FPGA Automata Processing

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science (Computer Engineering)

by

Theodoric Yang Xie

December 2017

Approval Sheet

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

Master of Science (Computer Engineering)

Theodoric Yang Xie

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

Mircea R. Stan, Adviser

Samira M. Khan, Committee Chair

Kevin Skadron

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

December 2017

d

Abstract

Dwindling inter-generational CPU performance and power consumption improvements previously made possible by semiconductor scaling motivate *hardware specialization* for a wide variety of tasks. In recent years, implementing certain algorithms as specialized circuits ("accelerators") has been proven to improve both speediness and power/energy efficiency compared to the equivalent CPU implementation. One particular domain that shows great promise for hardware specialization is **automata processing**. Finite automata are most commonly known as the back-end data structures for regular expressions, which are used in a wide variety of applications such as antivirus file scanning and packet payload inspection for network intrusion detection systems (NIDS). Several research efforts have extended the applicability of finite automata beyond just regular expression into domains such as machine learning, particle physics, bioinformatics, and pattern mining. The versatility of automata processing as well as its inefficiency on traditional von Neumann computer architectures informs the need for a flexible and high-performance accelerator for these applications.

In this thesis, an FPGA-based automata processing hardware accelerator is implemented in two different configurations: (1) a traditional discrete FPGA accelerator board attached over PCI-Express; and (2) a new tightly-coupled cache-coherent FPGA accelerator architecture utilizing the Intel Broadwell Xeon CPU + Arria 10 FPGA platform, known as the Hardware Accelerator Research Platform ("HARP").

Acknowledgments

I have been at UVA for about six years now, including my undergraduate education. I started out as a clueless high school graduate, and now I can finally contribute something to the computer engineering community with the contents of this master's thesis. I could not have reached this point without help and support from a huge number of people.

First and foremost, all three of my committee members (Dr. Mircea R. Stan, Dr. Kevin Skadron, Dr. Samira Khan) have all at some point motivated me to become a better researcher, engineer, or computer architect. Your enthusiasm and expertise has been truly inspirational for me as a young engineer.

I would additionally like to thank the fellow members of HPLP, CAP, and LAVA for their constant support and mentorship through the years - Jack Wadden, Sergiu Mosanu, Mateja Putic, Tommy Tracy, Vinh Dang, Chunkun Bo. Thank you all for not only being amazing colleagues, but friends as well!

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. Finally, I would like to thank Intel Corporation and Synopsys, Inc. for access to their professional-grade design tools, without which none of this work would have been remotely possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, finite automata have only been used as the back-end data structures for regular expression engines such as Google RE2 [1] and Intel HyperScan [2]. For decades, automata have faithfully served high-profile applications in domains such as deep packet inspection [3] [4] and antivirus file scanning [5] solely for the purpose of string pattern matching. However, recent research efforts have shown that automata processing can benefit other domains as well, including machine learning [6], pattern mining [7], bioinformatics [8], and even particle physics [9].

As companies and consumers alike generate more and more data, these automata processing applications must be able to scale their performance to meet the industry's needs. In the past, computer engineers have relied on generational improvements in CPU performance to keep up with demand, but as Moore's Law generational advances grind to a halt, this passive approach to scalability and high performance have become untenable. More drastic changes to system architecture are necessary if the industry wants to keep pace with an ever-growing quantity of data.

Prior works [10] [11] have shown that *spatial reconfigurable fabrics* such as Field-Programmable Gate Arrays (FPGAs) can offer significantly higher throughput over regular von Neumann CPUs for automata processing. The industry has realized the market potential for accelerating automata algorithms, and to this end, Micron has unveiled their Automata Processor [12], a course-grained spatial reconfigurable fabric tailored for automata workloads.

This thesis hypothesizes that FPGAs can offer significant speedup over a high-end CPU, even when accounting for data transfer and control overheads. To verify this claim, several tools are proposed that generate whole FPGA accelerator systems for automata processing, and performance metrics like processing throughput and spatial resource utilization will be analyzed.

1.1 Contributions

These are the main contributions of this thesis:

- A high-performance RTL generation tool for automata processing workloads on the FPGA using both LUT-based and BRAM-based designs called REAPR (Reconfigurable Engine for Automata Processing);
- The first effort to characterize automata performance on FPGAs beyond just regular expressions using REAPR, the results of which have been published in the Proceedings of the 27th International Conference on Field Programmable Logic [13];
- The first effort to characterize and optimize automata performance *with report offloading* for both traditional discrete accelerator systems and integrated ones.

1.2 Organization

The rest of this thesis is organized as follows:

- Chapter 2 provides background information about ideas and concepts frequently referred to through this work. This will include some basic theory of computation (regular expressions and automata theory), a brief overview of FPGA architecture, and modern FPGA platforms, including the two platforms explored in this thesis;
- Chapter 3 summarizes past efforts to accelerate automata processing on multiple platforms, including CPU and FPGA;
- Chapter 4 presents a tool called REAPR that generates automata FPGA kernels and analyzes REAPR's capability to accelerate automata processing on a discrete FPGA board using Xilinx SDAccel;
- Chapter 5 presents an extension of the REAPR work that utilizes both the CPU and FPGA of the Intel Broadwell + FPGA multi-chip module to co-process automata processing workloads, as well as a refined general purpose automata reporting architecture;
- Chapter 6 summarizes the thesis;
- Finally, Chapter 7 provides a list of future research directions inspired by the work in this thesis.

Chapter 2

Background Information

2.1 Regular Expressions and Automata Processing

Regular expressions ("regex") are special strings used to identify text patterns. They are typically composed of a series of literal characters that define sub-patterns along with some control characters that specify information like how many times a sub-pattern may occur. One of the most popular ways to represent regular expressions, and the one used in this thesis, is the Perl-Compatible Regular Expression ("PCRE"). Regex are widely used in a variety of applications ranging from text editors to bioinformatics; many problems involving searching and/or replacing a known text pattern can be solved using regex.

The back-end data structure for these text pattern matching engines is usually some kind of finite automaton [14]. Informally, finite automata are state machine-like structures that accept regular languages. Automata can be generated from regular expressions using algorithms such as the McNaughton-Yamada-Thompson Construction [11] or Gluschkov's algorithm [15]. There are two main types of finite automata: non-deterministic (NFA) and deterministic (DFA). NFAs can have multiple states active at any one time, allowing multiple simultaneous searches to take place at once for the same given input symbol. DFAs can only have one state active at a time, and are typically constructed from NFAs using a technique like subset construction. In some cases, translating an NFA to a DFA can result in the number of states increasing dramatically, a phenomenon known as "state explosion". Figure 2.1 shows the NFA of the regular expression `/a(a|b)a(a|b)a/`.

