throughput, pseudo-random number streams. We show that following reasonable technology scaling trends, spatial automata processors are up to $6.8x$ more power efficient than GPU-based techniques in the literature. We also show that other applications can be framed as Markov chains–such as agent-based modelling and random walks–and could benefit from accelerated automata processing.

This dissertation then identifies weaknesses in the existing toolchains to conduct automata processing research. In order to design and evaluate high-impact specialized accelerators, proper toolchains need to exist to identify bottlenecks, and inefficiencies in existing architectures and accelerators. These tools must be able to simulate, profile, analyze, and optimize applications, and must be flexible enough to answer new research questions. We present VASim, an open-source automata processing framework for automata processing research [78]. VASim is a multi-purpose software library that offers an object oriented view of automata states.

Using this object oriented view and associated data structures, VASim can simulate automata, optimize automata, transform automata, and be used to generate automata. At the time of writing, VASim is known to be in active use by over 5 universities, and has contributed to research in three top-tier computer architecture conferences.

This dissertation then identifies a lack of standardized benchmarks for fair evaluation of automata processing accelerators. In any area of acceleration, fair and standardized benchmarking techniques are necessary to support proper conclusions from experimental results. Prior benchmarks were either well motivated but small in number, or synthetically generated (at worst, completely arbitrary). We present ANMLZoo, a diverse, spatially-standardized set of 14 automata benchmarks for automata processing research [22]. Each benchmark is normalized to the capacity of one Micron AP D480 chip. Thus, comparisons using these benchmarks can trivially compare to the best performance of this unreleased architecture. Using this benchmark suite, we show that while GPUs outperform CPUs by leveraging latency hiding aspects of the GPU's memory architecture, spatial architectures generally perform orders of magnitude better than von Neumann architectures when computing real-world automata. However, spatial architecture capacity is very sensitive to the size and topology of automata.

This dissertation then identifies a lack of tools for design space exploration of spatial architectures. Spatial architectures are reconfigurable networks of connected processing elements. Because these networks rely on mapping application task-graphs to architectural resources in space, special place-and-route algorithms are required to implement. High-quality place-and-route tools are important to identify bottlenecks in spatial architecture resources such as processing elements and routing resources, and compare designs that trade-off capabilities and number of each resource. We present Automata-to-Routing, an automata-processing specific spatial architecture design-space exploration tool [37]. Automata-to-Routing (ATR) leverages VASim, ANMLZoo, and the versatile place and route tool (VPR) to enable the design of automata-specific spatial architectures. ATR also uses the sophisticated place-and-route algorithms in VPR to map automata graphs to the spatial resources on chip. We show that this toolchain is able to match the effectiveness of industry standard toolchains, modelling Micron's compiler toolchain within $2.5\% - 9.8\%$ accuracy for applications not bottlenecked by the AP's hierarchical routing matrix. This toolchain is also made up of completely open-source components, allowing much greater flexibility for research.

This base framework of a general purpose simulator, a benchmark suite, and a spatial architecture design and evaluation framework forms a foundation on top of which we can now perform architecture research. This dissertation identifies two areas of insight to build high-performance automata processing architectures.

The first insight is that output processing in spatial automata processing architectures can have a huge impact on system performance, and should be considered a first-class design constraint. We use the above

tools to characterize reporting behavior. We find that reporting behavior can be frequent and require high-throughput output architectures. We then develop and validate a trace-based simulation methodology to measure the impact of reporting on the performance of spatial automata processing architectures. We find that the impact can be very high in real systems (up to $46x$), and suggest designs that minimize the penalty of output processing in the common case [83]. Our new reporting architecture design is able to reduce reporting overheads by up to 84% and improve performance of applications that are heavily bottlenecked by reporting by up to $5.4x$.

