2018 IEEE International Symposium on High Performance Computer Architecture

Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures

Jack Wadden, Kevin Angstadt†, Kevin Skadron

Department of Computer Science
University of Virginia, Charlottesville, VA, 22904
{wadden, skadron}@virginia.edu

†Computer Science and Engineering
University of Michigan, Ann Arbor, MI, 48109
angstadt@umich.edu

ABSTRACT

Automata processing has seen a resurgence in importance due to its usefulness for pattern matching and pattern mining of "big data." While large-scale automata processing is known to bottleneck von Neumann processors due to unpredictable memory accesses, spatial architectures excel at automata processing. Spatial architectures can implement automata graphs by wiring together automata states in reconfigurable arrays, allowing parallel automata state computation, and point-to-point state transitions on-chip. However, spatial automata processing architectures can suffer from output constraints (up to 255x in commercial systems!) due to the physical placement of states, output processing architecture design, I/O resources, and the massively parallel nature of the architecture.

To understand this bottleneck, we conduct the first known characterization of output requirements of a realistic set of automata processing benchmarks. We find that most benchmarks report fairly frequently, but that few states report at any one time. This observation motivates new output compression schemes and reporting architectures. We evaluate the benefit of one purely software automata transformation and show that output reporting costs can be greatly reduced (improving performance by up to 40% without hardware modification. We then explore bottlenecks in the reporting architecture of a commercial spatial automata processor and propose a new architecture that improves performance by up to $5.1x$.

1. INTRODUCTION

Moore's law and the breakdown in Dennard scaling have led to architectures with large transistor budgets but with strict power constraints. To maintain performance scaling, architects are increasingly using domain-specific specialization in systems to increase both power efficiency and performance.

One application domain that has seen increased interest over the last decade is finite state machine computation or *automata processing*. Automata processing is one of the Berkeley 13 parallel motifs [1] and is an important kernel in network traffic analysis [2], and virus detection [3]. Due to the ever growing amount of data and increasing velocity of network traffic [4], quickly and efficiently identifying patterns and information in databases and packet streams has become increasingly important.

Furthermore, new research has shown that many important application domains can be represented as finite automata, and would benefit from specialized automata accelerators. Examples include natural language processing [5], network security [6], graph analytics [7], high-energy particle physics [8], pseudo-random number generation [9], bioinformatics [10, 11, 12], data-mining [13, 14, 15], and machine learning [16, 17]. Research continues to discover novel and exciting automata applications motivating the importance of acceleration of automata computation.

Many attempts have been made to accelerate finite automata computation using both CPUs [18, 19, 20] and GPUs [21, 20]. However, the challenging memory access patterns and large bandwidth requirements for automata simulation often cause memory bottlenecks on von Neumann architectures.

Spatial, reconfigurable architectures, such as field programmable gate arrays (FPGAs), usually offer much improved performance over von Neumann solutions [20]. 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 [20]. Thus, spatial architectures are *performance inelastic* and have the same performance no matter how many state transitions are computed in parallel.

However, spatial architectures suffer from two main drawbacks: I/O pin scarcity, and I/O bandwidth constraints. The first drawback, I/O pin scarcity, makes routing the results from many thousands of possible reporting automata states off chip extremely challenging. Simply routing reporting signals through I/O pins does not scale and is not applicable beyond small applica-

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00069

tions. Prior work has used application-specific compression to compress results to fit within a relatively narrow output bus [22]. While successful, this type of automata-specific reporting architecture is not always possible and not a general purpose solution to support a wide array of applications with many thousands of possible unique reports.

The second drawback, I/O bandwidth constraints, cause slowdowns when large amounts of information need to be quickly exported off-chip. 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 [23]; 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 paper focuses on characterizing and mitigating the output reporting problem for spatial automata processing architectures. To the best of our knowledge, this paper is 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, a standardized benchmark suite [20]. We use the VASim virtual automata simulator [24] 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 paper 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.

