REAPR: Reconfigurable Engine for Automata Processing

Ted Xie[1], Vinh Dang[2], Jack Wadden[2], Kevin Skadron[2], Mircea Stan[1]
[1]Department of Electrical and Computer Engineering, University of Virginia
[2]Department of Computer Science, University of Virginia
{ted.xie, vqd8a, wadden, skadron, mircea}@virginia.edu

Abstract— **Finite automata have proven their usefulness in high-profile domains ranging from network security to machine learning. While prior work focused on their applicability for purely regular expression workloads such as antivirus and network security rulesets, recent research has shown that automata can optimize the performance for algorithms in other areas such as machine learning and even particle physics. Unfortunately, their emulation on traditional CPU architectures is fundamentally slow and further bottlenecked by memory. In this paper, we present REAPR: Reconfigurable Engine for Automata PRocessing, a flexible framework that synthesizes RTL for automata processing applications as well as I/O to handle data transfer to and from the kernel. We show that even with memory and control flow overheads, FPGAs still enable extremely high-throughput computation of automata workloads compared to other architectures.**

I. Introduction

Many modern applications essential to domains such as high-performance computing, network security, and machine learning benefit from an approach based on finite automata. For instance, in the case of network security, regular expressions used for deep packet inspection are represented in memory as their equivalent nondeterministic finite automata; RegEx engines such as Intel HyperScan [1] perform automata transformations to maintain excellent performance in terms of both runtime and memory utilization. In the field of machine learning, Tracy et al. [2] have shown that automata can even be used for a high performance implementation of the Random Forest classification algorithm.

Previously, engineers have relied on Moore's Law scaling for computing power to keep up with demands for these workloads, despite the fact that traditional von Neumann architectures are poorly suited for automata processing. Now in the "post-Moore" era, it is no longer prudent to rely on advances in manufacturing for improvements in computational throughput. The unsuitability of traditional CPU architectures for automata processing is amplified by the increasing demand for such applications; heterogeneous accelerator-based solutions are the way forward.

Specifically, prior work [3] [4] has shown that digital circuits enable a one-to-one spatial mapping between automata states and circuit components such as registers and wires. Due to the volatility of many automata processing workloads such as network security and machine learning, in which the rulesets and algorithms change frequently, non-reprogrammable application-specific integrated circuits (ASICs) are too inflexible and too costly to be commercially viable, and therefore reconfigurable platforms such as FPGAs and the Micron Automata Processor [4] are an ideal target for these workloads.

Our goal in this paper is to develop a high throughput and scalable engine for enabling *spatial automata processing* on FPGAs, as opposed to prior work which focused purely on regular expressions. To achieve this goal, we have developed REAPR (**R**econfigurable **E**ngine for **A**utomata **PR**ocessing), a flexible and parameterizable tool that generates RTL to emulate finite automata.

The major contributions of REAPR are the following:

- The first effort to characterize **automata performance** for applications other than regular expressions on the FPGA using the ANMLZoo benchmark suite [5];
- A head-to-head comparison of FPGA computational throughput against a best-effort CPU automata processing engine (Intel HyperScan) and the Micron Automata Processor;
- The first automata-to-RTL translation tool with I/O generation capability;
- To our knowledge, the first attempt to run an automata processing application (Random Forest) on *actual hardware* by using AXI and PCI-Express to enable CPU-FPGA communication;
- A maximally-sized Levenshtein automaton to determine peak state capacity with complex routing;

II. Prior Work

A. CPU Automata Engines

In a nondeterministic finite automaton (NFA), symbols from the input stream are broadcast to each state simultaneously, and each state connects to several other states, each of which may or may not activate depending on whether a given state matches the incoming symbol. For each symbol, an NFA engine must determine the next set of activated states, which involves a linear-time scan of the adjacency lists of all states in the current activated set. In the worst case, the adjacency list may contain nearly all of the states in the automaton; therefore, the run-time on a CPU for simulating an m-state automaton on n symbols is $O(n \cdot m)$, and for non-trivial data with non-trivial automata ($n = m$), the overall runtime is quadratic. CPU NFA processing is additionally hampered by the so-called "memory wall" due to the NFA's pointer-chasing execution model, and therefore it is desirable to drastically reduce the number of memory accesses per input item. In order to mask memory latency, state-of-the-art NFA engines such as Intel HyperScan [1] perform SIMD vector operations to execute as many state transitions as possible for a given memory transaction. Even so, such optimizations can not escape the fact that sequential von Neumann architectures are *fundamentally* ill-suited for these type of workloads.

In order to improve the run-time complexity of automata traversals, some regular expression engines transform the NFA into its equivalent deterministic finite automata (DFA). A DFA only has one state active for any given symbol cycle and is functionally equivalent to an NFA; this is achieved through a process known as *subset construction*, which involves enumerating all possible paths through an NFA. Converting an NFA to DFA has the benefit of reducing the runtime to $O(n)$ for n symbols (note that now the runtime is independent of automaton size) and only requires one memory access per input symbol, but frequently causes an exponential increase in the number of states necessary; this phenomenon is often referred to as *state explosion*. Subset construction for large automata incurs a huge memory footprint, which may actually cause performance degradation due to memory overhead in von Neumann machines.