The second insight is that spatial architectures and von Neumann architectures have complimentary advantages and disadvantages, and can be used together to gain a large percentage of the advantages of both. Spatial architectures can execute parallel automata behavior without performance degradation, but suffer from capacity and routing constraints. Von Neumann architectures can simulate practically arbitrarily large automata graphs, with complex topologies, but suffer from severe performance degradation with even modest amounts of automata activity. By implementing densely-active regions of automata on spatial cores, and sparsely-active regions of automata on von Neumann cores, we can reduce the resource pressure on the spatial architecture, while maintaining a large proportion of total performance. To evaluate this idea, we first dynamically profile automata behavior, and show that automata tend to have this lop-sided, dense/sparse behavior. For Snort, a subset of the classic network-intrusion benchmark, only 8.3% of states account for 99.999999% of total automata work. We then evaluate the performance of hybrid execution and show that large proportions of automata states can be offloaded (up to 98%), and executed on a von Neumann processor, while inducing small, acceptable communication overheads (up to 6%).

By reducing the pressure on spatial resource requirements, this technique could enable the execution of larger automata graphs, motivate the use of smaller reconfigurable fabrics, allow execution of larger applications on domain specific reconfigurable overlays [88], and greatly reduce place-and-route times.

9.2 Impact and Future Direction

9.2.1 Methodologies for Domain-Specific Accelerator Research

In order to conduct high-impact computer architecture research, a foundation of analysis tools, proper compilers, and fair benchmarks are all necessary prior to making scientific conclusions about computer architectures. When conducting CPU-based research, existing, high-quality tools are often taken for granted. Many simulators exist that excel at different science for various reasons; compilers and their optimizations for CPUs are also well established and mostly mature; benchmark suites are well studied and generally

agreed upon. Even GPUs and their compilers are beginning to reach this point of maturity. For any domain specific accelerator, or system that claims high-performance for a domain or subset of computations, these foundational tools must exist.

This dissertation followed a path that first built this foundation, and then used it to conduct domain specific accelerator research. When focusing on design of accelerators for other application domains, we can use this methodology as a guide:

1. Motivating applications must be identified that are important to society. Applications should be obviously bottlenecked by computing power. Relaxing these bottlenecks should show obvious benefit to society.

2. Tools to analyze the structure and behavior of these applications must be developed. "Structure and behavior" are not well defined, and will vary depending on the best available underlying algorithms. Such structure and behavior could be as high-level as a profiled task graph, as generated in Chapter 8. For automata, this task-graph is the automata graph. For the deep neural network domain, this graph might be as high-level as a hidden layer of the network.

3. A diverse set of benchmark applications must be identified to provide consensus about bottlenecks and roadblocks to further acceleration. These benchmarks should be designed to allow easy comparison between different approaches to solving the problem. Benchmarks should serve to fairly measure improvement over all existing approaches.

4. New tools must be developed to allow for design space exploration of new architectures–both spatial and von Neumann, or a combination of both–in order to identify optimal approaches. For CPUs and GPUs, these tools are usually parameterizable cycle accurate simulators. For FPGAs, these tools parametrically model reconfigurable networks [38].

5. Once bottlenecks are identified, and tools exist to conduct design-space exploration, new configurations, or brand new architectures should be proposed that relax these bottlenecks.

A visualization of this methodology, showing how each step supports the next, is shown in Figure 9.1.

9.2.2 Designing Effective, Cross-Domain Tools

The above process can be extremely costly. Building compilers, benchmarks, simulators, and other design-space exploration tools takes years (even decades), and are continually re-evaluated to adjust to changing assumptions. As we rely on more and more on specialized accelerators for new application domains to