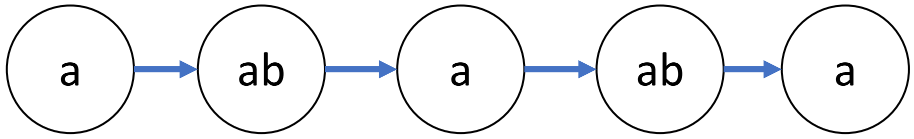

Figure 2.1: The NFA that implements the regular expression /a(a|b)a(a|b)a/.

2.2 FPGA Basics

Field-programmable gate arrays ("FPGAs") are integrated circuits whose configuration can be altered at run-time. Historically, these chips were first used to prototype integrated circuits or to implement "glue logic" between discrete digital circuit components, but since then have taken on a wide variety of roles. Modern FPGAs have even been used in place of application-specific integrated circuits (ASICs) for certain tasks due to their shorter time-to-market and high availability.

An FPGA is commonly thought of as a "sea of logic gates," where both the behavior and topology of these gates can be changed arbitrarily in the field. This functionality is achieved through a combination of two technological innovations: 1) look-up tables ("LUTs") for implementing n-input logic gates; and 2) switch boxes/connection boxes for implementing arbitrary connectivity between LUTs. In the "sea of gates" analogy, LUTs can be thought of as islands of activity and connection/switch boxes can be thought of as water between the islands that pass information back and forth.

An FPGA typically contains a 2-D array of so-called "logic blocks," each of which contains one or more LUTs along with some registers and multiplexers. These logic blocks, referred to as configurable logic blocks (CLBs) or adaptive logic modules (ALMs) by Xilinx and Intel respectively, are arranged in a 2-D array throughout the chip. Connection boxes ("CBs") connect logic blocks to each other while switch boxes ("SBs") connect connection boxes to each other. By changing the configurations of these logic, connection, and switch boxes, an FPGA engineer can implement almost any digital circuit with arbitrary behavior and interconnectivity. In addition to these resources, modern FPGAs will also contain other elements such as hard transceiver blocks (PCI-Express, DDR, QSFP), digital signal processors, or configurable high-capacity embedded memories called "block RAMs" or "BRAMs".

Historically, FPGAs have been programmed using register-transfer level (RTL) hardware description languages (HDLs) such as VHDL or Verilog. These languages enable developers to instantiate digital circuit components such as logic gates and memory cells and specify the wire interconnections between them. Compiling HDL code follows several main steps: Synthesis, Technology Mapping, Placement, and Routing.

- **Synthesis**: The compiler parses HDL code and infers what kind of circuit components have been instantiated (i.e. logic gates and registers) as well as their connectivity. During synthesis, the compiler will also typically perform some basic logic optimization to find the minimal form of each Boolean expression specified by the designer.

- **Technology mapping**: After primitive circuit components have been inferred, they must be translated to device-specific components. For example, a Boolean expression such as $a \cap b$ will be mapped into a 2-input LUT with the following contents: (0,0) = 0, (0,1) = 0, (1,0) = 0, (1,1) = 1.
- **Placement**: Given a list of populated LUTs, the compiler must then pack them into the FPGA chip's limited resources. This step reduces to the NP-hard *bin packing* problem. One common algorithm for placement is *simulated annealing*, where logic blocks are first randomly placed in the chip resources, then repeatedly swapped until the minimum cost configuration has been found.
- **Routing**: Once the placer has found a minimum cost placement, the compiler will attempt to find the minimum cost path connecting all of the circuit components in the desired configuration. This step generally uses a comparatively low-cost pathfinding algorithm such as Dijkstra's algorithm.

As FPGAs have moved beyond being just circuit prototyping platforms, companies like Xilinx and Intel have invested heavily in *high-level synthesis* languages, where standard procedural programming languages like C or OpenCL can be used to directly synthesize hardware, thus substantially lowering the barrier of entry for software developers who may not have extensive backgrounds in digital circuit design [16].

2.3 Modern FPGA Platforms

Today, Xilinx and Intel have realized the market potential for FPGA accelerator systems and have each released their own development platforms to bring FPGA performance and efficiency to the rest of the world. At first, these platforms were similar to GPGPU systems in that they were discrete chips with separate global memory linked to the CPU over PCI-Express. Recently, Intel has released a novel new tightly-coupled FPGA accelerator architecture called the Hardware Accelerator Research Platform ("HARP"), which is comprised of an Intel Broadwell 14-core Xeon CPU and an Intel Arria 10 FPGA. In this thesis, we will examine accelerator design experiences on a traditional discrete FPGA platform (Xilinx SDAccel) and the aforementioned Intel multi-chip module.

2.3.1 Xilinx SDAccel

SDAccel [17] is Xilinx's flagship product for bringing high-performance customizable hardware acceleration to the masses. At its core, SDAccel emulates the GPGPU development process - the company even advertises this product as the first GPU-like ecosystem for FPGA development. Specifically, the SDAccel software architecture closely mimics that of OpenCL in both its execution flow and syntax. On the host side, developers make familiar calls to Xilinx's flavor of the OpenCL host API. Much like with GPU OpenCL programming, the general host code flow is that some memory is allocated local to the CPU, then written to the target device's global memory, and finally read back into the CPU's local memory buffers. For kernel development, Xilinx offers support for RTL code (Verilog/VHDL), Xilinx high-level synthesis C, and even OpenCL.

As of late 2016, Xilinx has partnered with Amazon Web Services to bring SDAccel-compatible boards to AWS users via the Elastic Compute ("EC2") F1 service. These AWS FPGA instances use top-of-the-line Xilinx Virtex UltraScale+ series chips to complement AWS's existing high-performance nodes with GPUs and high-end CPUs.

2.3.2 Intel Broadwell + FPGA Multi-Chip Module

In 2015, Intel Corporation unveiled plans for a new high performance reconfigurable computing architecture that combined an enterprise-class Broadwell Xeon processor with a top-of-the-line Arria 10 FPGA [18]. While this is not the first product that closely coupled a CPU with an FPGA (many of Intel FPGA's embedded-class products combine a low-power ARM processor with a small Cyclone-series FPGA), this represents one of the first efforts to combine large enterprise-class chips through a shared last-level cache.

Similarly to SDAccel, this new product, known as the Hardware Accelerator Research Platform ("HARP"), allows the user to develop FPGA kernels with either RTL code, OpenCL, or Intel's SystemC-based high-level synthesis language. HARP's main CPU-FPGA communication bus is Intel's Core Cache Interface (CCI-P) [19]. CCI-P serves as a layer of abstraction on top of three communication busses, one of which is QPI and two of which are PCI-Express. CCI-P allows users to either manually select which of the three links they want to use or use the automatic channel selection mode, which opportunistically chooses which channel to send/receive information on.