Prior work by Becchi [6] attempted to leverage the best of both types of finite automata (the spatial density of NFA and temporal density of DFA). By intercepting the subset construction algorithm and not expanding paths that would result in a state explosion, Becchi achieved 98-99% reduction in memory capacity requirement and up to 10x reduction in memory transactions.

B. Automata on FPGA

1) NFA: Past implementations of NFAs on FPGA [3] [7] focused on synthesizing *only* regular expression matching circuits for applications such as antivirus file scanning and network intrusion detection. REAPR extends this prior work by focusing on a more diverse set of finite automata to address the fact that the workload for automata processing is much richer and more diverse than regular expressions. We extend the underlying approaches for NFA RTL generation from prior work, adapt it for other NFA applications, and detail our process in Section IV.

2) DFA: Several efforts [8] in accelerating automata processing with FPGAs use Aho-Corasick DFAs as the underlying data structures. A major motivator behind this design choice is the ease of translation between a DFA and a simple transition table, which is easily implemented using BRAM. One benefit to this approach is that BRAM contents can be hot-swapped easily, whereas a spatial design requires a full recompilation to realize even a single change. Because DFAs do not exploit the native bit-level parallelism in digital hardware and are much better suited to memory-bound CPU architectures, REAPR only focuses on the spatial implementation of NFAs.

C. The Micron Automata Processor

1) Architecture and Overview: The Micron Automata Processor (AP) [4] is a reconfigurable fabric for emulating finite automata in hardware, designed in a 50nm DRAM process. Its fundamental building block is the State Transition Element (STE), which is the hardware realization of an automaton state as well as the next-state transition logic. The first generation AP packs roughly 49,000 states per chip, and with 32 chips per board, one AP card holds nearly 1.6 million states. REAPR implements automata states in a similar manner but is more extensible due to the flexibility of the FPGA.

2) Programming Interface: In addition to a multi-lingual (C, Python, Java) SDK, Micron offers the Automata Network Markup Language (ANML), a format based upon XML for describing the interconnectivity of AP components such as STEs, booleans, and counters. Developers can either generate their own ANML files or generate them from regular expressions using the SDK's `apcompile` command [9].

D. Other Architectures

Several past efforts have proposed modifications to existing von Neumann architectures to specifically increase performance of automata processing workloads. HARE (Hardware Accelerator for Regular Expressions) [10] uses an array of parallel modified RISC processors to emulate the Aho-Corasick DFA representation of regular expression rulesets. The Unified Automata Processor (UAP) [11] also uses an array of parallel processors to execute automata transitions and can emulate *any* automaton, not just Aho-Corasick. However, because these works are 1) not FPGA-centric (both are ASICs), 2) based on the von Neumann model and not spatially distributed like REAPR, and 3) confined to a limited set of just regular expressions (as opposed to general automata applications), we do not directly compare to them.

III. Benchmarks

We synthesize the ANMLZoo [5] automata benchmark suite developed by Wadden et al. to determine the efficiency of REAPR. ANMLZoo contains several applications falling into three broad categories: regular expressions, widgets, and mesh. The applications, along with their categories, are listed below in Table I. Detailed descriptions of these benchmarks can be found in the ANMLZoo paper [5].

Benchmark Name	**Category**	**States**
Snort	RegEx	69,029
Dotstar	RegEx	96,438
ClamAV	RegEx	49,538
PowerEN	RegEx	40,513
Brill Tagging	RegEx	42,658
Protomata	RegEx	42,009
Hamming Distance	Mesh	11,346
Levenshtein Distance*	Mesh	2,784
Entity Resolution	Widget	95,136
Sequential Pattern Mining (SPM)	Widget	100,500
Fermi	Widget	40,738
Random Forest	Widget	33,220

TABLE I: ANMLZoo is a benchmark suite for automata engines, suitable for many platforms such as traditional CPUs, GPUs, FPGAs, and the Micron Automata Processor. *ANMLZoo's Levenshtein benchmark is actually three 24x20 automata, each with 928 states.

ANMLZoo is normalized for one AP chip, so these benchmarks synthesized for the FPGA will provide a direct comparison of equivalent kernel performance between the two platforms.

A. *Maximally-Sized Levenshtein Automaton*

In addition to comparing the relative performance of the AP versus an FPGA, it is also useful to know exactly what the upper bounds are for FPGA capacity. For this reason, we resize the Levenshtein benchmark such that it completely saturates the FPGA's on-chip LUT resources. We have chosen Levenshtein specifically because it is the smallest and therefore worst-performing application in ANMLZoo, due to the clash between its 2D-mesh topology and the AP's tree-based routing. The poor routing can be observed in the fact that Levenshtein has the smallest number of states in ANMLZoo, thus wasting the most computational potential. We believe that Levenshtein represents an application that not only is inefficient on the AP, but is very well-suited to the FPGA and its 2D-mesh routing network.

IV. RTL Generation

This work focuses mainly on the hardware synthesis of *nondeterministic* finite automata rather than DFA. The NFA's highly parallel operation of matching one single datum for many states ("Multiple Instruction Single Data" in Flynn's taxonomy) maps very well to the abundant parallelism offered by spatial architectures such as the FPGA and AP. While DFAs can also be implemented spatially, the argument is less compelling because 1) DFAs only need to perform a single symbol match per cycle, and therefore are better suited for von Neumann architectures and 2) DFAs often have a huge area requirement.