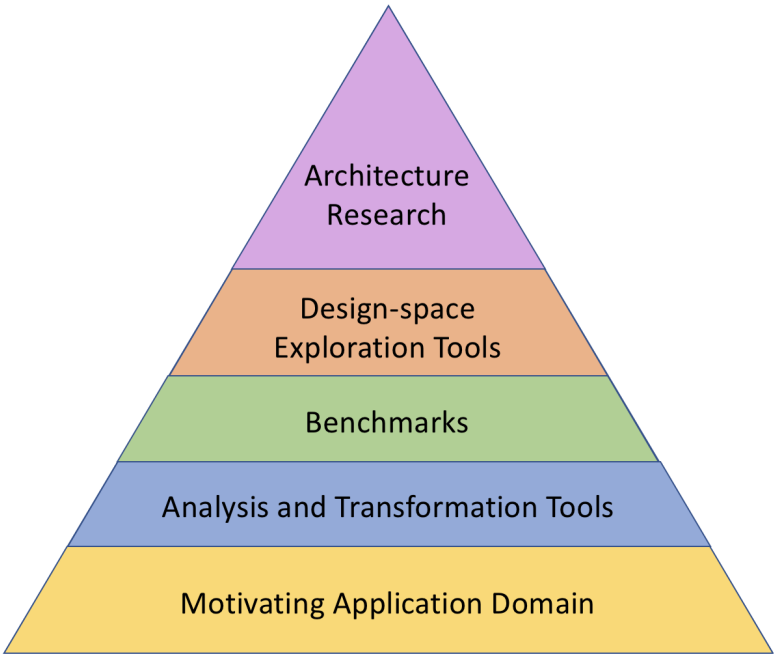

Figure 9.1: The architecture research pyramid. 1) important applications in a particular domain motivate 2) development of analysis tools for this domain. 3) benchmarks must be generated to properly characterize a domain so that a proper consensus is reached after analysis. 4) design-space exploration tools must be developed to 5) to identify optimal design points for domain-specific architectures.

overcome the disadvantages of general purpose computers, we will need a large amounts of tools work in order to design and build effective computer architectures. Currently, the area of machine learning has enough interest and investment to warrant such work. Benchmark suites [103, 104], and compiler frameworks for task graphs [105, 106, 107], are being developed.

To reduce the burden on researchers developing domain specific tools, research should be conducted into generic tools that can serve multiple domains. An example of such a tool is the versatile place and route (VPR) tool [38] used to develop Automata-to-Routing in Chapter 6. VPR was originally designed to serve the FPGA design community. However, VPR is so flexible as a tool, it can support design-space exploration of spatial architectures with *arbitrary* processing elements. This intuition was first used to develop models for coarse-grained reconfigurable architectures for image processing [108]. We used this intuition as inspiration to develop a completely new pipeline for the automata processing domain [37].

Another example of such a tool is the Low-Level Virtual Machine or LLVM [109]. LLVM provides a low-level intermediate program representation to allow for portable, and high-performance optimizations for von Neumann computers. However, LLVM's low-level intermediate representation might be too low-level to be applicable to optimizations on coarser-grained task-graphs, and was not designed with spatial/dataflow style code generation in mind. Thus, room exists for a higher-level intermediate task graph language and compiler

framework to implement truly machine independent optimizations such as redundant code elimination (analogous to redundant state elimination in the VASim compiler). Such high-level approaches are well-known for CPU-based optimization (e.g. abstract syntax tree optimization) [110], but such have not been evaluated for domain-specific optimization re-use. It is highly likely that a novel abstraction layer will be desired.

9.2.3 Analyzing Spatial/Temporal Trade-offs in Computer Architectures

The design of domain-specific accelerators is motivated by the observation that no one computer architecture best serves every style of computation. Thus, architects have increasingly focused on specializing accelerators for important application domains to improve performance, and power efficiency over general purpose processors. Complicating the design choices of specialized accelerators is that there are often large trade-offs when making design choices for even a single domain. For example, Chapter 8 highlighted that no single architecture is "best" when designing even for the automata processing domain–an extremely restricted, and simple model of computation. Even within this simple model, computation and data access patterns can vary so widely within a single application, that neither spatial architectures nor von Neumann architectures perform best in isolation, and an architecture that is able to take advantage of the combination of the two paradigms is desirable.

Researchers have not yet formalized an approach to identifying when spatial vs. von Neumann approaches are best for computations. Prior spatial/dataflow architecture approaches for general purpose computing have argued that various new control paradigms are required to gain efficiency over von Neumann or SIMD-based approaches [97]. Parashar et al. [97] survey prior approaches to spatial/dataflow computing and their advantages, but this work did not expose intuitions about inherent properties or behaviors of algorithms that made these types of architectures to perform better than large vector/SIMD, or out-of-order von Neumann architectures.