When developing for HARP, the user is exposed to three main components: 1) the Xeon CPU host code; 2) the Intel-provided FPGA "blue bitstream"; and 3) the user-defined FPGA "green bitstream". The host-side code, similarly to the OpenCL execution flow, simply allocates memory regions and sends control signals to the FPGA. The FPGA blue bitstream is provided by Intel and serves as a cache-coherent shim that ineteracts with the Xeon processor's TLB and last-level cache on the user's behalf. Finally, the green bitstream is completely user-defined and contains some kind of computation logic along with some basic control logic to communicate with the blue bitstream, which in turn interacts with the Xeon CPU. A diagram of the HARP ecosystem can be seen in Figure 2.2.

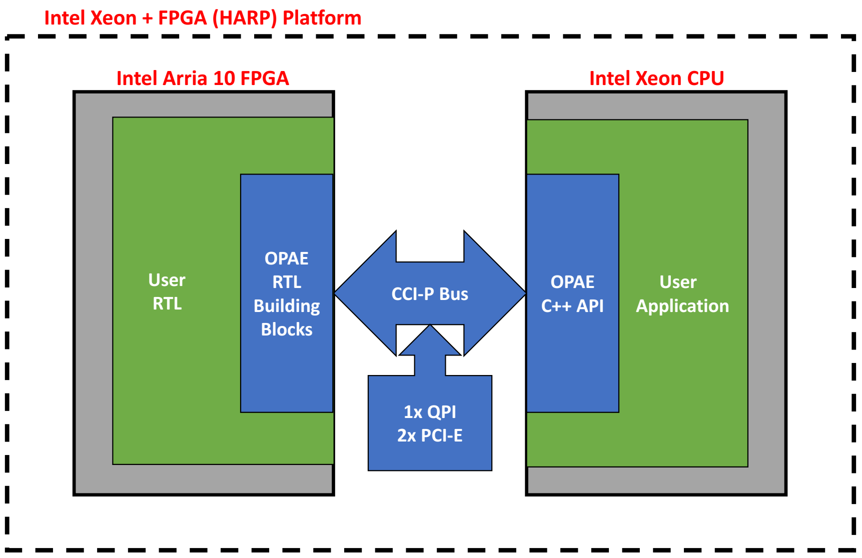

Figure 2.2: The overall HARP system topology. An Intel Xeon CPU is tightly coupled with an Intel Arria 10 FPGA through a cache-coherent interconnect link. THe green portions of the architecture are user-defined while the blue portions are static IP blocks used to implement features like the coherency protocol and firmware.

Chapter 3

Related Work

3.1 CPU Automata Processing 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 $O(n \cdot m)$. 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 [2] 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 [20] 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.

3.2 FPGA Automata Processing Engines

The projects described in this thesis are not the first efforts to accelerate automata processing using FPGAs. Several past research endeavours have also attempted to accelerate both NFA and DFA processing using reconfigurable fabrics.

3.2.1 NFA Engines

Past implementations of NFAs on FPGA [11] [10] 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 Chapter 4.

3.2.2 DFA engines

Several efforts [21] 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 abundant 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.

3.3 The Micron Automata Processor

In 2014, Micron Technologies unveiled a purpose-built automata accelerator appropriately named "The Automata Processor" [12], often referred to as the "AP". This product is designed in a 50nm DRAM process and computes one byte per cycle at a maximum frequency of 133 MHz for a throughput of 133 MBps (approximately 1 Gbps). Each Automata Processor populates a standard PCI-Express form factor with 32 chips per board, each of which contains roughly 49,000 automaton states per chip. The fundamental building block of the AP is the state transition element (STE). STEs are connected together with a hierarchical routing tree similarly to how the 2D routing mesh in an FPGA connects look-up tables together. In addition to automaton states, AP chips also have a limited number of special purpose elements such as up-counters and boolean logic gates.

To program the AP, Micron offers their AP software development kit, which enables users to write C++ host code and use a XML-based graph description language called "ANML" (Automata Network Markup Language) to design state machines which are implemented on the AP board. Developers can either create their own ANML files using the provided API or generate them from regular expressions using the SDK's `apcompile` command.

3.4 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) [22] uses an array of parallel modified RISC processors to emulate the Aho-Corasick DFA representation of regular expression rulesets. The Unified Automata Processor (UAP) [23] 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.

There have also been numerous efforts to process NFAs on GPUs [24] [25]. However, due to the mismatch between the GPGPU execution model (single instruction multiple data - SIMD) and that of the automata processing execution model (multiple instruction single data - MISD), GPUs are unable to achieve the same level of efficiency for NFA processing as spatial reconfigurable architectures like FPGAs and the AP do.

Chapter 4

REAPR: Reconfigurable Engine for Automata Processing

Prior work in accelerated automata processing was limited to just regular expressions and did not link the automata kernels with external high-speed communications interfaces such as PCI-Express. This thesis proposes REAPR (**R**econfigurable **E**ngine for **A**utomata **PR**ocessing to address these concerns. REAPR is an extensible and high-performance tool for generating automata processing accelerators on the FPGA, including rudimentary support for report offloading. In this work, we use REAPR to evaluate the potential of FPGAs to accelerate automata workloads besides just regular expressions. To do so, we examine a series of known automata benchmarks, attempt to find a realistic STE capacity estimate for the maximum capacity of our Xilinx Kintex UltraScale chip, and measure the processing throughput of the automata implementation of the Random Forest inference algorithm.

4.1 Benchmarks

We synthesize the ANMLZoo [26] 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 4.1. Detailed descriptions of these benchmarks can be found in the ANMLZoo paper [26].

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.

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 4.1: ANMLZoo is a benchmark suite for automata engines, suitable for many platforms such as traditional CPUs, GPUs, FPGAs, and the Micron Automata Processor. Note that the state counts listed in this table are for the raw, unoptimized ANML files.

4.1.1 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.

4.2 RTL Code 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 [11] [10]. 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* [15]. 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 4.1 shows a visual representation of it.