Spatial architectures implement automata states as transition logic ANDed with a single register representing whether the state is activated. This is the case for the AP as well as prior work [3] [7]. In the case of REAPR and the AP, the transition logic is actually merged with the state to transform a traditional NFA into a *homogeneous finite automaton* [12]. In these homogeneous FAs, the combined state-transition structure is referred to as a *state-transition element* (STE). Each STE's transition logic is one-hot encoded as a 1x256 memory column (the "character class") and is ANDed with the activation state register, the input to which is the reduction OR of enable signals coming from other states. With this design, a single STE will only output "1" when its activation state is driven high by other states *and* the current symbol is accepted in its character class. Algorithm 1 describes this process and Figure 1 shows a visual representation of it.

We propose two design methodologies to represent character classes in hardware using either the FPGA's lookup tables (LUTs) or BRAM.

A. *LUT-Based Design*

Each state must accept a range of characters corresponding to outgoing transitions in a canonical finite automaton. LUTs are well-suited for this task, due to their proximity to the state

Algorithm 1 NFA-RTL Translation Algorithm

```
 1: procedure NFA2RTL(incoming_symbol)
 2:     for each STE do
 3:         generate DFF dff
 4:         generate 1bx256 character class RAM cc
 5:         generate 1b signal activated
 6:         for each incoming iSTE do
 7:             activated |= iSTE.output
 8:         end for
 9:         generate 1b signal char_matches
10:         char_matches = cc[incoming_symbol]
11:         generate 1b output_signal output
12:         output= char_matches AND activated
13:     end for
14: end procedure
```

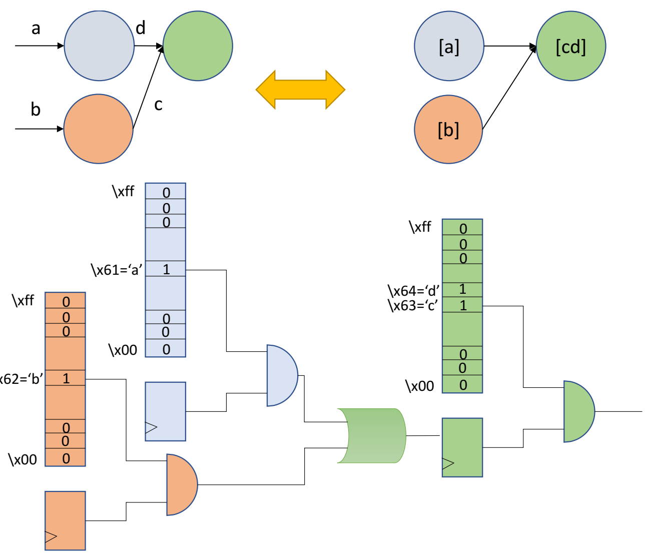

Fig. 1: Automata states can be easily mapped to registers and look-up tables ("logic").

registers within a CLB; a LUT-based flow will not need to use as much long-distance wiring to connect to a far-away BRAM.

B. *BRAM-Based Design*

The main disadvantage of using LUTs for storing the character class is the long compilation time; FPGA compilers aggressively minimize logic for LUT designs, which drastically increases compiler effort. Using BRAMs for transition logic circumvents the expensive optimization step and therefore significantly decreases compile time.

The AP's approach to generating hardware NFAs is very similar to the BRAM design, except that Micron stores the 256-bit columns into DRAM banks instead of FPGA BRAM. This has the benefit of high state density due to the higher density of DRAM compared to SRAM.

C. *I/O*

Prior works considered only *kernel* performance rather than *system* performance. While this approach has the ben-

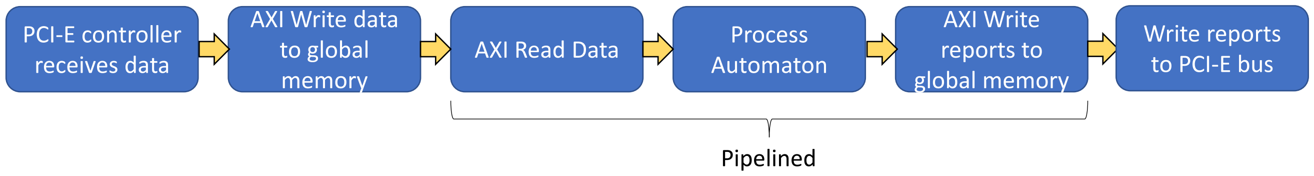
PCI-E controller receives data
AXI Write data to global memory
AXI Read Data
Process Automaton
AXI Write reports to global memory
Write reports to PCI-E bus
Pipelined

Fig. 2: Execution flow of AXI and PCI-Express transactions for automata processing kernels.

efit of greatly reducing the implementation difficulty of a research project, it does not provide a full analysis because real systems are not I/O-agnostic. A main contribution of REAPR is the inclusion of I/O circuitry over PCI-Express and AXI for the Random Forest benchmark, making REAPR the first work to offer a truly end-to-end automata accelerator design flow for FPGAs.