2. BACKGROUND

2.1 Automata Processing

Informally, finite automata are defined as a directed graph consisting of *states* and transition rules between states. Transitions between states in automata are driven by an input tape of symbols that are globally visible. One iteration of computation that considers an input symbol is called a *symbol cycle*. Each automaton has one or more start states that initiate computation. States that are currently performing computation are said to be *enabled*. During a symbol cycle, each enabled state compares the current symbol on the input tape with a pre-defined set of symbols. If the symbols match, the state *activates*, propagating a transition signal to all of its children. All children of activated states are then enabled to perform computation during the next symbol cycle.

Each automaton also has one or more *reporting* states. If a reporting state activates, the ID of the report state and the current position in the input symbol tape are recorded as output to the user. Reporting occurs when an automaton successfully identifies a specified pattern on the input tape.

2.2 Spatial Automata Processing

Von Neumann architectures struggle to handle the large number of difficult-to-predict memory accesses in large automata. However, spatial architectures excel at large automata processing. Spatial architectures (i.e., architectures with arrays of reconfigurable processing elements such as FPGAs) can place and route automata states in a reconfigurable fabric—akin to gates in a circuit. As long as automata graphs can fit within the resources of the spatial architecture, all states can compute and communicate in parallel and process a single input symbol per cycle. Thus, for highly-active automata, spatial architectures can be several orders of magnitude faster than von Neumann architectures [20].

However, spatial architectures can suffer from output reporting constraints. If many reports are generated on a particular cycle, the architecture is responsible for exporting those reports off chip. If these reports cannot be exported in a single cycle, the architecture must stall, incurring a performance overhead.

Prior work has explored many configurations on general purpose reconfigurable hardware [25, 26, 27, 28, 22]. While these implementations showed impressive acceleration potential over existing von Neumann techniques, none considered a wide variety of automata applications or I/O constraints for large automata.

Micron's Automata Processor [29] is an "automata specific" spatial architecture that gains efficiency over more general-purpose reconfigurable fabrics [30]. Prior work investigating performance of applications implemented on the AP often report nominal performance [5, 31], without regard to how automata reports are exported off chip. However, real-hardware performance depends heavily on how the architecture handles exporting reports generated by the automata [23].

In this paper, we study the impact of reporting on a diverse set of applications and study the impact of I/O constraints in spatial architectures on automata processing. We perform a cycle-accurate analysis of the AP reporting architecture using our proposed framework to draw attention to reporting overheads in existing systems and to characterize reporting requirements for future, more efficient architectures.

3. CHARACTERIZING AUTOMATA REPORTING BEHAVIOR

To understand typical reporting behavior in various automata use-cases, we first profile benchmarks from the ANMLZoo automata benchmark suite [20]. 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.

3.1 Experimental Methodology

We use the Virtual Automata Simulator (VASim) [24] 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 [32].

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).

Table 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

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 1.

3.2 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.

4. 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 sym-

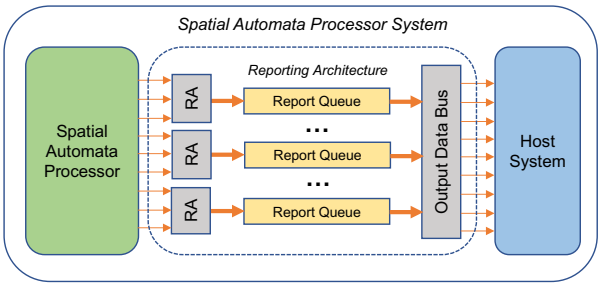

Figure 1: Abstract spatial automata processor system.

bols in the input symbol stream. While prior work often reports this nominal, "kernel" performance [5, 31], 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 [6, 11]. 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 [22]. However, this work only considers a single application (Random Forest [20]) and compresses reporting states using custom, application specific hardware, and is thus not a general purpose solution or analysis [22].

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 [24] 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.

4.1 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 1. Each major structure is described below.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.2 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 1. 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.