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
```

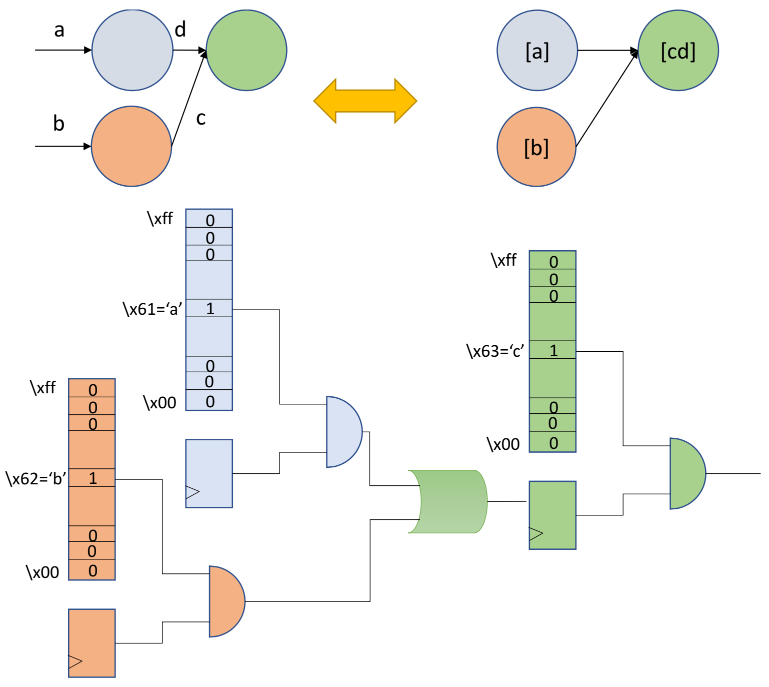

Figure 4.1: Automata states can be easily mapped to registers and look-up tables ("logic"). The regular expression that this NFA implements is `/(a|b)[cd]/`.

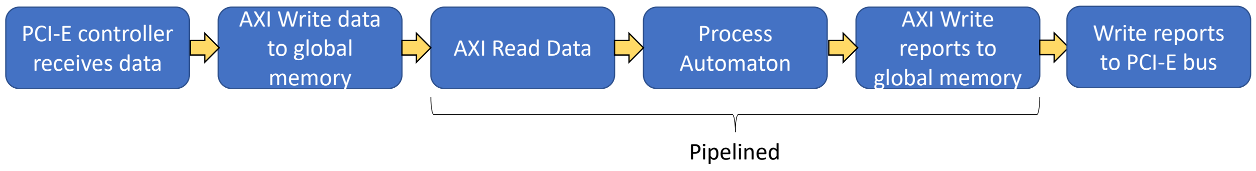
PCI-E controller receives data
AXI Write data to global memory
AXI Read Data
Process Automaton
AXI Write reports to global memory
Write reports to PCI-E bus
Pipelined

Figure 4.2: Execution flow of AXI and PCI-Express transactions for automata processing kernels.

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

4.2.1 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 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.

4.2.2 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.

4.2.3 I/O

Prior works considered only *kernel* performance rather than *system* performance. While this approach has the benefit 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 [17] 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, which prior works have relied on. The overall execution flow of REAPR with I/O is shown in Figure 4.2.

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 4.1, which shows data being copied from the input buffer to the output buffer after being added to 0xFA. In the generated Verilog (Listing 4.2), we can search for this dummy addition, and substitute the addition operation with our automata RTL kernel (Listing 4.3).

Listing 4.1: 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;
    }
}
```

Listing 4.2: Snippet of generated dummy computation Verilog code.

```
assign resultAB_fu_171_p2 = ($signed(loadAB_reg_239) + $signed(8'd250));
```

Listing 4.3: Code modifications necessary to hook in automata processing kernel.

```
//assign resultAB_fu_171_p2 = ($signed(loadAB_reg_239) + $signed(8'd250));
wire [7:0]   automata_indata;
wire [7:0] automata_reports;

assign automata_indata = loadAB_reg_239;

my_automata automata_U(
    .clock(ap_clk),
    .reset(1'b0),
    .run(1'b1),
    .data_in(automata_indata),
    .reports(automata_reports)
);
```

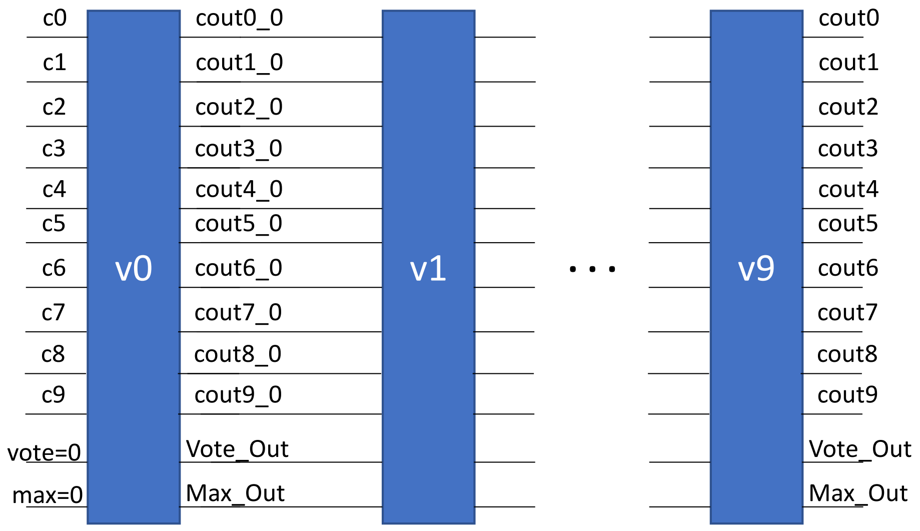

Figure 4.3: The pipelined voter module for Random Forest compresses the output size from 1,661 to just 8 bits.

4.2.4 Reporting Architecture (Match Output Offloading)

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 [6]. 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 **41.2 MBps**, only about 16% of the expected 250 MBps. Efficient reporting is therefore a crucial part of developing high performance automata processing kernels.

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 [6], 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 4.3, 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.

4.3 Evaluation Methodology

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 [27] 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.

4.4 Results

4.4.1 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 4.4.