We adopt a high level synthesis (HLS)-centric approach by designing the I/O *interface* using HLS and modifying the generated Verilog code to integrate our automata kernels. Xilinx SDAccel [13] then generates AXI and PCI-Express circuitry for our kernels. Testing automata circuits with real data on real hardware allows us to obtain more realistic benchmark results compared to simulations, as prior works have done. The overall execution flow of REAPR with I/O is shown in Figure 2.

To integrate our RTL kernels into HLS-generated Verilog, we design the I/O kernel to have some very basic dummy computation. A simplified code snippet is shown in Listing 1, which shows data being copied from the input buffer to the output buffer after being added to 0xFA. In the Verilog, we can search for this dummy addition, and substitute the addition operation with our automata RTL kernel.

Listing 1: I/O kernel with dummy computation

```
void io_kernel(din* indata, dout* outdata) {
    for (int i=0; i<DATA_SIZE; i++) {
        outdata[i] = indata[i] + 0xFA;
    }
}
```

D. Reporting

A major challenge to implementing automata on FPGAs is not in the kernel itself, but rather in the I/O. For every 8-bit symbol processed by REAPR, thousands of reports may fire, requiring per-cycle storage on the order of kilo-bits. This massive amount of data transfer has a non-negligible overhead on overall throughput.

To illustrate the detrimental effects of I/O on performance, consider the following example. In the Random Forest application, there are 1,661 reporting states corresponding to 10 feature classifications [2]. The host CPU post-processes this report data, so all of it must be preserved. A 10 MB input file will therefore generate 16.61 GB worth of output signals. Assuming 250 MHz kernel clock rate and a 10 GBps PCI-Express link with a single-stream blocking control flow, the overall end-to-end throughput of the system can be expressed as follows:

$$Throughput = \frac{10MB}{\frac{10MB}{10GBps} + \frac{10MB}{250MBps} + \frac{16.61GB}{10GBps}}$$

Evaluating the above expression gives an overall throughput of just **5.8 MBps**, only about 20% of the expected 250 MBps. Efficient reporting is therefore a crucial part of developing high performance automata processing kernels.

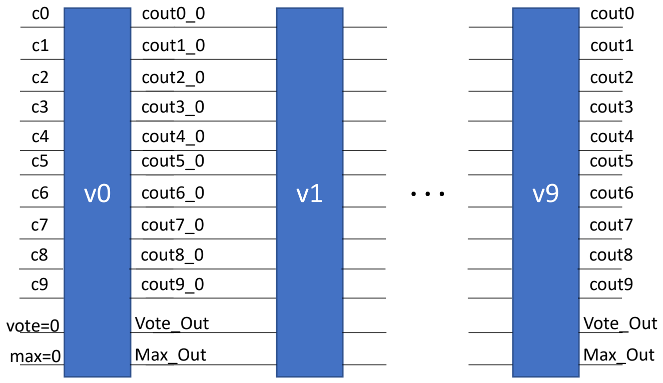

Fig. 3: The pipelined voter module for Random Forest compresses the output size from 1,661 to just 8 bits.

To demonstrate an example of efficient report processing, we delegate the voting stage of the Random Forest algorithm *on-chip* so that instead of exporting 1,661 bits of report information per cycle, we can just export the vote instead. The Random Forest ("RF") kernel in the ANMLZoo benchmark suite is trained for the MNIST hand-writing database for digits 0-9 [2], so only four bits are necessary to encode the vote per cycle. However, because the minimum word width of SDAccel is 8 bits (one byte), we set the vote output width to be 8 bits instead. This enables a factor of 207 reduction in necessary report storage compared to the original 1,661 bits.

Each of the report bits in the RF kernel corresponds to one of ten possible feature classifications. The voter module, shown in Figure 3, contains ten identical stages. Each voter stage v_i takes as input 10 classification vectors (c_0 - c_9), the determined vote from the previous stage ($vote$), and the number of votes corresponding to that classification (max). Each stage i will calculate the Hamming Weight w of classification vector c_i and compare that to max. If $w > max$, then the current stage passes i as $vote$ and w as max. All of the classification vectors c_i are passed to the next stage. Because the throughput of this voter module is one vote per cycle, it has no negative impact on the overall throughput of the Random Forest kernel.

V. Evaluation

All FPGA metrics were obtained for the Xilinx Kintex UltraScale 060 FPGA (Alpha Data ADM-PCIE-KU3 board) with an X16 PCI-Express interface, 2,160 18 Kb BRAMs and 331k CLB LUTs. The FPGA's host computer has a quad-core Intel Core i7-4820k CPU running at 3.70 GHz and 32 GB of 1866 MHz DDR3 RAM. CPU performance results were obtained on a six-core Intel Core i7-5820k running at 3.30 GHz with 32 GB of 2133 MHz DDR4 RAM.

To obtain the synthesis and place & route results, we use Xilinx Vivado's *Out of Context* (OOC) synthesis and implementation feature. OOC allows us to synthesize RTL designs for which the number of pins exceeds the maximum number on our selected chip (1,156) in the absence of a general-purpose report-offloading architecture. For future work, we hope to implement such an architecture to obtain more confident data regarding throughput, power consumption, and resource utilization.

All CPU benchmark results are obtained by running a modified version of VASim [14] that uses Intel's HyperScan tool as its automata processing back-end and an ANML (instead of regular expression) parser as its front-end. We choose HyperScan as a general indicator of a state-of-the-art highly optimized CPU automata processing engine.