5. 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 [29]. The next sections describe the architecture of the Micron D480, the parameters used to

```
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
1  total_cycles ← 0;
2  queue_entries ← 0;
3  foreach c ⇒ E ∈ R do
4  |   total_cycles ← total_cycles + 1;
5  |   set P;
6  |   foreach report event e ∈ E do
7  |   |   add RA(ST(e)) to set P;
8  |   end
9  |   for i ← 1 to |P| do
10 |   |   if i > 1 then
11 |   |   |   total_cycles ← total_cycles + 1;
12 |   |   end
13 |   |   queue_entries ← queue_entries + 1;
14 |   |   if queue_entries = q then
15 |   |   |   total_cycles ← total_cycles + (k * q);
16 |   |   |   queue_entries ← 0;
17 |   |   end
18 |   end
19 end
20 total_cycles ← total_cycles + (k * queue_entries);
21 return total_cycles
```

Algorithm 1: Cycle-Accurate Reporting Overhead Simulation for a Single Report Queue

configure our spatial architecture simulator to match the D480 reporting architecture, simulator validation, and simulated performance results for the ANMLZoo benchmark suite. The lessons learned in the case study will help guide the design of future spatial automata processors, such as those developed on FPGAs.

5.1 The AP D480 Reporting Architecture

Figure 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 [23]. 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 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

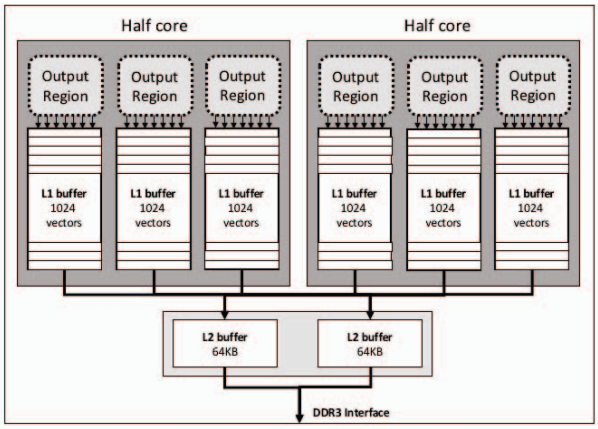

Figure 2: The Micron D480 reporting architecture.

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

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 [23]. Table 3 shows the simulator configuration parameters to match the Micron D480.

5.1.1 *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) [23], 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

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 3: Spatial architecture simulator configuration corresponding to the Micron D480 AP [23].

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.

5.2 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 [23].

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. 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 4.2. 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.

5.3 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.

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.

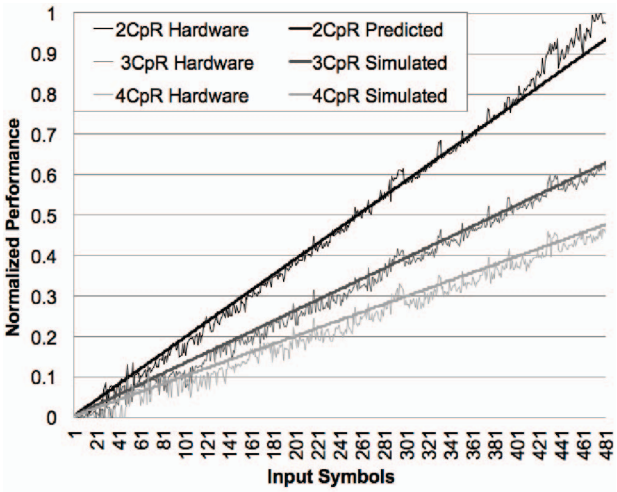

Figure 3: Normalized performance of alpha release AP D480 hardware compared to performance predicted by our trace-based, cycle accurate simulator. Predicted performance matches real performance to within 2.3%-4.6%.

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 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%.

5.4 ANMLZoo Reporting Overheads

We use the simulation methodology described above to simulate performance of ANMLZoo applications on the Micron D480. Figure 4 shows the overhead associated with output reporting for all 12 non-synthetic AN-

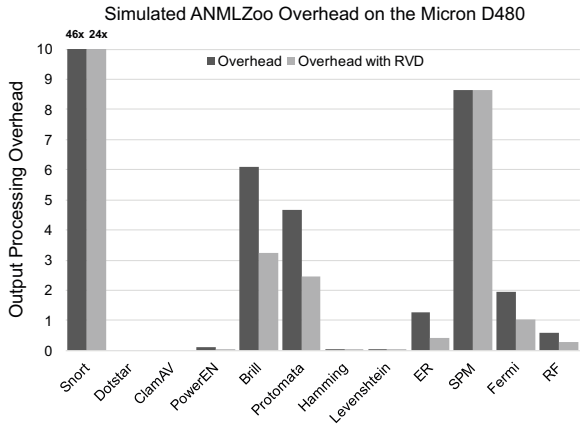

Figure 4: Simulated Micron D480 output processing overhead for each non-synthetic application in the ANMLZoo benchmark suite.

