AutomataZoo: A Modern Automata Processing Benchmark Suite

Conference Paper · September 2018

DOI: 10.1109/IISWC.2018.8573482

CITATIONS
18

READS
238

11 authors, including:

Jack Wadden
University of Michigan
31 PUBLICATIONS **315** CITATIONS
SEE PROFILE

Elaheh Sadredini
University of Virginia
31 PUBLICATIONS **356** CITATIONS
SEE PROFILE

Lingxi Wu
University of Virginia
4 PUBLICATIONS **37** CITATIONS
SEE PROFILE

Yizhou Wei
University of Virginia
5 PUBLICATIONS **96** CITATIONS
SEE PROFILE

AutomataZoo: A Modern Automata Processing Benchmark Suite

Jack Wadden, Tommy Tracy II, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei,
Jeffrey Udall, Matthew Wallace, Mircea Stan, Kevin Skadron
{wadden,tjt7a,es9bt,lw2ef,cb2yy,jd3wn,yw2bc,ju2c,mw3yg,mircea,skadron}@virginia.edu
University of Virginia, Charlottesville, VA 22903

***Abstract*—Automata Processing is an important kernel for many application domains, and is challenging to accelerate using general purpose, von Neumann computers. New research into accelerators for automata processing motivated the creation of ANMLZoo, a standardized automata benchmark suite that reflects the many modern use-cases for automata processing. While researchers have adopted ANMLZoo as the de-facto benchmark suite to measure improvements in automata processing, certain drawbacks have emerged after years of use. In this work, we first examine opportunities for improvement over ANMLZoo's benchmarking methodology. We then propose a new benchmarking methodology that aims to generate benchmarks better suited for modern automata processing research.**

We then present AutomataZoo, a new automata processing benchmark suite that uses our new benchmarking methodology. AutomataZoo is composed of 24 automata processing benchmarks from well-known domains, as well as from newly published and novel use-cases for automata processing. AutomataZoo benchmarks use open-source tools to generate full automata applications and provide standard, large input stimuli. Where arbitrary choice is involved, we design multiple benchmarks that vary important design parameters in order to help architects evaluate application-level trade-offs affecting performance/power and/or chip capacity. This paper shows the advantages of these new properties by performing previously difficult-to-perform experiments and comparisons.

I. Introduction

Automata processing is a computing model commonly used for identifying patterns in large data sets. Automata (also known as finite automata) are directed graphs of states with transition rules among states. Automata compute on streaming data by processing one byte at a time and making transitions between states in parallel according to transition rules. If an automaton enters into a final state, a pattern has been found in the input data, and is reported to the user.

Automata processing is important to accelerate because it is a central kernel in deep packet inspection (DPI) [1], [2], and virus [3] and malware detection [4]. Other, newer application domains have been discovered that benefit from accelerated automata processing including bioinformatics [5]–[8], natural language processing [9]–[11], data analytics and mining [12]–[15], simulation [16], and machine learning [17], [18].

However, automata processing is challenging to accelerate using traditional CPUs and GPUs because of the unstructured memory accesses involved in traversing automata graphs [19]. These unstructured memory accesses reduce the effectiveness of caches, and also make leveraging SIMD-style architectures difficult in general. To solve these issues, prior research has investigated using accelerators with large, fast scratchpads to accelerate automata traversal rule lookups [20], or implementing automata state-machines as circuits in reconfigurable hardware, such as an FPGA [21]. Micron's Automata Processor [22] is an automata-specific reconfigurable fabric implemented in DRAM, announced in 2013 but not yet productized.

In order to measure improvements in automata processing software engines and hardware accelerators, automata *benchmarks* are desirable. Automata benchmarks consist of a benchmark automaton graph (or graphs) and corresponding input stimulus that act as a proxy for real-world automata behavior and topology. Prior efforts to benchmark automata processing usually involved a subset of real-world automata rules such as the Snort DPI ruleset [1]. While this ruleset is an important benchmark, 1) it is not sufficient, and 2) other benchmarks are desired to provide a set of workloads that address the growing diversity of automata applications. To increase the number of benchmarks, researchers developed synthetic "DPI-like" automata/input generators [23]. In some cases, a lack of a standardized benchmarking methodology led to the construction and use of purely synthetic automata benchmarks that did not reflect diverse, real-world automata workloads [24].

To solve the lack of diversity with prior benchmarking methodologies, Wadden et al. developed ANMLZoo [19]. ANMLZoo is a collection of 13 real-world and synthetic automata processing benchmark suites from prior work. ANMLZoo is an easy-to-access and easy-to-use benchmark suite that reflects new diversity in automata applications. ANMLZoo uses a common automata format and open-source toolchain to standardize analysis and optimization of automata graphs. One novel feature of ANMLZoo is that each benchmark automata graph is "spatially standardized" to use all of the resources of one Micron D480 AP chip. This means that each ANMLZoo benchmark represents the approximate capability of this specialized architecture. Because the Micron D480 was not widely available, this feature allowed for easy, approximate comparisons between it and existing or novel automata processing accelerators[1]. For example, a hypothetical new architecture that computes one ANMLZoo benchmark

[1]For a more in depth description of automata processing, acceleration challenges, Micron's AP, and the ANMLZoo benchmark suite, we refer the reader to Wadden et al [19].

using 50% of its state resources has approximately 2x more capacity (and 2x performance with input parallelization [25]) than the AP.

As a result of the above features, ANMLZoo was quickly adopted and effectively used in a variety of explorations of automata processing accelerators [21], [25]–[28]. Each of these explorations was successfully able to compare to the capabilities of the Micron D480, without access to the Micron software SDK or compiler [29]. ANMLZoo has thus shown to be highly useful for automata processing acceleration research. However, as more researchers use the benchmark suite and the automata processing research space evolves, we have identified important drawbacks, which this work seeks to rectify.

Some of the highest impact drawbacks are listed below:

- **Promotes a now obsolete baseline:** Micron's D480 AP is not commercially available, and no longer reflects the best capacity and performance of modern spatial/reconfigurable automata accelerators.
- **Benchmarks inherited features of the Micron D480 architecture:** Because ANMLZoo standardized and optimized benchmarks to the Micron D480 fabric, they inherited the capacity and routing features of this architecture.
- **Cut down or inflated benchmarks:** In order to standardize benchmarks to a Micron D480 chip, in some cases only parts of larger automata applications were used to generate some benchmarks, making cross-algorithm application performance comparisons difficult.
- **Benchmarks are single design points:** Some applications have infinitely large design spaces, making benchmark selection challenging. ANMLZoo chose a single design point for each application domain (amenable to implementation on the Micron D480), and does not explore design trade-offs in standard benchmarks.

This paper first discusses these drawbacks in detail, with examples from existing ANMLZoo benchmarks. We then propose a new benchmarking methodology that builds off of known strengths of the ANMLZoo benchmark suite, and attempts to reduce the impact of the above drawbacks, or eliminate them all together. We then present AutomataZoo, a new automata processing benchmark suite that uses this new benchmarking methodology. AutomataZoo is composed of 24 new benchmarks, covering existing and novel use-cases for automata processing. Each benchmark represents—to the best of our abilities—real-world use-cases of automata processing. The benchmark suite includes instructions outlining how each automata and input stimulus was generated or sourced using open-source code bases. Where benchmark design is semi-arbitrary, or architecture-specific optimizations were employed, AutomataZoo offers multiple design points in order to create a fairer and more versatile benchmark suite.