Because the AP and REAPR have similar run-time execution models and are both PCI-Express boards, we can safely disregard data transfer and control overheads to make general capacity and throughput comparisons between the two platforms. While in reality the I/O circuitry has a non-negligible effect on both capacity *and* performance for both platforms, we aim to draw high-level intuitions about the architectures rather than the minutia of reporting.

A. ANMLZoo Benchmark Results

Our primary figure of merit to quantify capacity is the CLB utilization for the FPGA chip. CLB usage is a function mainly of two variables: state complexity and routing complexity. Automata with very simple state character classes will require very few CLBs to implement. Similarly, very complexly routed applications (for instance, Levenshtein) have so many nets that the FPGA's dedicated routing blocks are insufficient so the compiler instead uses LUTs for routing. The CLB utilization can be observed in Figure 4.

CLB utilization ranges from 2-70% for the LUT-based design and 1.4-46% for the BRAM-based design. In most cases, using BRAM results in a net reduction in CLB utilization because the expensive state transition logic is stored in dedicated BRAM instead of distributed LUTs.

Figure 4 also shows the results of compiling ANMLZoo in the BRAM flavor. Theoretically, the total state capacity for BRAM automata designs is the number of states per 18 Kb BRAM cell multiplied by the number of cells. Ideally, we would be able to map transition logic to a 256-row by w-column block, where $w = \frac{18Kb}{256b} = 72$. The closest BRAM configuration we can use is 512×36, which means that we can only fit 36 256-bit column vectors into one BRAM cell instead of 72. Multiplying the number of states per cell (36) by the number of cells (2,160) gives a per-chip BRAM-based state capacity of **77,760**. Most applications' BRAM utilization is almost exactly their number of states divided by the total on-chip BRAM state capacity except for Dotstar, ER, and SPM. In these cases, the applications have more than the 77k allowable states for the BRAM design, so REAPR starts implementing states as LUTs after the limit is surpassed.

Figure 5 shows the state complexity in ANMLZoo applications, which ranges from 1-3 CLBs per state. While the complexity for logic-based automata varies dramatically based on the complexity of transition logic and enable signals (node in-degree), for BRAM it remains relatively consistent at roughly 1 CLB per state. Notable exceptions to this trend are Hamming, Levenshtein, and Entity Resolution. Hamming and Levenshtein are mesh-based automata with high routing congestion, and ER is so large that the on-chip BRAM resources are exhausted and LUTs are used to implement the remaining states.

We use Vivado's estimated maximum frequency (Fmax) to approximate throughput for REAPR, the results of which are displayed in Figure 6. Because the hardware NFA consumes one 8-bit symbol per cycle, the peak computational throughput will mirror the clock frequency. For ANMLZoo, REAPR is able to achieve between 222 MHz (SPM) and 686 MHz (Hamming Distance) corresponding to 222 MBps and 686 MBps throughput.

One interesting result of the estimated power analysis reported by Vivado (see Figure 7) is the observation that the BRAM implementation consumes much more power (1.6W - 28W) than the LUT designs (0.8W - 3.07W). The reason for this discrepancy is twofold: 1) BRAMs in general are much larger circuits than LUTs, and powering them at high frequencies is actually quite expensive; 2) routing to and from BRAM cells requires using many of the FPGA's larger long-distance wires which tend to dissipate more energy. In future work, we hope to program all of these BRAM-based circuits onto actual hardware and measure TDP to verify the power consumption.

Figure 8 shows the power efficiency of ANMLZoo applications, which we define as the ratio between throughput and power consumption. In all cases, the LUT-based designs are significantly more power efficient than the BRAM designs due to the much lower power consumption.

Using Fmax (without any reporting or I/O circuitry) as the computational throughput, we can determine the speedup (seen in Figure 9) against a high-end CPU running Intel HyperScan. In the worst case, REAPR is on par with HyperScan and in the best case achieves over a 2,000x speedup for the SPM application for both the BRAM- and LUT-based designs.

B. Random Forest with I/O Circuitry

Using the pipelined voter module, we are able to achieve an average throughput of **240 MBps** for the Random Forest kernel, including data transfer and control overheads. Compared to HyperScan's performance of 1.31 MBps for this

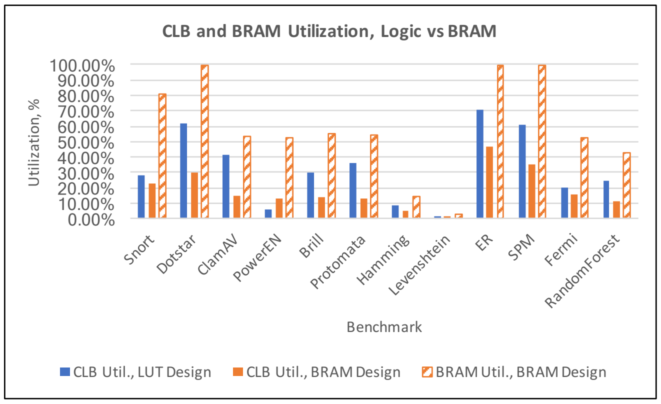

Fig. 4: CLB and BRAM utilization for ANMLZoo benchmarks using the LUT-based and BRAM-based design methodologies. Note that none of the benchmarks exceed more than 70% utilization, and that in most cases the BRAM-based design uses fewer CLBs.