Future work should attempt to identify fundamental intuitions behind when spatial, dataflow architectures are a better approach than temporal, von Neumann based architectures. In this dissertation, we were able to expose some of this intuition for automata processing. We show that spatial architectures excel when automata behavior in the graph is dense, and von Neumann architectures excel when automata behavior in the dataflow graph is sparse. In the future, this dissertation and these intuitions may lay the ground work for new analyses of the benefits of spatial vs. von Neumann architectures for other, domain-specific compute paradigms.

Bibliography

[1] Micron Inc. Designing for the Micron D480 Automata Processor. http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html.

[2] David R. Brown, Harold B. Noyes, Irene Junjuan Xu, and Paul Glendenning. Methods and systems for routing in a state machine, March 25 2014. US Patent 8,680,888.

[3] Algo-Logic. Intel xeon + fpga. http://algo-logic.com/Intel-Xeon-FPGA.

[4] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the USENIX Large Installation Systems Administration Conference (LISA)*, 1999.

[5] Vern Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[6] Indranil Roy. *Algorithmic Techniques for the Micron Automata Processor*. PhD thesis, Georgia Institute of Technology, 2015.

[7] ClamAV. ClamAV Rules. Available at https://www.clamav.net/.

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

[9] Mateja Putic, A.J. Varshneya, and Mircea Stan. Hierarchical Temporal Memory on the Automata Processor. In *Proceedings of the IEEE Micro Special Issue on Cognitive Architectures*. IEEE, 2016.

[10] Mateja Putic and Mircea Stan. Dendroplex: Synthesis, Simulation, and Validation of Hierarchical Temporal Memory on the Automata Processor. In *Proceedings of the Design Automation Conference (DAC)*, 2017.

[11] 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. *Proceedings of Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[12] Indanil Roy and Srinivas Aluru. Finding Motifs in Biological Sequences Using the Micron Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 415–424, 2014.

[13] 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 *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2018.

[14] Indranil Roy, N. Jammula, and Srinivas Aluru. Algorithmic Techniques for Solving Graph Problems on the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 283–292, May 2016.

[15] Ke Wang, Yanjun Qi, Jeffrey J Fox, Mircea Stan, and Kevin Skadron. Association rule mining with the Micron Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 689–699, 2015.

[16] 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)*, 2016.

[17] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. Frequent Subtree Mining on the Automata Processor: Challenges and Opportunities. In *Proceedings of the International Conference on Supercomputing (ICS)*, New York, NY, USA, 2017. ACM.

[18] Chunkun Bo, Ke Wang, Jeffrey J. Fox, and Kevin Skadron. Entity Resolution Acceleration using Micron's Automata Processor. *Proceedings of Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[19] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. Brill Tagging on the Micron Automata Processor. In *Proceedings of the IEEE International Conference on Semantic Computing (ICSC)*, pages 236–239, 2015.

[20] K. Zhou, J. Wadden, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 355–360, Oct 2015.

[21] 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.

[22] Jack Wadden, Vinh Dang, Nathan Brunelle, Tom Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2017.

[23] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. iNFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[24] 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)*, pages 18:1–18:10, 2013.

[25] Xiaodong Yu and Michela Becchi. Exploring different automata representations for efficient regular expression matching on GPUs. In *ACM SIGPLAN Notices*, volume 48, pages 287–288, 2013.

[26] 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.

[27] Zachary K Baker, Hong-Jip Jung, and Viktor K Prasanna. Regular expression software deceleration for intrusion detection systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2006.

[28] Y. H. E. Yang and V. K. Prasanna. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *Proceedings of the IEEE International Conference on Computer Communications*, pages 1853–1861, 2011.

[29] Xiang Wang. Techniques for efficient regular expression matching across hardware architectures. Master's thesis, University of Missouri-Columbia, 2014.

[30] Michela Becchi. *Data structures, algorithms and architectures for efficient regular expression evaluation.* PhD thesis, Washington University in St. Louis, 2009.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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. In *Proceedings of the 2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 622–629, Oct 2016.

[37] 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.

[38] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[39] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology, 2006.

[40] J. M. Champarnaud. Subset construction complexity for homogeneous automata, position automata and ZPC-structures. *Theoretical Computer Science, Workshop on Implementing Automata '98*, 267(1):17 – 34, 2001.

[41] Pcre - perl compatible regular expressions. https://www.pcre.org/.