We then demonstrate how this new methodology and new features of the benchmark suite improve benchmarking of automata processing engines and architectures:

- We demonstrate a new approach to benchmark generation that creates benchmarks that better represent the behavior and semantics of application domains.
- We show how multiple variants of benchmarks from the same application domain can help expose trade-offs in automata application and accelerator design.
- We show how benchmarks with architecture-specific optimizations can result in benchmark bias. Thus our benchmark suite includes optimized and unoptimized variants, or allows users to generate either.
- We show how benchmarks that are full kernels enable fairer cross architectural comparisons by comparing automata-based benchmarks with non-automata-based algorithms that compute the same kernel.
- We introduce two new benchmarks that benefit from bit-level descriptions (as opposed to traditional byte-level descriptions such as regular expressions). Sub-byte-level pattern descriptions are common in file meta-data headers and malware signature descriptions. We describe how bit-level descriptions can be easily generated using open-source automata software development kits, and converted to byte-level automata using automatic transformations.
- We propose a new benchmark generation methodology for string scoring automata that uses profiling to build benchmarks that can closely approximate the work of an infinitely large class of problem sizes.

II. Areas for Improvement in ANMLZoo

Since its release, ANMLZoo has been adopted as the defacto standard automata processing benchmark suite. ANMLZoo is especially useful because it introduces a wide variety of automata processing use-cases that have diverse static and dynamic properties. These benchmarks are easily accessible, available in a standard automata format, and can be analyzed, optimized, and evaluated using the VASim automata simulation environment [19]. ANMLZoo's spatial standardization feature allows for no-hardware comparisons to Micron's Automata Processor, enabling researchers to easily compare or peg their accelerator's performance to this accelerator architecture [25].

However, as ANMLZoo becomes more popular, and is used in more evaluations of ever more capable hardware accelerators, some areas for improvement have been identified that reduce its usefulness. This section outlines these areas for improvement, and provides examples using benchmarks from the ANMLZoo benchmark suite.

A. Promotes an Obsolete Baseline

ANMLZoo benchmarks are designed to fit the Micron D480 AP. However, the Micron D480 has not been made commercially available, and since its announcement in 2013, FPGAs have benefited from increased capacities and operating frequencies and now show improved performance [21]. Thus, by building benchmarks that are pegged to particular hardware, ANMLZoo promotes comparison to a now obsolete baseline. Future evaluations should adopt new baselines based on real-hardware evaluations that reflect the current state-of-the-art.

B. Benchmarks Inherited Features of the Micron D480

Each ANMLZoo benchmark was designed to maximize the resources of one Micron D480 AP chip. This means that each ANMLZoo benchmark is constrained by either the capacity or routing resources of the AP. In some cases, complex automaton topology greatly limited the number of automata that could fit on each chip for a particular benchmark. For example, the Levenshtein benchmark maximizes the routing resources of the AP, but only uses 6% of the architecture's state capacity. Wadden et al. showed that the Micron D480's tree-based routing architecture caused this inefficiency, and that a more traditional, 2D or island style routing fabric allowed for much higher utilization of the fabric. Thus, the benchmarks are permanently defined by both the unique capabilities and weaknesses of a particular piece of hardware.

C. Benchmarks Optimized For AP Features

In some cases, automata application developers optimized and transformed automata to take advantage of specific features of the AP. For example, Sequential Pattern Mining [13] has automata filter structures that can be configured to compute any pattern that can fit into the structure by reconfiguring the character sets recognized by the automata states. This allows the AP to re-use the automata structures for multiple problem sizes, reducing the cost of reconfiguration, and greatly increasing total system performance. While these designs are sometimes desirable when full executing on the AP, they might induce poor performance on other architectures, that is, they might not be *performance portable*. Benchmarks with such designs are therefore not ideal, as they might unfairly impact the performance of non-AP-like automata accelerators or software engines.

D. Cut-Down or Artificially Inflated Benchmarks

Another downside to spatial standardization is that in some cases, automata needed to be pruned or artificially inflated in order to represent the exact capabilities of a Micron AP chip.

A downside to artificial pruning is that it changes kernel results. Results from automata processing on cut-down or pruned benchmarks might change the final results of the full kernel. In these cases, benchmark performance cannot be fairly compared to other algorithms on other architectures that might perform better than automata processing when computing the same kernel. For example, the ANMLZoo Random Forest benchmark contained a pruned subset of the decision trees learned for a particular model. This means that the results from the decision trees in the benchmark are a subset of those necessary to properly classify the inputs. Thus, the Random Forest benchmark cannot be fairly compared to an algorithm that computes the full decision tree.

A downside to artificial inflation is that we might change a canonical application. Sometimes automata processing tasks have fixed, "canonical" problem sizes. For example, consider Protomata [30], which attempts to identify the locations of 1309 protein motifs from the Prosite database in an input protein sequence. New Prosite motif patterns are found rarely, and thus the benchmark can be considered a static, fixed workload to accelerate. These 1,309 patterns did not saturate the resources of an AP chip. Thus, ANMLZoo added an additional 1,000 synthetic "Prosite-like" motif pattern filters in order to generate a Protomata benchmark that maximized the resources of the AP chip [19]. However, this benchmark is not representative of the actual needs of the fixed application, and demotivates architects from identifying methodologies for using excess spatial resources to increase performance or reduce power consumption.

E. Semi-Arbitrary Benchmark Design

Benchmarks should be designed to capture behavior that is representative of real-world use. The more representative the benchmark, the more meaningful the results of experiments. In the case of fixed, canonical problem sizes mentioned above, benchmarks can simply be set to these sizes. However, some automata processing application domains do not have canonical problems, and can have an infinite number of possible choices for benchmarks. ANMLZoo constrained the range of benchmark choices by picking automata problems that represented the capabilities of one AP chip. For example, in the case of Hamming [7] and Levenshtein [5] filters, where pattern strings and similarity score determine the size and number of the automata, ANMLZoo designed the length and scoring distance of these filters be routable on a single AP chip. If we remove this constraint in favor of larger or more realistic problem sizes, benchmark can become more arbitrary.

Inputs for benchmark automata were also chosen semi-arbitrarily. For example, for ClamAV virus detection, ANMLZoo used a version of the VASim automata simulator binary as input to represent a file that needs to be scanned for a virus. As a result, the ANMLZoo ClamAV benchmark is only stimulated by one file type, and detects no viruses.

III. Benchmark Generation Methodology

In order to provide improved automata processing benchmarks, we outline a new benchmark generation methodology that produces benchmarks that will better serve automata processing researchers in the future.

100% open-source software: All new benchmarks must be generated using freely available, open-source software. Some automata applications are generated using custom automata-graph producing software, while other applications are regular expression derived. We use the tools in the MNCaRT open-source automata processing toolchain [31] to generate automata. MNCaRT includes the VASim automata SDK [32] and pcre2mnrl, a regular expression to automata compiler that leverages Intel Hyperscan [33], an industry standard regular expression engine. This improves the verifiability of benchmarks and allows users to more easily explore new automata designs, optimizations, and transformations that are perhaps more amenable to their software engine or accelerator. Instructions for how each benchmark was generated, and/or the source of inputs, must be provided. Thus users can generate

new automata graphs and new input stimuli for automata wherever possible.

Free-form benchmarks: Where workloads are fixed or "canonical", we generate benchmarks without regard for the particular capabilities of a target architecture. Thus the corresponding benchmarks are a complete set of rules and patterns, and are only modified if some rules or patterns are not supported by open-source tools or if the rules are applied selectively within an application. In the case of arbitrary choice (e.g. Hamming and Levenshtein) we build automata benchmarks that represent realistic problem instances (large or small) but attempt to keep the automata manageable in system memory of a reasonable research desktop system. If benchmarks are too large to fit into the resources of a target spatial architecture, researchers must develop ways to evaluate sequential runs of the partitioned benchmark.