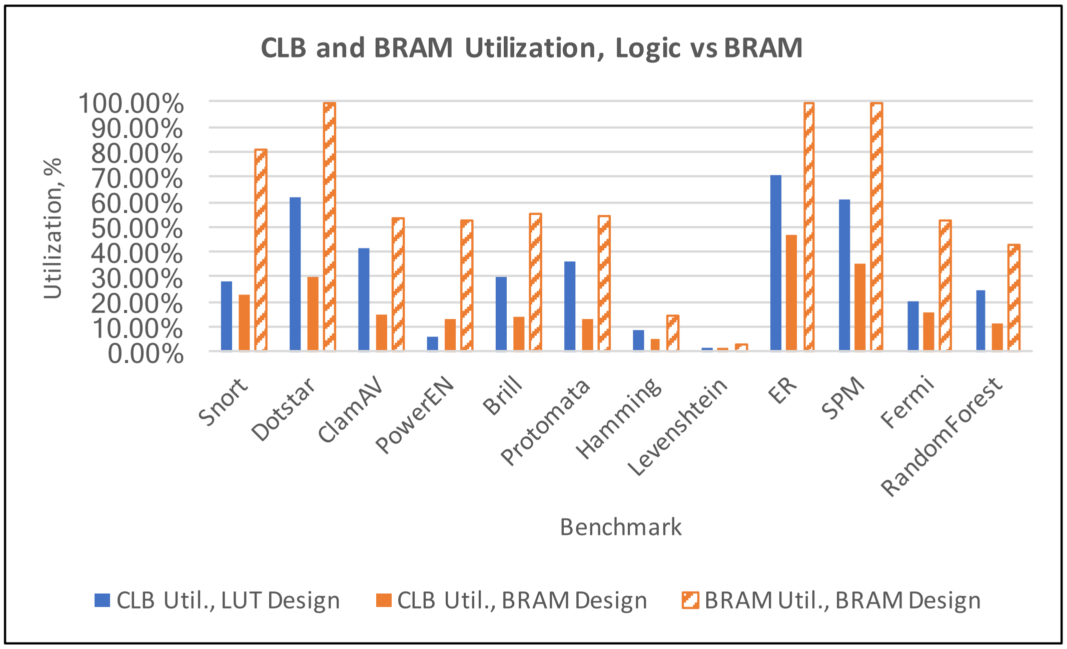

Figure 4.4: CLB and BRAM utilization for ANMLZoo benchmarks using the LUT-based and BRAM-based design methodologies. Note that none of the benchmarks exceed more than 70% utilization, and that in most cases the BRAM-based design uses fewer CLBs.

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 4.4 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.

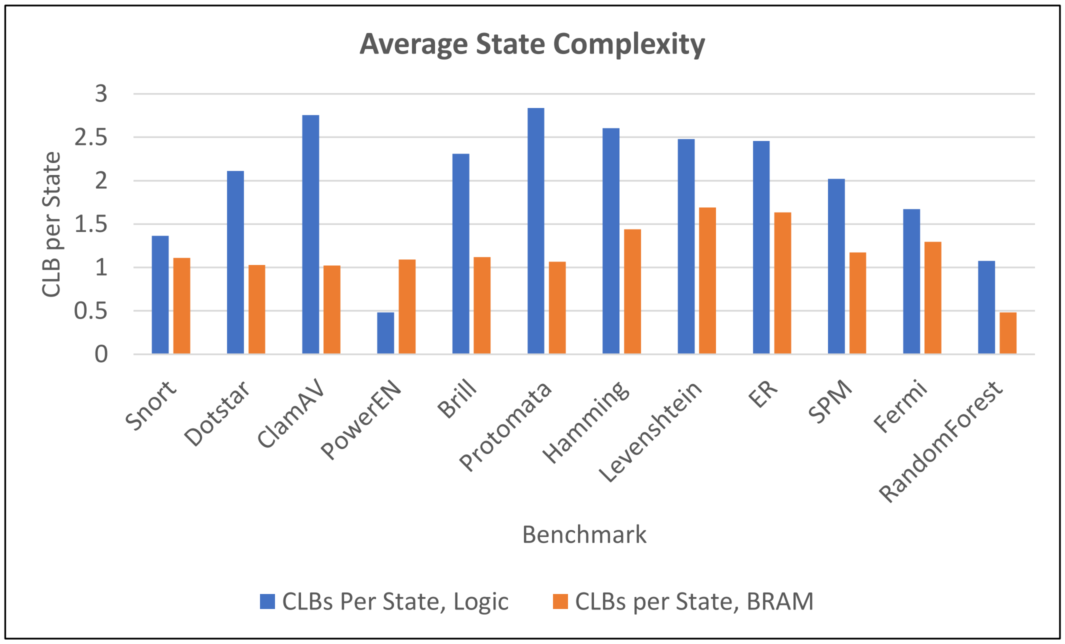

Figure 4.5: The average state complexity for an automaton is the ratio of CLBs placed and routed versus the number of automaton states. Automata with very simple character classes intuitively need less logic (CLBs) than more complex automata.

Figure 4.5 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 4.6. 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 4.7) 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

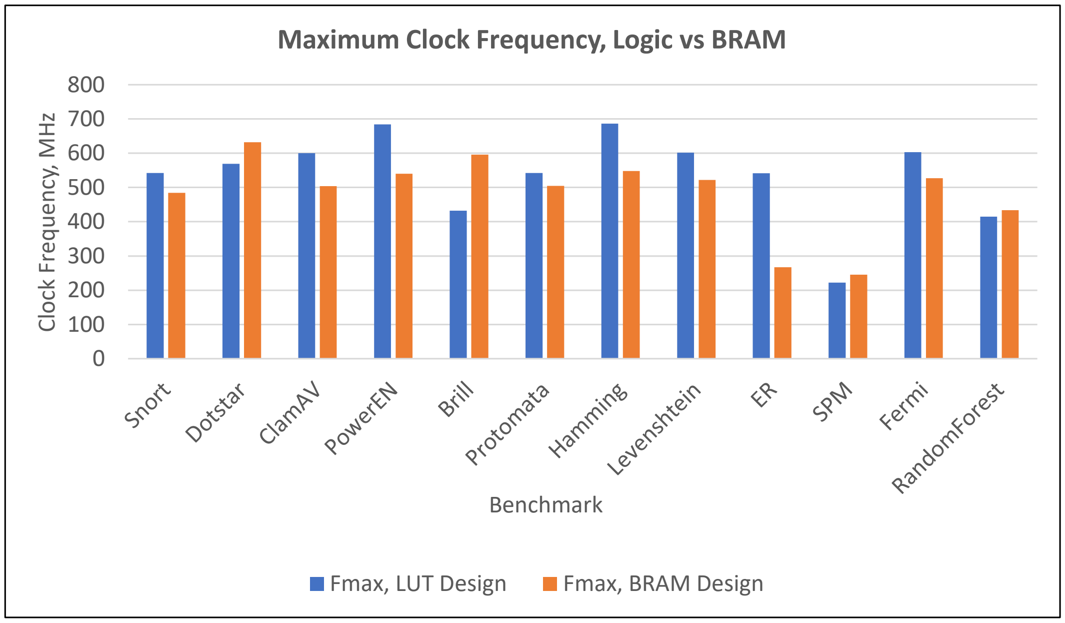

Figure 4.6: Clock frequency in most cases is degraded when comparing LUT-based automata to BRAM-based, and in general ranges from 200 MHz to nearly 700 MHz.

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 4.8 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 4.9) against an Intel Core i7 5820k CPU running Intel HyperScan [1]. 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.

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

[1]The HyperScan-based CPU engine and benchmark results were designed and collected by my colleague Jack Wadden.