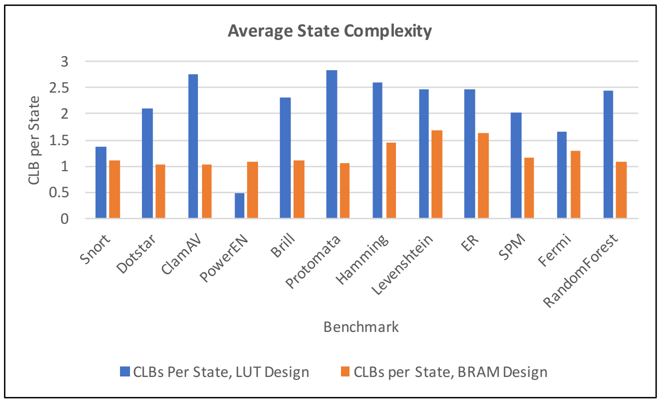

Fig. 5: The average state complexity for an automaton is the ratio of CLBs placed and routed versus the number of automaton states. Automata with very simple character classes intuitively need less logic (CLBs) than more complex automata.

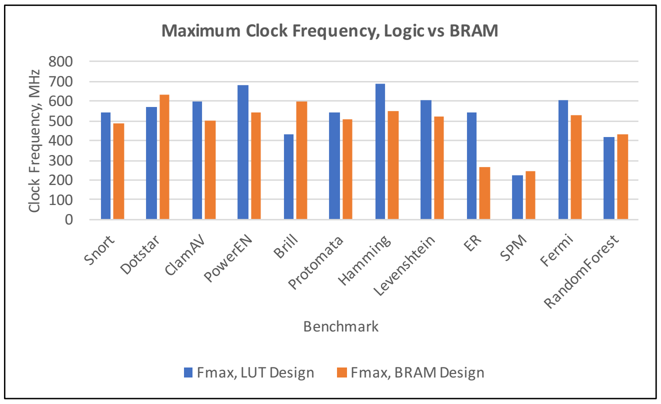

Fig. 6: Clock frequency in most cases is degraded when comparing LUT-based automata to BRAM-based, and in general ranges from 200 MHz to nearly 700 MHz.

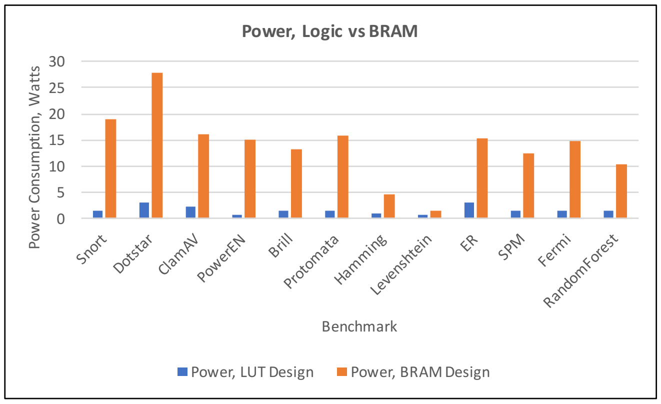

Fig. 7: Estimated power consumption for LUT NFAs is very low; the most power hungry application, Dotstar, is estimated to consume barely more than 3 W. The BRAM implementation is much more inefficient, peaking at above 25 W for three applications.

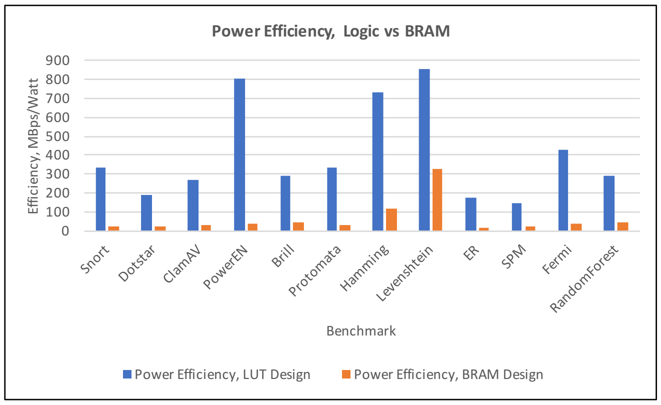

Fig. 8: Estimated power efficiency of the LUT-based design greatly outstrips that of BRAM. Mesh benchmarks (Hamming and Levenshtein) perform very well in this aspect.

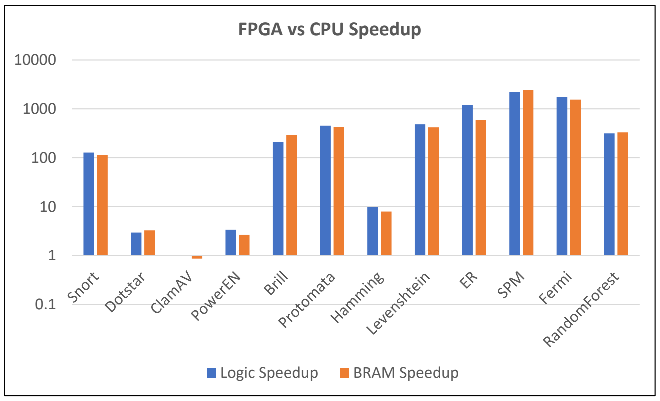

Fig. 9: Speedup ranges from 1.03x to 2,188x for LUT-based automata and 0.87x - 2,009x for BRAM-based automata.

application, we achieve a **183x** speedup on real hardware.