Reduce arbitrary choice wherever possible: Arbitrary choice is practically impossible to avoid in some benchmarks, but certain approaches to benchmark generation can be used to reduce the amount of arbitrary choice when workloads are not fixed. We suggest two methodologies to reduce arbitrary parameter choice: multiple benchmark variants per application domain and profile-driven benchmark pruning.

Multiple benchmark variants per application domain with different parameter choices can give a sense for relationships between those parameters and benchmark size or system performance. Section VI shows how three different benchmarks, with varying parameters, expose a complex trade-off space. Section VII examines performance portability of optimizations for spatial architectures, and their impact on the performance of CPU-based automata engines.

Some automata filters (e.g. Hamming and Levenshtein) can be built for any pattern length. We propose a novel methodology to reduce the parameter space for benchmarks: *profile-driven pruning*. We rely on the intuition that many states towards the ends of long automata filters do little to no work. We build benchmark automata filters so that they capture a vast majority of the work done for longer filters; thus a single benchmark sub-filter can be a proxy for the difficulty for an entire class of longer filters. Profile-driven pruning, and how it was used to generate AutomataZoo's Hamming and Levenshtein benchmarks is discussed in detail in Section X.

IV. Automata Zoo

We use the methodology outlined in the previous section to generate a new benchmark suite that we call AutomataZoo. AutomataZoo is composed of 24 benchmarks from 13 different application domains that have been shown to benefit from accelerated automata processing. Prior benchmarks that were synthetic (Synthetic), and/or purely arbitrary (PowerEN), or that did not have open-source software generators (Fermi) were omitted from AutomataZoo. The benchmarks and summary statistics are shown in Table I. Each application domain can have multiple benchmark variants. Each variant attempts to vary design-choice parameters to allow the benchmark suite to capture domain-level design trade-offs. *Active set* is the average number of states that compute (attempt a match) per input symbol. Active set is often used as a proxy for performance on sequential, memory-based architectures such as CPUs. For spatial-architectures, such as Micron's D480 or FPGAs, performance is independent of active set, and usually much higher, however, these architectures can be capacity and routing constrained. Active set computations were computed using the standard inputs for each AutomataZoo benchmark using VASim. Below, we discuss each application domain and how the associated benchmarks were generated. In the cases where ANMLZoo contained a benchmark from the same application domain, we describe how the new AutomataZoo benchmark(s) improves upon ANMLZoo's design.

Snort is a widely used network intrusion detection system that attempts to find patterns in network pattern capture files. Each Snort rule can include a regular expression pattern, and these regular expression patterns are used to construct the Snort benchmark.

While ANMLZoo only used a restricted number of regular expressions to ensure a particular benchmark size, AutomataZoo considers every regular expression from the Snort ruleset that can be successfully compiled by the pcre2mnrl tool in the MNCaRT automata processing toolchain [31] (e.g. pcre2mnrl does not support back references). ANMLZoo also did not account for semantics of Snort-specific regular expression flags or Snort rules leading to an extremely high false positive rate. How AutomataZoo handles these flags, and how the AutomataZoo Snort benchmark is more representative of the Snort application is discussed in Section V.

ClamAV is a virus-detection tool that relies on a publicly available database of patterns to scan for viruses in byte streams. These patterns are converted to regular expressions using a tool supplied with the benchmark and then compiled to automata. ANMLZoo used a subset of these patterns in order to keep the benchmark size within the resources of a Micron AP chip. AutomataZoo does not restrict the number or contents of ClamAV patterns, but only considers patterns that are able to be compiled by the open-source Hyperscan regular expression compiler in the pcre2mnrl tool [31]. The original ANMLZoo benchmark used a single executable as the basis for its input stimulus. We build a more representative input by constructing a disk image including various files and two embedded virus fragments from VirusSign [34] that trigger ClamAV rules.

Protomata is an automata-based application that searches for a set of 1309 protein motif patterns from the PROSITE database [35]. These patterns can be converted to automata [8] to scan for protein motifs in UniProt protein sequence databases [36].

The 1309 protein patterns did not fill the resources of an Micron D480 AP chip, and thus, ANMLZoo used a synthetic protein rule generator to fill the AP chip. However, new protein motifs are rarely found, and the real application does not require more patterns. Thus, AutomataZoo uses the original, non-synthetic protein patterns. This smaller, more realistic benchmark might motivate accelerator designs that are able

Benchmark	Domain	Input	States	Edges	Edges/ Node	Subgraph Count	Avg. Size	Std. Dev	Compressed States	Compr. factor	Size vs ANMLZoo	Active Set
Snort	Network Intrusion Detection	PCAP file	202,043	235,488	1.17	2,486	81.27	1.13	162,514	0.20x	4.71x	409.358
ClamAV	Virus Detection	Disk image	2,374,717	2,377,737	1.00	33,171	71.59	0.47	2,254,598	0.05x	53x	356.532
Protomata	Motif Search	Uniprot Database	24,103	24,031	1.00	1,309	18.41	1.13	22,289	0.08x	0.58x	712.884
Brill	Part of Speech Tagging	Brown Corpus	115,549	157,917	1.37	5,946	19.43	0.02	72,637	0.37x	2.76x	78.2558
Random Forest A	Machine Learning	Custom	248,000	248,000	1.00	8,000	31	0	240,001	0.03x	7.60x	862.504
Random Forest B	Machine Learning	Custom	248,000	248,000	1.00	8,000	31	0	240,001	0.03x	7.60x	1,043.18
Random Forest C	Machine Learning	Custom	992,000	992,000	1.00	16,000	62	0	976,001	0.02x	30.93x	2,334.97
Hamming 18x3	String Similarity	Random DNA	108,000	183,000	1.69	1,000	108	0	87,881	0.19x	7.81x	1,944.38
Hamming 22x5	String Similarity	Random DNA	192,000	347,000	1.81	1,000	192	0	168,936	0.12x	15.01x	6,324.49
Hamming 31x10	String Similarity	Random DNA	451,000	857,000	1.90	1,000	451	0	427,782	0.05x	38.01x	19,617.8
Levenshtein 19x3	String Similarity	Random DNA	109,000	445,000	4.08	1,000	109	0	91,007	0.17x	34.21x	4,528.69
Levenshtein 24x5	String Similarity	Random DNA	204,000	1,251,000	6.13	1,000	204	0	183,459	0.10x	68.97x	18,033.9
Levenshtein 37x10	String Similarity	Random DNA	557,000	6,221,000	11.17	1,000	557	0	530,981	0.05x	199.62x	85,866.1
Seq. Match 6w 6p	Ordered Pattern Counting	Custom	51,570	110,016	2.13	1,719	30	0	51,570	0x	0.51x	5,538.98
Seq. Match 6w 6p wC†	Ordered Pattern Counting	Custom	53,289	113,454	2.13	1,719	31	0	53,289	0x	0.53x	5,555.98
Seq. Match 6w 10p	Ordered Pattern Counting	Custom	85,950	185,652	2.16	1,719	50	0	85,950	0x	0.86x	5,465.23
Seq. Match 6w 10p wC†	Ordered Pattern Counting	Custom	87,669	189,090	2.16	1,719	51	0	87,669	0x	0.87x	5,497.23
Entity Resolution	Duplicate entry identification	100k names	413,352	641,136	1.55	10,000	41.34	0.11	309,475	0.25x	54.40x	57.5615
CRISPR CasOffinder	DNA pattern search	DNA	74,000	94,000	1.27	2,000	37	0	47,901	0.35x	NA	191.64
CRISPR CasOT	DNA pattern search	DNA	202,000	336,000	1.66	2,000	101	0	162,930	0.19x	NA	953.753
YARA	Malware pattern search	Malware files	1,047,528	1,025,863	0.98	23,530	44.52	0.22	482,233	1x	NA	579.739
YARA Wide	Malware pattern search	Malware files	115,246	112,910	0.98	2,620	43.99	0.20	82,837	0.28x	NA	123.964
File Carving	File metadata search	Multi-media files	2,663	156,601	58.81	9	295.89	93.30	699	0.74x	NA	15.6547
AP PRNG 4-sided	Pseudo-random number generation	Pseudo-random bytes	20,000	32,000	1.60	1,000	20	0	NA	NA	NA	4,500
AP PRNG 8-sided	Pseudo-random number generation	Pseudo-random bytes	72,000	128,000	1.78	1,000	72	0	NA	NA	NA	2,500