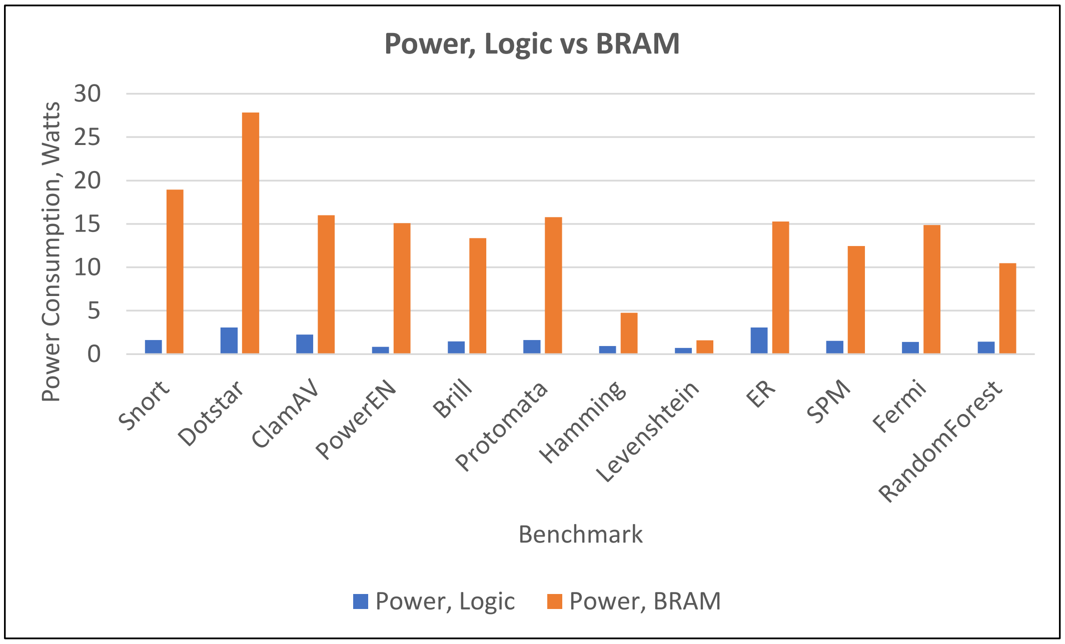

Figure 4.7: Estimated power consumption for LUT NFAs is very low; the most power hungry application, Dotstar, is estimated to consume barely more than 3 W. The BRAM implementation is much more inefficient, peaking at above 25 W for three applications.

4.4.3 Maximally-Sized Levenshtein Automaton

To demonstrate the true power of FPGAs for automata processing, we have developed a new "standard candle" [2] for the Levenshtein benchmark using the approach described by Tracy et al. [28]. 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.

4.5 Discussion

4.5.1 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 5.8x speedup over the estimated worst-case performance of 41.2 MBps.

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

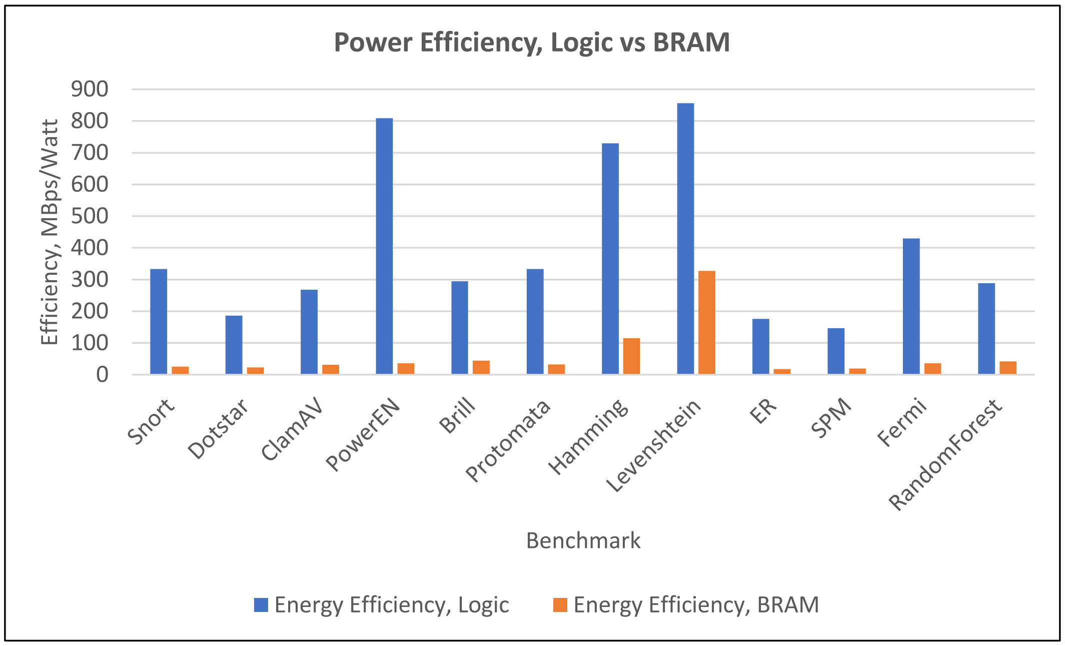

Figure 4.8: Estimated power efficiency of the LUT-based design greatly outstrips that of BRAM. Mesh benchmarks (Hamming and Levenshtein) perform very well in this aspect.

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.

4.5.2 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.

4.5.3 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

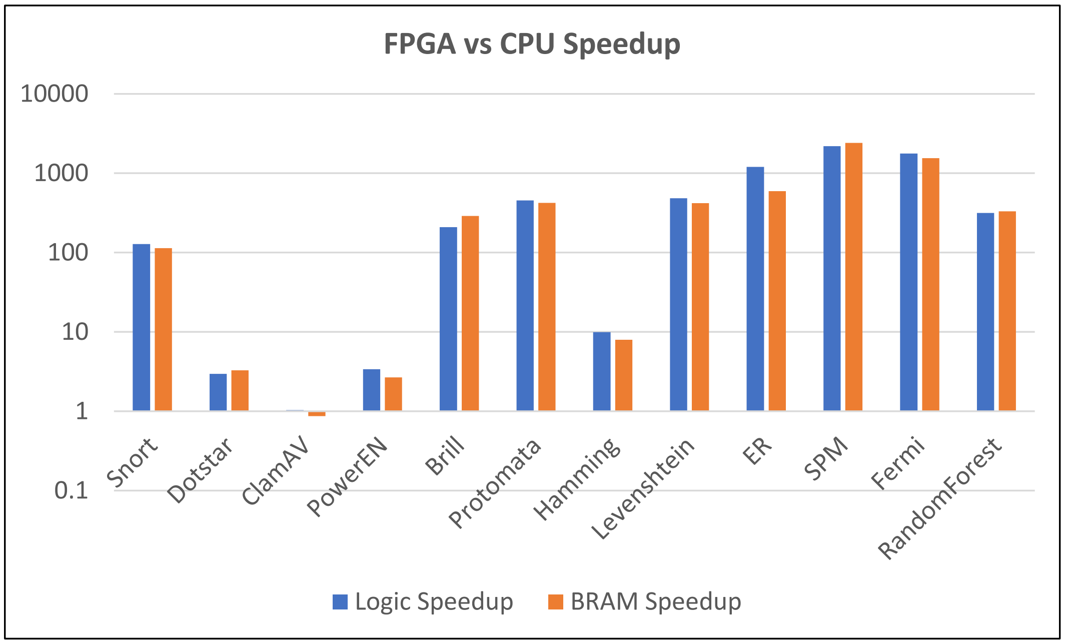

Figure 4.9: Speedup ranges from 1.03x to 2,188x for LUT-based automata and 0.87x - 2,009x for BRAM-based automata.