MLZoo benchmarks [20]. 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 paper attempts to understand what causes high reporting overheads and propose solutions to mitigate them.

6. 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 [20]. 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.

6.1 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 5.1.1) 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 ∈ R do
    if |Children(r)| > 0 then
        continue
    end
    port sink;
    ports(sink) ← {r};
    R ← R \ {r};
    char set match ← Matches(r);
    foreach state r' ∈ R do
        if |Children(r')| > 0 then
            continue
        end
        if match ∩ Matches(r') == ∅ then
            ports(sink) ← ports(sink) ∪ {r'};
            R ← R \ {r'};
        end
    end
end
return ports
```

Algorithm 2: Disjoint Report Merging

6.2 DRM Algorithm

Pseudocode for identifying disjoint character sets in reporting states is provided in Algorithm 2. 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 architecture will be merged and not increase reconfigurable fabric resource requirements, and a more realistic measure of potential benefit.

6.3 DRM Potential Study

We apply the DRM algorithm described above to every automata benchmark in the ANMLZoo benchmark suite [20], 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 4 shows the original and compressed number of reporting states.

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 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.

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.

6.4 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. 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.

7.1 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 3 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 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 between 1 and 7. While the common case is sparse reporting, it is also important to make sure we do not hurt perfor-

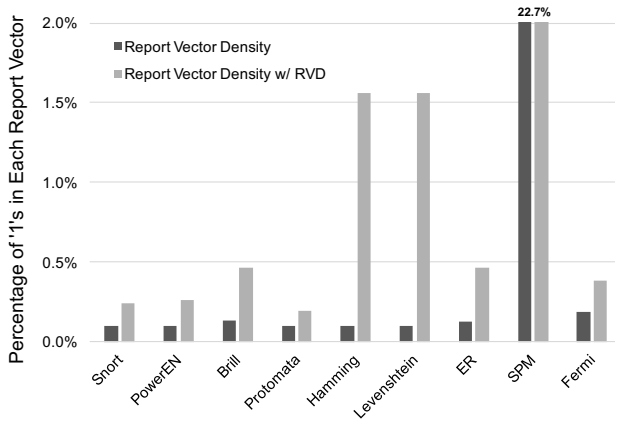

Figure 5: Report vector density (ratio of '1's to total bits) for all applications in ANMLZoo. Most applications have extremely sparse reporting vectors. Report Vector Division (RVD) statically re-sizes report vectors to reduce vector sparsity known at compile time.

mance 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.

7.2 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 5.1.1, 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.

7.2.1 *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

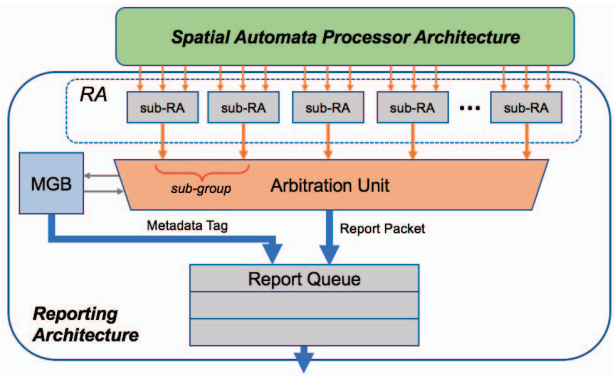

Figure 6: Spatial Reporting Architecture with report aggregation split into sub-modules. The Metadata Generator Block tags report packets with RAD configuration information, the sub-RA ID where the packet was generated in this configuration, and the cycle index the packet was generated. The Arbitration Unit combines and arbitrates packets from sub-RAs to be pushed to the report queue.

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 6.

7.2.2 *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.

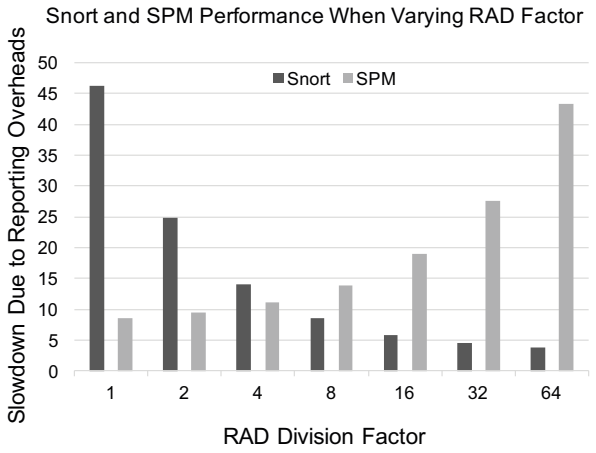

Figure 7: Reporting overheads as a function of increasing RAD factor for Snort and SPM. Snort has sparse reporting behavior, and thus benefits from smaller packets. SPM has dense reporting behavior, and benefits from larger packets.

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.

7.2.3 *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 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, 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.

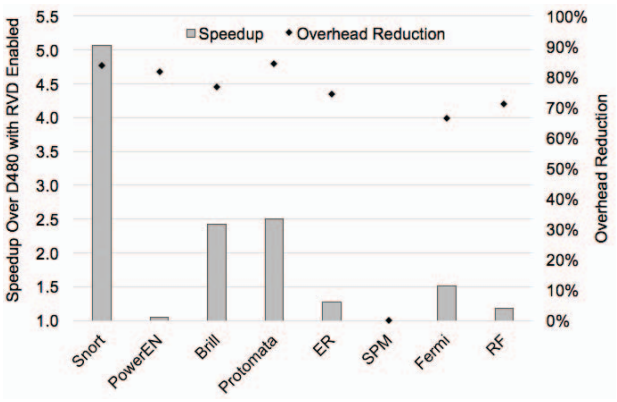