TABLE I

AUTOMATAZOO BENCHMARKS.*Subgraphs* ARE DISTINCT, CONNECTED COMPONENTS WITHIN THE AUTOMATA BENCHMARK. *Compressed states* IS THE NUMBER OF STATES IN THE BENCHMARK AFTER VASIM'S STANDARD, PREFIX-MERGING-BASED OPTIMIZATIONS ARE APPLIED. † WC INDICATES THE BENCHMARK INCLUDES NON-STANDARD COUNTER ELEMENTS.

to use excess automata state capacity to improve performance for smaller rulesets.

Brill tagging is a rule-based approach for correcting incorrect part-of-speech tags in text. Ambiguous or incorrect tags are identified using regular-expression patterns and then updated offline with the correct tags [9]–[11]. ANMLZoo used a program to learn and generate 2050 tag update rules, but this software was never publicly released. AutomataZoo uses a new, open-source Brill rule generator by Sadredini et al [11], [37]. We use 5000 rules generated by this tool as the AutomataZoo benchmark. Adding rules in the benchmark enables better evaluation of trade-offs between rule-based and machine-learning-based approaches to part-of-speech tagging [11].

Random Forest is a machine learning model based on ensembles of decision trees; Tracy et al. developed an automata-based approach [18]. ANMLZoo included a single Random Forest benchmark (trained on the MNIST digit recognition database) that was generated using open-source code. AutomataZoo uses the Random Forest code to generate three benchmarks that highlight state number/performance/model quality trade-offs while varying different hyperparameters discussed in the Random Forest work [18]. These benchmarks and experiments highlighting their usefulness for exposing design trade-offs in spatial architectures are presented in Section VI.

ANMLZoo's Random Forest benchmark also pruned some decision trees from the trained model in order to fit the benchmark into the resources of the Micron AP. Each AutomataZoo Random Forest benchmark is a full model that has interpretable results, allowing researchers to compare end-to-end performance against non-automata-based decision tree accelerators and software engines. We show how this feature enables new comparisons with non automata-based algorithms in Section VIII.

Hamming and Levenshtein Hamming distance [7] and Levenshtein/Edit Distance [5] are two string scoring kernels. ANMLZoo included both kernels as benchmarks, but each benchmark was designed to account for the limited routing resources of the Micron D480 AP. AutomataZoo uses a profile-based approach to generate a set of new Hamming and Levenshtein benchmarks that better represent real-world use-cases, and that are not limited by the architecture of the Micron D480. We also include three design points as separate benchmarks in order to evaluate the impacts of different design parameters on automata size, topology, and performance. The development of these benchmarks is discussed in detail in Section X.

Sequence Matching uses automata to count sorted sequences of item sets in order to identify frequent sets [13]–[15]. ANMLZoo included this kernel as Sequential Pattern Mining (SPM) [19]. In order to fit into a single Micron D480 chip, the ANMLZoo SPM benchmark pruned itemsets from the full automata. Furthermore, the ANMLZoo benchmark did not include counter elements, a special automata feature of the Micron D480 that is integra for counting support during

pattern mining. AutomataZoo rebuilds this benchmark using open-source software, includes a full, interpretable sequence matching kernel, and also includes a variant that uses counters, greatly reducing the output reporting requirements of the kernel. Furthermore, we include a version of the benchmark that utilizes the symbol replacement feature leveraged in the literature. Symbol replacement allows automata structures to be quickly reconfigured for new automata problem sets, but often requires extra, unused states. AutomataZoo allows developers to evaluate the impact of these extra states. An example evaluation showing the impact of extra states on CPU-based automata engines is presented in Section VII.

Entity Resolution attempts to find duplicate entries in a streaming database. This is difficult because duplicates entries, such as names, might have variations in their representation or typos. Entity resolution was shown to benefit from accelerated automata processing by Bo et al. [12]. ANMLZoo contained an Entity Resolution benchmark of 1,000 name patterns generated from a lexicographically similar 500 name database. Thus, these names were mostly similar, making the automata highly compressable, and not representative of the behavior of common-case entity resolution. AutomataZoo builds an entirely new Entity Resolution toolchain with a name generator that can introduce arbitrary names of different formats, and also introduce various errors in order to generate a realistic input for the automata. The new Entity Resolution benchmark is composed of patterns to resolve matches for over 10,000 unique names.

CRISPR/Cas9 is a novel technology that enables arbitrary editing of genomes at targeted locations. Identifying these targeted locations is challenging, and has been shown to benefit from accelerated automata processing by Bo et al. [6]. AutomataZoo builds a CRISPR benchmark so that users can identify scaling trends and compare to other techniques for this important emerging technology. Bo compared to two algorithms: CasOFFinder (GPU) and CasOT (CPU), and developed two different automata filters to mimic the behavior of these programs. We therefore develop two CRISPR benchmarks (OFF and OT) to allow users to compare pros and cons of each method, as well as properly compare existing and future state-of-the-art techniques on CPUs, GPUs, and FPGAs. The benchmarks were developed using open-source software [38]. For each benchmark, we generate 2,000 filters, reflecting a problem size that is larger than most existing explorations, and the largest evaluated in Bo's work.

YARA is a malware pattern description language that lets users define rules to match certain 4-bit (nibble) level patterns in files using hexadecimal strings [4]. These strings can have wildcards, and bounded or unbounded jumps (sequences of wild cards). The YARA pattern language also allows for byte-level text strings, and regular expressions. YARA rules were not previously evaluated using automata-processing engines, because of their nibble-level features. Most automata processing languages (regular expressions) and engines (REAPR [21] Hyperscan [33], RE2 [39], AP [22]) do not consider sub-byte patterns. We develop a specialized tool to convert nibble-level patterns to byte-level automata in order to generate a YARA benchmark. We also consider *widened* YARA rules (rules encoded to read two bytes per input symbol) and develop a special transformation to generate byte-level, widened automata as a separate benchmark variant. Benchmark generation using nibble-level pattern conversion and automata widening is discussed in more detail in Section IX-A.

File Carving is the task of identifying files in a stream of input bytes. File carving is useful if a file system has been corrupted or deleted and files or metadata need to be recovered. Existing approaches use small, exact match header and footer patterns to identify start and stop locations of files in a byte stream [40]. However, small exact matches occur frequently, and produce a large number of false positives. To reduce false positives, we build a File Carving automata that recognizes complex file header and footer patterns for zip, mpeg-2, and mpeg-4 file types. We also include patterns to identify desirable forensic metadata such as e-mails and social security numbers.

Some file metadata patterns can span byte boundaries, and such patterns are not easy to represent using a regular expression or byte-based automata. To overcome this challenge, we first describe sub-byte-level patterns using bit-level automata and then automatically transform them into a byte-level automata using automata striding [41]. This technique is discussed more in Section IX.