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 4.6 and 4.7.

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.

4.5.4 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.

4.5.5 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 [29]. 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 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.

4.5.6 Normalizing for Process Node

The AP is designed in 50 nm DRAM while our Kintex UltraScale FPGA is based on a 20nm 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.

4.6 Conclusion

In this chapter 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 and 5.8x speedup over the estimated worst-case performance of a naive reporting protocol. 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.

Chapter 5

Automata Processing on Intel HARP

In many graph analytics domains such as social networks and computer networks, there is a notion of a "power law," where a majority of the activity is concentrated in a small number of nodes. This chapter of the thesis will show that the same power law exists in automata graphs, and that this activity imbalance can actually be used to reduce resource utilization and compilation time (thus increasing productivity) for closely-coupled FPGA accelerator systems, where the FPGA computes the so-called "hot set" of an automaton and the CPU co-processes the "cold set." This chapter will additionally propose a novel reporting architecture for automata workloads and will benchmark its performance in the context of this hybrid spatial/von Neumann automata processing workflow.

The work presented in this chapter was the result of significant collaboration between myself and Jack Wadden, who designed the partitioning algorithm and reporting architecture. I implemented the RTL design of the system architecture and gathered place and route benchmark results as well as collaborated on the performance model described later in this chapter.

5.1 The Case for Multi-Platform Automata Processing

In the previous chapter, REAPR was shown to significantly improve processing throughput for a wide variety of automata processing workloads while occupying up to 70% of the Xilinx Kintex UltraScale FPGA's available configurable logic blocks. While many of the ANMLZoo benchmarks fit in a relatively small portion of the chip, the fact that some benchmarks like Entity Resolution approach the chip's capacity means that users will be fairly restricted by problem size with no real way of resolving this limitation. One possible way of side-stepping this concern is to exploit the aforementioned power law. Intuitively, it is possible to find a

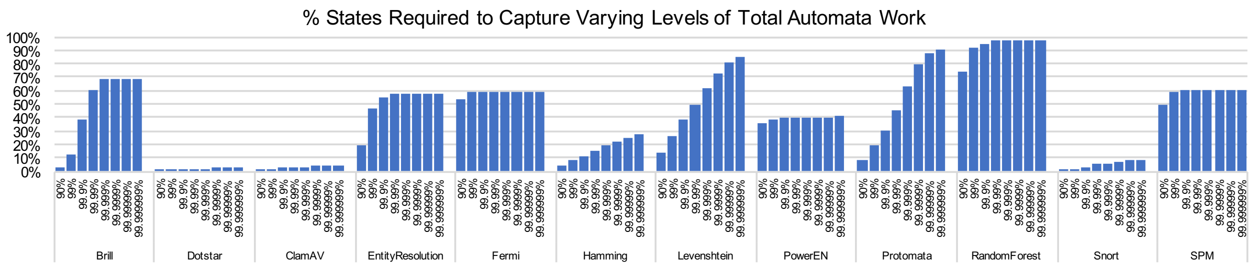

Figure 5.1: Percentage of states required to capture different levels of total work done by the automata.

graph cut in the automaton where one side has high activity and the other side has low activity. These two halves will be called the “hot set” and “cold set” respectively.

During run time, most of the activity will be contained within the FPGA’s hot set, while the CPU’s cold set will mostly sit idle. When an activation signal on the FPGA side must traverse the boundary between the hot and cold sets, that signal must be sent over a communication bus between the FPGA and CPU, thus incurring some I/O protocol overhead. As long as there are no backward-facing automaton state connections from the CPU back to the FPGA, the FPGA should never have to stall while waiting for the CPU to finish running its cold set computation. In this configuration, the FPGA and the CPU can both run at their peak processing rates.

The rest of this chapter will explore how to design an efficient reporting architecture for hybrid automata accelerators and various tradeoffs and metrics associated with the proposed partitioning technique.

5.2 Profiling Automata Activity

To find the delineation between the hot and cold sets, we have modified VASim to keep track of how many cycles each automata state has been activated. Because ANMLZoo does not contain a large number of input data sets, we must repeatedly partition the given testing data with an 80/20 split following the Pareto principle. By repeating this procedure, we can identify which states encapsulate various levels of behavior; for this experiment, we have chosen to find graph cuts that clearly delineate between one and eight 9’s of total activity (90% to 99.999999%). The results of this experiment can be seen in Figure 5.1, which plots the activity levels of each benchmark against the percentage of the original states required to capture that level.

The results of this experiment show that in some benchmarks such as Dotstar and ClamAV, a very small number of states is required to capture most of the activity; in the case of Dotstar, only 2.5% of the original number of automata states can capture 99.999999% of all activity! Conversely, in the Random Forest benchmark, a large percentage of the original states are necessary to capture most of the activity. This is due to the fact that the Random Forest kernel has many backward-facing loops, which causes the activity distribution to be more evenly spread out than in other strictly feed-forward automata.

When partitioning the automata workload between the accelerator and host, there will be a compile-time/throughput tradeoff associated with the partitioned activity level. When the CPU handles very high activity levels (i.e. there is minimal logic on the FPGA), compilation will be very fast, but the CPU might be overwhelmed by the amount of requisite processing and therefore bottleneck the entire system. Meanwhile, if the FPGA bears most of the computational burden, the CPU will have an almost negligible amount of work and process symbols very quickly, but the RTL compilation time will be much longer. Therefore, it is desirable to select an activity level that fairly balances the number of states on the accelerator and CPU, and therefore minimizes both compile time *and* CPU workload.

To do so, we again repeatedly perform a Pareto split on the input datasets and simulate the CPU's processing capability as a function of activity level using VASim. The results of this step are shown in Figure 5.3. To find the threshold between CPU bottlenecking and compilation time explosion, we simply choose an activity level for each benchmark that exceeds the FPGA's maximum processing potential. HARP's maximum system clock frequency is 400 MHz, and since REAPR processes one byte per cycle, the maximum throughput for any automata application on this platform will be 400 MBps. Intuitively, the threshold that maximizes both FPGA and CPU potential will be the lowest activity level that surpasses the 400 MBps peak; at this point, the CPU is guaranteed to never bottleneck the FPGA, and the FPGA takes on as much of the computational burden as it can without leaving the CPU with too light of a workload. For Fermi and PowerEN, it is not possible to partition the automata graphs in such a way that the CPU side ever processes above 400 MBps, so those benchmarks will not benefit from this partitioning technique. The chosen activity thresholds for the remaining ANMLZoo benchmarks are recorded in Table 5.1. The percentage reduction in number of automata states for the ANMLZoo benchmarks can be seen in Figure 5.2.