C. Maximally-Sized Levenshtein Automaton

To demonstrate the true power of FPGAs for automata processing, we have developed a new "standard candle" [1] for the Levenshtein benchmark using the approach described by Tracy et al. [15]. By generating and synthesizing larger and larger edit distance automata, we have discovered that for a distance of 20 (the same as the ANMLZoo Levenshtein), the longest Levenshtein kernel we can fit on our Kintex Ultrascale FPGA has a length of **1,550**, requiring 63,570 states. Compared to the 2,784 states in the 24x20 ANMLZoo Levenshtein benchmark, the FPGA achieves a **22.8x** improvement in per-chip capacity.

VI. Discussion

A. The Importance of I/O

Our implementation of Random Forest, including the pipelined voter mechanism, achieved 240 MBps overall throughput. This data point proves that I/O handling can have a substantial impact on overall system performance. By delegating the voting portion of the Random Forest algorithm on-chip, REAPR enables FPGA developers to achieve a 40.7x speedup over the estimated worst-case performance of 5.8 MBps. Moreover, the compacted output data stream allows the kernel to operate at 96% of its estimated 250 MBps throughput, indicating that I/O overheads are minimized with our approach.

Assuming that similar high performance reporting can be implemented for other benchmarks (i.e. these other protocols are also roughly 73% efficient), REAPR is still in the worst case roughly on-par with a best-effort CPU approach, and in the best case orders of magnitude faster. This verifies that FPGAs are an excellent platform for automata, and that future work should focus on efficient reporting protocols.

B. The Importance of Application and Platform Topology

In the case of the Hamming and Levenshtein benchmarks, both of which have 2D mesh topologies, the AP compiler was unable to efficiently place and route due to a clash with the AP's tree-like hierarchical routing network. Such a limitation does not exist on the FPGA, which has a 2D mesh routing topology, exemplified in the FPGA's 28x capacity improvement for Levenshtein compared to the AP. Additionally, Hamming and Levenshtein were among the two best-performing benchmarks in terms of power efficiency. Therefore, applications using 2D mesh-like automata are better suited for the closer-matching 2D routing network available on an FPGA.

C. Logic vs. BRAM

In general, using BRAM to hold state transition logic enables significant savings in terms of CLB utilization; in the BRAM design methodology, CLBs are only used for combining enable signals and in some cases routing rather than those two tasks as well as transition logic. In most ANMLZoo benchmarks except for the synthetic benchmark PowerEN, the overall CLB utilization decreases by an average of 16.33%. Similarly, the average state complexity is greatly improved (except for PowerEN), in some cases by as much as 2.7x. We suspect PowerEN is an outlier due to its high BRAM utilization and high routing complexity. The compiler is forced to route complex enable logic to far-away BRAMs, and doing so exhausts on-chip routing resources, so Vivado defaults to using LUTs as "pass-through" LUTs to successfully place and route the design.

Improved CLB utilization comes primarily at the cost of both maximum clock rate and power consumption. Routing to far-off block RAM cells requires using expensive long-distance wiring in the FPGA fabric, which causes clock speed to be degraded and power consumption to increase significantly. The effect can be observed in Figures 6 and 7.

If an engineer wants to fit as many states as possible into an FPGA, it would be ideal to use a combined LUT and BRAM approach. For applications where state capacity is a limiting factor, an engineer can pass arguments to REAPR to completely saturate BRAM first, and then start using LUTs to implement states after that. This feature in REAPR has already been employed to synthesize ANMLZoo benchmarks with more than 77k states when targeting BRAM. For future work we anticipate maximally sizing other benchmarks using both BRAM and LUTs.

D. FPGA Advantages Over the Micron Automata Processor

One FPGA chip offers significantly greater per-chip capacity compared to the first generation AP. Whereas one AP chip is maximally utilized for all ANMLZoo benchmarks, we have shown that FPGAs in the worst case are only filled to less than 70% of logic and 99.7% of BRAM, and in the best case only 2% of logic and 3.24% of BRAM are utilized. Simultaneously, FPGAs run at higher clock speeds (222 MHz - 686 MHz) for all ANMLZoo applications. Theoretically, the speedup of a high-end FPGA chip versus the AP ranges from 1.7x to 5.2x, disregarding the effects of I/O and reporting.

E. FPGA Disadvantages Compared to the Micron Automata Processor

Despite that FPGAs excel in *per-chip* capacity, their *per-board* capacity lags far behind the AP. Whereas an FPGA board such as the Alpha Data KU3 typically contains just one chip, the AP board contains 32. In an exceedingly large application, an automata developer would need multiple FPGA boards whereas the AP compiler natively supports partitioning automata across multiple chips [9]. Assuming that the per-board cost is relatively similar for an AP and a high-end FPGA, then the AP has a significant *capacity-per-dollar* advantage over FPGAs. Furthermore, the AP can process multiple streams simultaneously on its many chips. In the best case, each chip may process its own stream, resulting in an aggregate throughput of **4.2 GBps**. For the same form factor, an AP board is capable of achieving

[1]A standard candle in the context of spatial automata processing is an automaton that completely saturates on-chip resources.

roughly 6x the performance of one FPGA board. This is especially important because datacenters typically optimize their hardware purchase decisions based on total cost of ownership (TCO), and the AP's significant advantage in multi-chip capacity and throughput makes it an excellent platform if the datacenter wishes to specialize some nodes for automata processing.