AP PRNG [16] is a methodology for modeling Markov Chains using finite automata. Automata transitions are usually driven by a structured input, such as English text, or a DNA sequence. But if random symbols are used to simulate automata, transitions become probabilistic. Wadden et al. [16] showed that many parallel automata could be used to generate high-throughput, high-quality pseudo-random numbers. We include AP PRNG as a benchmark in AutomataZoo and include open-source software to generate a variety of AP PRNG automata. Our AP PRNG benchmark has 1,000 8-state Markov Chains and is driven by uniformly distributed pseudo-random input.

V. Improving Representative Behavior of Benchmark Automata

Snort is a set of rules used to identify malicious or suspicious network traffic. Each rule can be composed of multiple patterns and patterns can be described using regular expressions. ANMLZoo included a Snort benchmark composed of a subset of these regular expression patterns. The benchmark did not account for Snort-specific modifiers or the semantics of the snort rule the regular expression was associated with. The resulting benchmark included some regular expressions that matched extremely frequently when applied to the entire input stream. For example, the standard input PCAP file included with ANMLZoo causes the Snort benchmark in ANMLZoo to match patterns on 99.5% of all input bytes. Such high reporting rates are known to cause output reporting bottlenecks in Micron's D480 [42].

We examined these rules and found that the regular expressions that matched extremely frequently usually had Snort spe-

cific regular expression modifiers. Snort modifiers are special regular expression flags that guide the Snort system to apply the pattern to a specific part of an input packet (e.g. a URI buffer). Because these regular expressions were designed with this selective application in mind, we choose not to include them in the benchmark. After removing 2,856 rules with Snort specific modifiers, we found that reporting rates decreased by about 5x.

These reporting rates were still extremely high relative to what should be expected from the actual Snort system. In the most extreme case, one rule was responsible for over half of all reports, an extreme outlier. This was because our benchmark regular expressions did not account for information from within the Snort rule they were derived from, and only considered the information encoded in regular expression-level modifiers. We examined rules that reported extremely frequently and found that other modifier keywords, (similar to Snort specific regular expression modifiers) also guided regular expressions to be applied to selective regions of a packet. For example, the outlier rule discussed above used the *isdataat* Snort modifier [1] to check if data exists downstream relative to the match. These regular expressions should be matched in context with other information, and we choose to exclude them from the AutomataZoo benchmark. Once we excluded 182 regular expressions from Snort rules with *isdataat* modifiers, reporting rates drop by a further 2x.

By excluding rules that should not be matched on the entire PCAP file, we build an automata processing benchmark that is more representative of the actual Snort application behavior. We also apply this technique to YARA rules, which also contain context sensitive patterns, and remove rules that report frequently out of context.

VI. Building Benchmark Variants to Explore Design Trade-Offs

Our new benchmark generation methodology included a goal to reduce arbitrary design where possible. One way to reduce arbitrary benchmark design is to build multiple benchmark variants for the same application. In this way, a family of benchmarks within one domain can expose important trade-offs in application-specific automata design. These variants can then provide intuition into how changing parameters will impact performance in new systems.

A. Example: Random Forest Decision Tree Ensembles

An example of an application domain where there is a lot of arbitrary choice in benchmark design is automata-based Random Forest decision tree ensembles [18]. When training a Random Forest model, it is necessary to select the maximum number of leaves in a tree, and the number of features used to classify an input. Increasing the number of leaves in a tree generally increases the mean model accuracy. However, trees with more leaves are larger, and increase the number of automata states required to implement the model, increasing automata activity and state capacity requirements.

Increasing the number of features in a model also generally increases the mean model accuracy. However, more input features require deeper trees and deeper trees require more symbol inputs (cycles) to perform a classification. Tracy et al. [18] demonstrated a linear relationship between the number of features used for inference and the runtime of the automata-based model on the automata processor.

Interestingly, non-automata-based algorithms do not suffer from this performance trend. While the number of features used by the model is an extremely important hyperparameter to consider for performance on spatial architectures, it is not for von Neumann architectures, something previously unexposed in the ANMLZoo benchmark suite.

B. Benchmark Design Methodology

In order to expose various design trade-offs discussed above, AutomataZoo chooses *multiple* sets of hyperparameters as inputs to the training algorithm to build benchmark variants. All Random Forest models are trained on the MNIST handwritten digit dataset.

We selected three different sets of parameters, with each set differing in one parameter dimension. We trained each model with 20 trees and varied the number of features and the maximum number of leavers per tree. The hyperparameters chosen for each benchmark, and the resulting accuracy of the model are shown in Table II.

Variant	Features	Max Leaves	States	Accuracy	Runtime
A	270	400	248k	93.37	1.35x
B	200	400	248k	92.91	1.0x
C	200	800	992k	93.85	1.0x

TABLE II
Random Forest benchmark variant trade-offs. Increasing the number of features increases accuracy, but increases runtime. Increasing the maximum leaves in the tree increases automata accuracy, but increases automata size.

Variant A and variant B differ by the number of features in the input stream. We chose 270 features for variant A because a feature count greater than 270 had a negligible impact on the accuracy of the model, and only decreased performance. Variant B and C use 200 features because this number assured model accuracy stayed above 90%.

Variant B and C differed by the number of leaf nodes per tree, which increases the size of each tree. When considering 200 features, increasing the maximum leaf count from 400 (variant B) to 800 (variant C) increases model accuracy by almost a full percent. However, the size of the model increases by 4 times.

These three models expose varying design choices that have huge impacts on performance on varying architectures. Architectures that are good at processing smaller automata faster might benefit from Random Forest models with higher feature count, but lower leaf count (variant A). Architectures that have large capacities might be able to handle increased leaf count, and could benefit from the performance of lower feature count (variant C). If high accuracy is not required, architectures should attempt to use lower feature count and lower

CPU Engine	6 Wide	6 Wide Padded	Overhead
VASim	165.94	210.231	26.7%
Hyperscan	0.9209	0.94783	2.92%

TABLE III
IMPACT OF MICRON AP-SPECIFIC OPTIMIZATIONS ON CPU-BASED AUTOMATA PROCESSING PERFORMANCE.

leaf count in order to increase performance and reduce state capacity requirements (variant B). None of these intuitions could be derived without considering multiple benchmarks for this domain, something that ANMLZoo did not provide.

VII. IMPACTS OF ARCHITECTURE-SPECIFIC DESIGNS

Just as many different sequential programs can compute the same algorithm, and many algorithms can compute a final kernel, many different automata can be designed to recognize the same language. Equivalent automata that are smaller are often preferred because smaller automata tend to have less activity, meaning higher performance on von Neumann-based architectures, and also require fewer spatial resources if implemented in a spatial architecture. However, not all automata transformations might be performance portable. Some automata optimizations, especially when designed to take advantage of features of a particular architecture might hurt performance of other automata processing methods.

When designing a benchmark, it is therefore important to allow users to turn on or off these architecture-specific optimizations, or provide both versions as separate benchmark variants for exploration. One critique of ANMLZoo was that it included automata that were optimized for the Micron Automata processor, and might have unfairly disadvantaged other competing architectures. In particular, padding automata with extra states in order to enable soft reconfiguration for larger patterns [12], [13], is a feature that CPUs do not need, and that could unfairly hurt performance of other architectures. Soft reconfiguration or symbol replacement allows spatial automata processors to reconfigure the character sets of each automata state, without reconfiguring the routing fabric. This results in a lower cost reconfiguration time on Micron's Automata processor, and is important for end-to-end automata processing performance of certain applications.