[42] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

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

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

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

[46] Michela Becchi and Patrick Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 1:1–1:12, 2007.

[47] Jan Holub. Bit parallelism - nfa simulation. In Bruce W. Watson and Derick Wood, editors, *Implementation and Application of Automata*, pages 149–160, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[48] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling pcre to fpga for accelerating snort ids. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 127–136, New York, NY, USA, 2007. ACM.

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

[50] Kubilay Atasu, Florian Doerfler, Jan van Lunteren, and Christoph Hagleitner. Hardware-accelerated regular expression matching with overlap handling on IBM PowerEN processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1254–1265, 2013.

[51] 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)*, pages 533–545, 2015.

[52] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 111–120, 2002.

[53] Ted Xie, Vinh Dang, Chunkun Bo, Jack Wadden, Kevin Skadron, and Mircea Stan. REAPR: Reconfigurable Engine for Automata Processing. In *Proceedings of the International Conference on Field Programmable Logic (FPL) to appear*. IEEE, 2017.

[54] Ted Xie. Reapr: Reconfigurable engine for automata processing. https://github.com/ted-xie/REAPR.

[55] Micron Automata Processor SDK. http://micronautomata.com/.

[56] Nicholas Metropolis. The beginning of the monte carlo method. *Los Alamos Science, No. 15*, 1987.

[57] Pierre L'Ecuyer and Richard Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Trans. Math. Softw.*, 33(4), August 2007.

[58] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[59] George Marsaglia. DIEHARD: a battery of tests of randomness. *See http://stat. fsu. edu/ geo/diehard. html*, 1996.

[60] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, James Dray, San Vo, Andrew Rukhin, Juan Soto, Miles Smid, Stefan Leigh, Mark Vangel, Alan Heckert, James Dray, and Lawrence E Bassham Iii. A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2001.

[61] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[62] Markus Manssen, Martin Weigel, and Alexander K Hartmann. Random number generators for massively parallel simulations on GPU. *The European Physical Journal-Special Topics*, 210(1):53–71, 2012.

[63] John K Salmon, Mark A Moraes, Ron O Dror, and David E Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2011.

[64] Hybrid memory cube specification 2.0. http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.0_Public.pdf.

[65] Sven Banisch. *Agent-Based Models as Markov Chains*, pages 35–55. Springer International Publishing, Cham, 2016.

[66] S.H. Rodger and T.W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett, 2006.

[67] Michela Becchi. Regular expression processor. http://regex.wustl.edu/index.php/Regular_Expression_Processor.

[68] X. Yu, B. Lin, and M. 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.

[69] Michela Becchi and Patrick Crowley. Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNext)*, pages 25:1–25:12, 2008.

[70] Andreas Malcher. Minimizing finite automata is computationally hard. *Theoretical Computer Science*, 327(3):375 – 390, 2004. Developments in Language Theory.

[71] Lucian Ilie and Sheng Yu. Algorithms for computing small nfas. In *Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*, MFCS '02, pages 328–340, London, UK, UK, 2002. Springer-Verlag.

[72] J.-M. Champarnaud and F. Coulon. Nfa reduction algorithms by means of regular inequalities. *Theoretical Computer Science*, 327(3):241 – 253, 2004. Developments in Language Theory.

[73] Lucian Ilie, Roberto Solis-Oba, and Sheng Yu. Reducing the size of nfas by using equivalences and preorders. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Combinatorial Pattern Matching*, pages 310–321, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[74] Golden G. Richard III and Vassil Roussev. Scalpel: A frugal, high performance file carver. In *Refereed Proceedings of the 5th Annual Digital Forensic Research Workshop (DFRWS'05)*, 01 2005.

[75] Flourian Buckholtz. The structure of a pkzip file. https://users.cs.jmu.edu/buchhofp/forensics/formats/pkzip.html.

[76] 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)*, pages 593–605, 2016.

[77] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz–open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484, 2001.

[78] 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.

[79] 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.

[80] Jack Wadden. Vasim: The virtual automata simulator. https://github.com/jackwadden/VASim.

[81] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 600–612, New York, NY, USA, 2017. ACM.