Another important metric for datacenter-scale deployment is productivity. Compiling the ANMLZoo applications requires on average about 10 hours for the LUT-based designs and 5 hours for the BRAM-based designs. Static applications easily tolerate this long implementation latency, but latency-sensitive domains like network security and machine learning can not. In the example of network security, a 10-hour downtime when fixing a zero-day vulnerability is completely unacceptable. Meanwhile, compiling these ANMLZoo benchmarks with the AP tools takes only minutes, orders of magnitude faster than the FPGA compilation. This can be attributed to the fact that the AP is specialized for automata processing, so there are fewer degrees of freedom for the compiler to consider.

F. Normalizing for Process Node

The AP is designed in 50 nm DRAM while our Kintex Ultrascale FPGA is based on a 20 nm SRAM process, roughly 2.5 ITRS generations ahead. To compare against the AP fairly, we can project expected capacity for a next-generation AP manufactured in a similar process, albeit for DRAM. With 2x transistor density increases per generation, the same chip area has 5.7x the capacity of the 50 nm AP. Therefore, an AP made in a modern process theoretically could pack 285k states in one chip, or roughly 9.1 million per board.

Per-chip capacity is additionally affected by the overall chip size. Judging by the package sizes, an FPGA chip is much larger than an AP chip, and therefore is able to fit more states simply due to its larger area. State capacity per unit area for both platforms would have been a very informative metric, but unfortunately the die size of our FPGA is not available online, so we are unable to make this comparison.

VII. Conclusion

In this paper we presented REAPR, a tool that generates RTL and I/O circuitry for automata processing. Using REAPR, we showed that the spatial representation of nondeterministic finite automata intuitively maps to spatial reconfigurable hardware, and that these circuits offer extremely high performance on an FPGA compared to a best-effort CPU automata processing engine (up to 2,188x faster). We compared REAPR's performance to a similar spatial architecture, the Micron Automata Processor (AP), in terms of capacity and throughput, and found that generally the FPGA outperforms the AP in both of those areas on a per-chip basis. However, since there are many chips per AP board, the Micron product outperforms the FPGA on a per-board basis.

We analyzed two different methods of generating automata RTL: LUT-based and BRAM-based, and found that LUT representations are more compact and lower power, and that BRAM designs are faster to compile. We determined that for Levenshtein distance, the FPGA is capable of achieving over 28x higher capacity than the AP, and that an application-specific reporting protocol for Random Forest on FPGA resulted in a 183x speedup over the CPU. In summary, we have extended prior work about regular expressions on FPGAs and extended it for a more diverse set of finite automata to show how FPGAs are efficient for automata applications other than regular expressions.

VIII. Acknowledgements

This work was supported in part by NSF grant no. CCF-1629450, a grant from Xilinx, and support from C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] Intel Corporation, "HyperScan - High Performance Regular Expression Matching Library," https://github.com/01org/hyperscan [Online].

[2] T. Tracy II, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards Machine Learning on the Automata Processor," in *International Supercomputing Conference-High Performance Computing*, pp. 200-218, 2016.

[3] Y. Yang and V. Prasanna, "High-Performance and Compact Architecture for Regular Expression Matching on FPGA," in *IEEE Transactions on Computers*, vol. 61, iss. 7, pp. 1013-1025, 2012.

[4] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Automata Processing," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, iss. 12, pp. 3088-3098, 2014.

[5] 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*, 2016.

[6] M. Becchi and P. Crowley, "A Hybrid Finite Automaton for Practical Deep Packet Inspection," in *ACM Conference on Networking Experiments and Technologies (CoNEXT)*, 2007.

[7] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[8] T. Hieu and T. Thinh, "A Memory Efficient FPGA-based Pattern Matching Engine for Stateful NIDS," in *IEEE International Conference on Ubiquitous and Future Networks (ICUFN)*, 2013.

[9] "Micron Automata Processor: Developer Portal", http://micronautomata.com/ [Online].

[10] V. Gogte, A. Kolli, M. Cafarella, L. D'Antoni, and T. Wenisch, "HARE: Hardware Accelerator for Regular Expressions," in *IEEE/ACM International Symposium on Microarchitecture*, Taipei, 2016, pp. 1-12.

[11] Y. Fang, T. Hoang, M. Becchi, and A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *International Symposium on Microarchitecture (MICRO)*, 2015.

[12] P. Caron and D. Ziadi. Characterization of Glushkov automata. Theoretical Computer Science, 233(1):7590, 2000.

[13] "SDAccel Development Environment," https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html [Online].

[14] J. Wadden, "Virtual Automata Simulator - VASim," https://github.com/jackwadden/vasim [Online].

[15] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins, "Nondeterministic Finite Automata in Hardware - the Case of the Levenshtein Automaton," in *International Workshop on Architectures and Systems for Big Data*, 2015.