To measure the impact of extra states on CPU performance, we build two variants of the Sequence Matching benchmark. The first benchmark is designed with sequence matching filters that can handle sets of size 10. Thus, these filters can be configured to count sequences of size 1-10. However, each size-10 filter is soft-configured to count ordered sets of size 6, and is thus padded with extra states that do no computation. The second benchmark is designed to count the same set of size-6 sets, but the filters are built to match and are not padded with extra states. We then run both automata using VASim, and Intel's Hyperscan. Results are shown in Table III.

VASim sees a large performance impact from the extra padded states–26.7% overhead–while Hyperscan sees a more modest impact–2.92%. This shows that architecture-specific optimizations can have large performance impacts and are not necessarily performance portable. Even a 2.92% overhead is substantial considering it can be removed by a simple automata transformation. Future evaluations of automata processing across various architectures should account for such transformations that may not be performance portable.

VIII. ENABLING FULL-KERNEL ALGORITHM COMPARISONS

Some ANMLZoo benchmarks were generated by building full automata applications, larger than the AP chip's resources, and then trimming to fit within the chip's resources. We refer to these benchmarks as "cut-down."

The Random Forest benchmark in ANMLZoo is an example of a "cut-down" benchmark. The Random Forest decision tree inference kernel is composed of multiple decision trees. To spatially standardize the benchmark, ANMLZoo generated full decision tree models, slightly larger than the Micron D480 chip, and then removed tree paths until the automata could be placed-and-routed using a single AP chip. This methodology created a valid automata processing benchmark that is useful for measuring relative performance of the automata-based Random Forest algorithm. However, a cut-down automata benchmark no longer does the same computation of the trained model. As discussed in Section II, this makes it impossible to compare the performance of the AP on this application to other, non-automata-based algorithms that compute the same kernel using this benchmark.

Where possible, AutomataZoo is composed of full benchmarks that compute full kernels. In the case of Random Forest automata, this means that each benchmark is composed of a fully trained classifier, and the performance of automata-based classifications can be compared to other classification algorithms. To show the usefulness of this new feature, we compare the performance of the AutomataZoo Random Forest benchmarks on a CPU and FPGA versus a native Random Forest algorithm on a CPU.

A. Methodology

We measure the performance of a trained Random Forest model when executed using automata-based techniques on the CPU and FPGA versus a non-automata-based decision tree algorithm developed for the CPU. For CPU-based automata processing, we use Intel's Hyperscan [33] automata processing engine via the MNCaRT automata processing ecosystem [31]. For FPGA-based automata processing, we use the REAPR automata processing engine [21]. For native decision tree computation on the CPU, we use single and multi-threaded (MT) versions of Scikit Learn [43].

All CPU results were measured on an Intel i7-5870k 6-core server with 12 logical cores. When multi-threading, we use 12 threads. FPGA results were gathered by placing and routing the REAPR automata design targeting a Xilinx Kintex Ultrascale XCKU060 FPGA, and multiplying the resulting maximum virtual clock frequency by the number of input symbols required to drive the automaton.

B. Results

Results are shown in Table IV. All results are normalized to single-threaded hyperscan performance.

Hyperscan	Scikit Learn	SciKit Learn MT	REAPR FPGA
1x	141.5x	401.1x	817.9x

TABLE IV
PERFORMANCE (KILO CLASSIFICATIONS/SECOND) OF EACH RANDOM FOREST AUTOMATA PROCESSING METHODOLOGY. RESULTS ARE NORMALIZED TO SINGLE-THREADED HYPERSCAN PERFORMANCE.

Scikit Learn shows impressive performance advantages over Hyperscan–a 141.5x improvement in single-threaded performance. This provides evidence to support that automata-processing is not the most efficient approach to Random Forest decision tree computation on the CPU. However, the FPGA-based REAPR automata processing engine is still the highest performing architecture/algorithm combination, but shows reduced performance improvements vs. Scikit Learn versus automata processing using Hyperscan.

These results highlight the usefulness of having high-level, full benchmark kernels as it allows easy apples-to-apples comparisons of automata-processing versus other algorithms. This comparison could not have been performed with a constrained benchmark, and shows a more realistic acceleration potential of FPGAs over CPUs for this kernel.

IX. ENABLING SUB-BYTE PATTERN SETS

Most automata processing applications define patterns at a byte granularity. Pattern languages such as regular expressions are explicitly designed to consider bytes one at a time, and do not allow finer-grained bit- or nibble-level patterns to be represented. Thus, finer-grained patterns must be pre-converted to byte-level patterns, and then built as automata or written as regular expressions. Here, we present two new application domains–YARA Rules and File Carving–that utilize sub-byte-level patterns. We develop methodologies using open-source tools to help define sub-byte-level patterns and automatically convert such patterns to common byte-level automata that can be run on most available automata-processing engines.

A. Converting YARA Rules to Automata

YARA rules allow patterns to be represented as exact string matches, hexadecimal 4-bit (nibble-level) expressions, or regular expressions [4]. Exact string matches and regular expressions are easily compiled to automata using existing open source toolchains, however, nibble-level expressions are not supported. Nibble-level expressions include regular expression like descriptions of patterns, including grouping, alternation, and nibble-level wildcards. An example hex pattern with nibble-level wildcards for a YARA rule is shown below. Each space-separated hex value pair represents an 8-bit byte pattern. A '?' represents 4-bits that can take any value.

9C 50 A1 ?? (?A ?? 00 — 66 A9 D?) ?? 58 0F 85

YARA rules can also be *widened*. Widening applies the pattern to 16-bit symbols and assumes that every other byte in the input is zero. Widening is not supported by our open-source toolchain.

In order to enable these features, we build a pipeline that leverages Plyara, an open-source YARA rule parser [44]. We first parse each YARA rule and extract exact string matches, regular expressions, and hexadecimal patterns. We then take each hexadecimal pattern and convert it to a regular expression. When converting hexadecimal patterns to a regular expression, the parser identifies nibble-level pattern wildcards and converts them into a complex byte-level character set within the regular expression. Once the parser emits all YARA patterns as regular expressions, we compile all regular expressions into automata using the pcre2mnrl tool, a part of the MNCaRT automata processing ecosystem [31]. For each automaton that needs to be widened, we apply a widening pass, written as a VASim automata transformation. The widening pass pads the automata with states that only recognize zero. Widened rules constitute their own YARA benchmark variant separate from the original patterns.

B. 8-Striding for Bit-level File Patterns

File Carving is an application that attempts to find file metadata patterns in streams of bytes. Some metadata patterns contain bit-fields within bytes that can take multiple values. Bit-fields can even cross bit boundaries, further complicating the pattern and making regular expressions and automata representation very difficult. For example, consider the date and time stamps in a PKZip file header that use the standard MS-DOS format [45]:

- **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.

Because of the complexity of these patterns, they are usually left as wildcards, allowing possible false positives. To enable these sub-byte, and cross-byte patterns we first allow users to write patterns and generate automata using bit-level automata. Each automata state considers a 0 or 1 symbol, and transitions are made on every bit of input. These bit-level automata are a much more natural medium to define complex bit-fields such as the ones above. Once a bit-level automaton is generated, the automaton is automatically *8-strided*. Striding transforms automata so that they consume multiple symbols per cycle [41]. 8-striding automatically generates an automaton that consumes 8, 1-bit symbols per cycle (i.e. 1 byte), a more common form executable by automata processing engines and architectures.

Both YARA rules and File Carving have a large potential for acceleration using automata processing, and should greatly benefit from these automated automata generation techniques.