Figure 8: Speedups and reduction in reporting overhead due to RAD. SPM did not benefit from RAD because it generates dense reporting vectors.

8. DISCUSSION AND FUTURE WORK

This paper 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 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 [14], or classed report voting [16, 22].

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 this paper 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.

9. 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 paper first characterizes reporting behavior of automata applications. To the best of our knowledge, this paper 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 $5.1x$.

10. ACKNOWLEDGEMENTS

We would like to thank the reviewers for their valuable feedback. This work was partly funded by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, Achievement Rewards for College Scientists (ARCS), and NSF grant no. CCF-1629450.

11. REFERENCES

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, pp. 56–67, Oct. 2009.

[2] M. Roesch, "Snort: Lightweight intrusion detection for networks.," in *Proceedings of the USENIX Large Installation Systems Administration Conference (LISA)*, 1999.

[3] ClamAV, "ClamAV Rules." Available at `https://www.clamav.net/`.

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

[5] 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.

[6] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, "High Performance Pattern Matching Using the Automata Processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1123–1132, 2016.

[7] I. Roy, N. Jammula, and S. Aluru, "Algorithmic Techniques for Solving Graph Problems on the Automata Processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 283–292, May 2016.

[8] M. H. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda, "Using the Automata Processor for fast pattern recognition in high energy physics experiments—a proof of concept," *Nuclear Instruments and Methods in Physics Research*, 2016.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[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] 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.

[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] Intel, "Hyperscan." `https://github.com/01org/hyperscan`.

[19] Google, "Re2." `https://github.com/google/re2`.

[20] 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.

[21] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NFA Pattern Matching on GPGPU Devices," *SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010.

[22] 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.

[23] Micron Inc., "Designing for the Micron D480 Automata Processor." `http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html`.

[24] 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.

[25] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Washington, DC, USA), pp. 227–238, IEEE Computer Society, 2001.

[26] Y.-H. E. Yang, W. Jiang, and V. 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)*, (New York, NY, USA), pp. 30–39, ACM, 2008.

[27] M. Becchi, C. Wiseman, and P. 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)*, pp. 30–39, 2009.

[28] X. Wang, "Techniques for efficient regular expression matching across hardware architectures," Master's thesis, University of Missouri-Columbia, 2014.

[29] 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.

[30] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "Supplementary material for an efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, 2014.

[31] 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.

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