[82] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das. Cache automaton: Repurposing caches for automata processing. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 373–373, Sept 2017.

[83] Jack Wadden, Kevin Angstadt, and Kevin Skadron. Characterizing and mitigating output reporting bottlenecks in spatial-reconfigurable automata processing architectures. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2018.

[84] Michela Becchi, Mark Franklin, and Patrick Crowley. A workload for evaluating deep packet inspection architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 79–89. IEEE, 2008.

[85] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. *ACM SIGPLAN Notices*, 49(4):529–542, 2014.

[86] Sourangsu Banerji. Computer Simulation Codes for the Quine-McCluskey Method of Logic Minimization. *arXiv preprint arXiv:1404.3349*, 2014.

[87] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 216–225, 2011.

[88] R. Karakchi, L. Richars, and J. Bakos. A dynamically reconfigurable automata processor overlay. In *2017 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2017.

[89] 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.

[90] Jack Wadden. Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. https://github.com/jackwadden/ANMLZoo.

[91] A. Munoz, S. Sezer, D. Burns, and G. Douglas. An approach for unifying rule based deep packet inspection. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5, June 2011.

[92] Cisco and its Affiliates. Snort users manual 2.9.11. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/snort_manual.html.

[93] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. High Performance Pattern Matching Using the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1123–1132, 2016.

[94] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 110–119. IEEE, 2014.

[95] Krste Asanovi, 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.

[96] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 403–415, New York, NY, USA, 2017. ACM.

[97] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered instructions: A control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM.

[98] Intel. Intel cci: Core cache interface. https://01.org/sites/default/files/downloads/opae/cci-p-mpf-overview.pdf.

[99] Martin Šošić and Mile Šikić. Edlib: a c/c++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[100] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[101] Yatish Turakhia, Gill Bejerano, and William J Dally. Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213. ACM, 2018.

[102] Manuel Holtgrewe. Mason–a read simulator for second generation sequencing data. *Technical Report FU Berlin*, 2010.

[103] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference workloads for modern deep learning methods. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–10. IEEE, 2016.

[104] Baidu Research Sharan Narang. Deepbench. https://svail.github.io/DeepBench/.

[105] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[106] Richard Wei, Vikram Adve, and Lane Schwartz. Dlvm: A modern compiler infrastructure for deep learning. *arXiv preprint arXiv:1711.03016*, 2017.

[107] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel® ngraph. 2018.

[108] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. Evaluating programmable architectures for imaging and vision applications. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[109] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[110] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[111]

[112] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 1064–1072, May 2007.

[113] Eric E Schadt, Steve Turner, and Andrew Kasarskis. A window into third-generation sequencing. *Human molecular genetics*, 19(R2):R227–R240, 2010.

[114] Intel. Open programmable acceleration engine. https://opae.github.io/.

[115] Ben Langmead. Aligning short sequencing reads with bowtie. *Current protocols in bioinformatics*, pages 11–7, 2010.

[116] Geoff Langdale. HyperScan in Suricata: State of the Union. 2016. http://suricon.net/wp-content/uploads/2016/11/SuriCon2016_GeoffLangdale.pdf.

[117] Marvin Tom and Guy Lemieux. Logic block clustering of large designs for channel-width constrained FPGAs. In *Proceedings of the 42nd annual Design Automation Conference (DAC)*, pages 726–731. ACM, 2005.

[118] Tim Bray. The JavaScript object notation (JSON) data interchange format. 2014.

[119] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, Sept 1955.

[120] 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, 2012.

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

[122] 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 (PACT)*, pages 389–400, 2012.

[123] Titan IC Systems. RXP regular eXpression processor soft IP. http://titanicsystems.com/Products/Regular-eXpression-Processor-(RXP).

[124] H. D. Cheng and K. S. Fu. VLSI architectures for string matching and pattern matching. *Pattern Recognition*, 20(1):125–144, 1987.

[125] 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 (TVLSI)*, 12(7):727–734, 2004.

[126] 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.

[127] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of Architectures for Networking and Communications Systems (ANCS)*, pages 50–59, 2008.

[128] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, pages 155–164, 2007.

[129] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1):75–90, 2000.