X. PROFILE-DRIVEN MESH AUTOMATA PRUNING

ANMLZoo included two automata benchmarks in the *mesh* family. Mesh automata are used to generate string simi-

larity scores such as Hamming [7] and Levenshtein Edit Distance [5]. Mesh automata accomplish this by positionally encoding scores according to whether or not the input string matches or does not match an encoded pattern string.

Both Hamming Distance and Levenshtein/Edit Distance automata have three parameters that determine their size: the length of the encoded string (l), the edit or hamming distance threshold (d), and the number of encoded strings (N).

A. ANMLZoo Mesh Benchmarks

The original ANMLZoo paper examined the routing complexity of these meshes on Micron's D480 AP and showed that larger mesh dimensions tended to greatly increase the required routing resources [19]. Other work has shown that the two-dimensional nature of the topology of mesh automata tended to overwhelm the AP's hierarchical routing matrix, causing underutilization of on-chip state resources when compared to a more-traditional 2D, island-style routing matrix [27]. The two ANMLZoo mesh benchmarks, Hamming and Levenshtein, were designed with this in mind, to fit within the resources of the Micron AP. Because of underutilization and routing constraints, the resulting benchmarks were relatively small compared to real-world problem sizes, and were only able to compute N=93, l=20, d=3 Hamming Distance filters and N=24, l=20, d=3 Levenshtein filters [19]. Real-world bioinformatics pattern search problems can require many thousands of filters ($N > 1000$) that are hundreds of base-pairs in length ($l > 100$) with score thresholds larger than 2 [46] or 3 [19] ($d > 3$).

B. Profile-Driven Selection

We propose three improvements motivated by real-world application requirements and use-cases.

(1) Multiple, larger, scoring thresholds: We generate benchmarks corresponding to multiple scoring thresholds ($d = 3, 5, 10$) that support longer, more realistic scoring patterns.

(2) Profile-Driven Filter Length Selection: Mesh automata behave as filters, and find exponentially fewer matches as the encoded pattern length (l) increases. Intuitively, this makes sense; longer, more specific filters will find patterns less often than shorter filters. At some point, filters become so specific that they barely ever find patterns. We profile automata filters with varying lengths (l) and pick benchmark automata where filters find patterns fewer than approximately once every 1,000,000 inputs. We assume these shorter filters can act as proxies for longer filters that share the same pattern prefix. For example, a pruned filter that finds a match approximately every 1,000,000 symbols does 99.99999% of the work of *any* longer filter, and is thus a reasonable proxy benchmark for any longer filter.

(3) Profile-Driven Filter Number Selection: AutomataZoo benchmarks are then composed of 1,000 of these filters ($N = 1000$) that collectively report approximately once every 1,000 inputs on average. This methodology allows our $\{d, l\}$ benchmark to filter out approximately 99.99% of the candidates found by larger $\{d, l + m\}$ automata, while also maintaining a bounded, and reasonable benchmark size. In this way, these mesh benchmarks represent an entire class of automata, and can be used as proxies for much longer patterns.

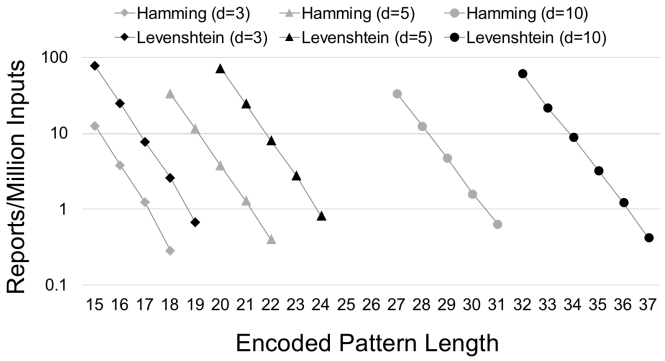

Fig. 1. Average number of pattern matches (reports) per million input symbols for various Hamming and Levenshtein automata configurations. We choose the shortest filters that generate less than 1 report every million inputs as our benchmark automata.

C. Benchmark Design Methodology

We choose scoring distances (d) of 3, 5, and 10. While these choices are semi-arbitrary, we choose three scoring distances to expose the sensitivity of various parameters to this variable.

For each scoring distance d, we build 10 Hamming and Levenshtein filter automata ($N = 10$) of a certain length l using randomly generated DNA pattern strings. We use VASim [32] to simulate all $\{d, l\}$ candidate filter automata on 1,000,000 random DNA base-pair inputs ($\{$a,t,g,c$\}$) inputs. We run 10 trials of this simulation and gather the average number of patterns found per filter, per 1,000,000 inputs. If the average number of patterns found per filter is more that 1/1,000,000, we increment l by 1 and re-run the experiment using candidates $\{d, l+1\}$. If the filters report less frequently than this threshold, we stop, and choose this $\{d, l\}$ as our benchmark filter dimensions.

D. Results

Results of our experiment are shown in Figure 1. As the encoded pattern length (l) is increased, the filters become more specific, and the average number of patterns found in random input decreases exponentially. When the average number of reports per million inputs drops below 1, we stop the evaluation and use this data point as the design for each benchmark filter. For example, for Levenshtein filters that find patterns within edit distance 5 ($d = 5$, black triangles) in an input stream, we need filters with a pattern length 24 ($l = 24$) to reduce the average number of patterns found per filter to below 1.

Each AutomataZoo benchmark is generated by building 1,000 ($N = 1000$) $\{d, l\}$ Hamming and Levenshtein filters using these final parameters. The final benchmark variant parameters are shown in Table V.

XI. Future Work

While AutomataZoo improves upon ANMLZoo, it does not solve all difficulties with automata processing benchmarking. Below are existing difficulties we believe deserve additional research to mitigate.

Kernel	Scoring Distance (d)	Pattern Length (l)
Hamming	3	18
Hamming	5	22
Hamming	10	31
Levenshtein	3	19
Levenshtein	5	24
Levenshtein	10	37

TABLE V
VARIANT PARAMETERS FOR HAMMING AND LEVENSHTEIN AUTOMATA BENCHMARKS. PATTERN LENGTH PARAMETERS WERE DETERMINED USING DYNAMIC PROFILING, AND DESIGNED TO LIMIT RANDOM PATTERN MATCHES TO ONCE EVERY 1,000 INPUTS.

Further improving representative behavior: While AutomataZoo improves representative behavior of applications, it does not account for all context-specific behavior. Automata benchmarks are currently defined to be applied to a streaming input, i.e. all rules are applied equally to all input. In reality, many applications have context dependent rules that allow the existence of one pattern to trigger search for another. One example is context-dependent rules in Snort and YARA rule sets. Some rules are only applied to parts of the input stream, and are very rarely required. Prior work has explored exposing this application context to generate better regular expressions for Snort [47], however, this work is not publicly available. Building context-sensitive automata usually requires extensive domain-specific knowledge and custom parsers, but is attainable with proper effort, and will better guide architects towards more practical architectures.

Adding support for extended finite automata: AutomataZoo introduces two benchmarks that take advantage of counter elements, an advanced feature of the Micron AP. Counters can enable efficient representation of some PCRE range terms [48], but in general are not considered when designing finite-automata-based applications. Future work should explore adding more benchmarks, or benchmark variants, that utilize non-standard automata processing elements. For example, prior work has explored extending the finite automata model to support for PCRE backreferences [48]. Backreferences are nodes that only match an exact sequence previously found by an automaton in the input. Storing the result of prior computation requires some sort of memory, which is not a part of the standard finite automata computation model. Snort contains a large number of these backreferences, and they are an important part of the Snort ecosystem. Future work could address this problem by adding novel, non-standard automata types to the benchmark suite that are more fully featured, and allow for wider applicability of automata processing [48]–[51].

Reducing the impact of architecture-specific optimizations and transformations: It is important that benchmark automata are presented with as few optimizations as possible. The earlier optimizations are applied, the harder it is for low-level tools to identify appropriate transformations and optimizations for underlying software or hardware engines. However, open-source automata tools often apply optimizations to automata at high-levels that are difficult to undo. This downside is supported by the experiments in Section VII. Future work should be conducted to automatically undo high-level optimizations and compartmentalize all optimizations and transformations into lower layers (e.g. VASim [32]), similar to the design of high-performance software compilers.

XII. CONCLUSIONS

This paper presented AutomataZoo, a modern automata processing benchmark suite that improves upon existing automata processing benchmark suites. We first discuss areas for improvement with ANMLZoo benchmark suite, the de-facto standard for automata benchmarking. ANMLZoo standardized its automata benchmarks to fit within the reconfigurable fabric of the Micron D480 automata processor. This meant that many benchmarks were not full kernels, relatively small in size, and sometimes optimized specifically for this architecture. ANMLZoo also did not include multiple variants of benchmarks, where certain important trade-offs in benchmark design existed, or certain architecture-specific optimizations were included.

AutomataZoo attempts to improve upon ANMLZoo by creating benchmarks that are architecture-independent. Our paper proposes a new benchmark design methodology that creates automata problem sizes that better represent realistic application use-cases. As a result, AutomataZoo benchmarks are both larger and smaller than corresponding ANMLZoo benchmarks. AutomataZoo also includes multiple benchmark variants within application domains where important design trade-offs exist. We show that these trade-offs can have a large impact on design decisions when measuring performance on spatial architectures. AutomataZoo also includes variants of benchmarks that include architecture-specific optimizations, allowing researchers to evaluate the performance portability of varying optimizations across different automata processing architectures. Each AutomataZoo benchmark is a full kernel, enabling apples-to-apples comparisons with non-automata-based algorithms. We demonstrate the usefulness of these full benchmarks by performing previously impossible experiments comparing accelerated automata-based algorithms with state-of-the-art, non-automata-based algorithms.

ACKNOWLEDGMENT

This work was supported in part by Semiconductor Research Corporation (SRC), and the Achievement Rewards for College Scientists (ARCS). We are also grateful for the extensive comments and advice provided by Geoff Langdale.

REFERENCES

[1] M. Roesch, "Snort: Lightweight intrusion detection for networks.," in *Proceedings of the USENIX Large Installation Systems Administration Conference (LISA)*, 1999.
[2] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.
[3] ClamAV, "ClamAV Rules." https://www.clamav.net.
[4] VirusTotal, "Yara." https://virustotal.github.io/yara/.
[5] T. Tracy II, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. 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.
[6] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms," in *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2018.

[7] I. Roy and S. Aluru, "Finding Motifs in Biological Sequences Using the Micron Automata Processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 415–424, 2014.

[8] I. Roy, *Algorithmic Techniques for the Micron Automata Processor*. PhD thesis, Georgia Institute of Technology, 2015.

[9] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill Tagging on the Micron Automata Processor," in *Proceedings of the IEEE International Conference on Semantic Computing (ICSC)*, pp. 236–239, 2015.

[10] 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)*, pp. 355–360, Oct 2015.

[11] E. Sadredini, D. Guo, C. Bo, R. Rahimi, K. Skadron, and H. Wang, "A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators," in *24th ACMSIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*, ACM, 2018.

[12] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity Resolution Acceleration using Micron's Automata Processor," *Proceedings of Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[13] K. Wang, E. Sadredini, and K. Skadron, "Sequential Pattern Mining with the Micron Automata Processor," in *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2016.

[14] K. Wang, Y. Qi, J. J. Fox, M. Stan, and K. Skadron, "Association rule mining with the Micron Automata Processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 689–699, 2015.

[15] E. Sadredini, R. Rahimi, K. Wang, and K. Skadron, "Frequent Subtree Mining on the Automata Processor: Challenges and Opportunities," in *Proceedings of the International Conference on Supercomputing (ICS)*, (New York, NY, USA), ACM, 2017.

[16] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron, "Generating efficient and high-quality pseudo-random behavior on Automata Processors," in *Proceedings of the 2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 622–629, Oct 2016.

[17] M. Putic and M. Stan, "Dendroplex: Synthesis, Simulation, and Validation of Hierarchical Temporal Memory on the Automata Processor," in *Proceedings of the Design Automation Conference (DAC)*, 2017.

[18] T. Tracy II, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the automata processor," in *Proceedings of the International Conference on High Performance Computing*, Springer, 2016.

[19] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2017.

[20] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *Proceedings of the ACM International Symposium on Microarchitecture (MICRO)*, pp. 533–545, 2015.

[21] T. Xie, V. Dang, C. Bo, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable Engine for Automata Processing," in *Proceedings of the International Conference on Field Programmable Logic (FPL) to appear*, IEEE, 2017.

[22] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[23] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 79–89, IEEE, 2008.

[24] K. Atasu, F. Doerfler, J. van Lunteren, and C. 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)*, pp. 1254–1265, 2013.

[25] A. Subramaniyan and R. Das, "Parallel automata processor," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 600–612, ACM, 2017.

[26] 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)*, pp. 373–373, Sept 2017.

[27] J. Wadden, S. Khan, and K. 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.

[28] 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.

[29] "Micron Automata Processor SDK." http://micronautomata.com.

[30] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, "High performance pattern matching using the automata processor," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1123–1132, May 2016.

[31] 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*, vol. 17, pp. 84–87, Jan 2018.

[32] J. Wadden and K. Skadron, "VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research," Tech. Rep. CS2016-03, University of Virginia, 2016.

[33] Intel, "Hyperscan regular expression processing engine." https://github.com/01org/hyperscan.

[34] "Virussign." http://virussign.com/.

[35] P. Database. http://www.uniprot.org/proteomes/UP000005640.

[36] U. Databases, "Human proteome up000005640." http://www.uniprot.org/proteomes/UP000005640.

[37] S. Elaheh, "Brillplusplus." https://github.com/elaheh-sadredini/BrillPlusPlus.

[38] C. Bo, "Crispr." https://github.com/chunkunbo/CRISPR.

[39] Google, "Re2 regular expression processing engine." https://github.com/google/re2.

[40] G. G. Richard III and V. Roussev, "Scalpel: A frugal, high performance file carver.," in *DFRWS*, 2005.

[41] M. Becchi, *Data structures, algorithms and architectures for efficient regular expression evaluation*. PhD thesis, Washington University in St. Louis, 2009.

[42] J. Wadden, K. Angstadt, and K. 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.

[43] "Scikit-learn: Machine learning in python." http://scikit-learn.org/stable/.

[44] C. Buia, "Plyara: Parse yara rules and operate over them more easily.." https://github.com/8u1a/plyara.

[45] PKWare, ".zip file format specification." https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.

[46] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying automata processing: Gpus, fpgas or micron's ap?," in *Proceedings of the International Conference on Supercomputing*, ICS '17, (New York, NY, USA), pp. 1:1–1:11, ACM, 2017.

[47] 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)*, pp. 1–5, June 2011.

[48] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, (New York, NY, USA), pp. 25:1–25:12, ACM, 2008.

[49] 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*, vol. 32, no. 10, pp. 1822–1833, 2014.

[50] K. Angstadt, A. Subramaniyan, E. Sadredini, R. Rahimi, K. Skadron, W. Weimer, and R. Das, "ASPEN: A scalable in-SRAM architecture for pushdown automata," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 51, Oct 2018.

[51] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: A genome sequencing accelerator," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 69–82, June 2018.