For every remaining ANMLZoo benchmark, we must find the activity level at which the CPU is guaranteed not to bottleneck the FPGA. This occurs when the CPU's throughput outstrips the HARP engine's maximum throughput, which at HARP's maximum frequency of 400 MHz with one byte processed per cycle means 400 MBps. We measure the CPU performance for each benchmark at each of the activity levels and measure the throughput achieved by VASim. The results of the activity profiling experiment are shown in Table 5.1.

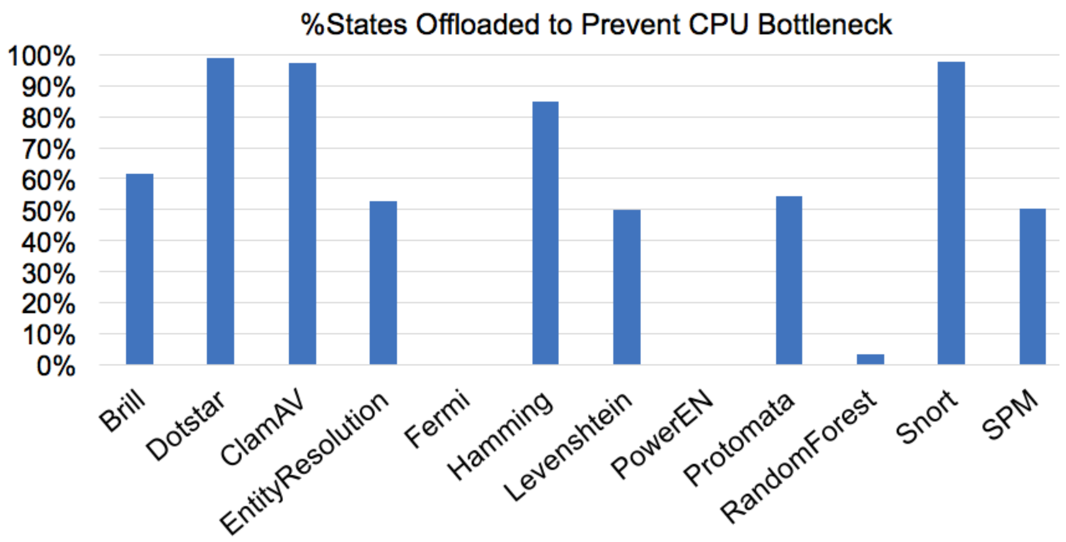

Figure 5.2: For most applications, the proposed partitioning technique can offload a huge number of automata states to the CPU. For Fermi and PowerEN, doing so is not possible, so there are no results for those benchmarks.

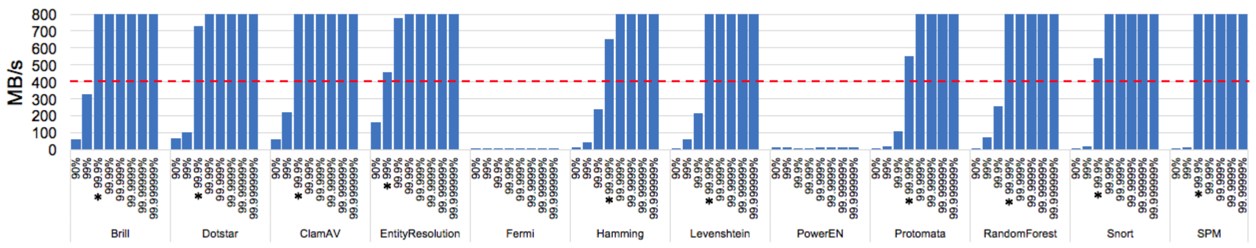

Figure 5.3: CPU performance for offloaded computation versus activity level. The dotted line indicates the maximum possible processing throughput for REAPR on the HARP platform - 400 MBps.

5.3 Hybrid System Architecture

5.3.1 Automata Processing Engine

To actual process the incoming symbol stream, we use REAPR's LUT-based design to generate automata RTL code for the optimized baseline designs as well as the selected activity levels. For I/O, we use Intel's Open Programmable Accelerator Engine (OPAE) [30] IP cores to facilitate data transfer with the Xeon CPU's last level cache. OPAE provides both FPGA building blocks for interacting with the CCI-P bus and a host-side C++ API that interact with OPAE-enabled accelerators. Using the virtual addressing mode provided by OPAE trivializes host/accelerator memory transaction logic by abstracting away the complexities of the QPI and PCI-Express channels that interface the HARP FPGA and CPU.

Benchmark Name	Chosen Partition
brill	99.9%
clamav	99.9%
dotstar	99.9%
er	99%
hamming	99.99%
lev	99.99%
protomata	99.99%
rf	99.99%
snort	99.9%

Table 5.1: Relevant ANMLZoo benchmarks with their selected activity-optimal partitions. These are the activity levels at which the CPU cannot possible bottleneck FPGA throughput, i.e. the CPU runs at more than 400 MBps.

5.3.2 Reporting Architecture

One major limitation of the Xilinx SDAccel implementation of REAPR is the lack of a real reporting architecture. In the REAPR chapter, the only application that had actual end-to-end performance numbers on real hardware was Random Forest, which was only made possible by creating a highly customized match offloading circuit that reduced the number of reporting signals from 1,661 to 8. This approach is not generalizable to other automata applications and also tied to Xilinx's particular flavor of high-level synthesis C, and therefore is not even portable to other FPGA platforms.

This chapter of the thesis presents the RTL implementation of a high performance general-purpose reporting architecture initially proposed by Wadden et al [31]. This circuit is composed of several components: the automata report vector, report aggregators ("RAGGs"), an arbiter, and a 128-bit shift register.

The standard RTL interface for automata designs is shown in Figure 5.4. This is the format generated by REAPR and is comprised of five main signals. The first input signal is the 8-bit **data input** signal, which is broadcasted to all automata states. The next three, enable, reset, and clock, are standard digital circuit control signals, while the last N-bit **Reports** signal is the buffer that stores all report events for a given cycle in the automaton. Because the number of reports as well as the locations of reports inside the output vector are both unpredictable, it is especially difficult to apply traditional signal compression techniques on this output data, which ranges on the order of kilobits per cycle.

Figure 5.5 shows the overall topology for the reporting architecture for a relatively small automaton with 128 bits in its output buffer. The bottom 64 bits of reports are handled by RAGG 0 while the top 64 are handled by RAGG 1. For other automata, the report vector will be split into 64-bit chunks, each of which is assigned its own RAGG.

At the top of the diagram, the automata processing module, represented as the blue cloud, processes the

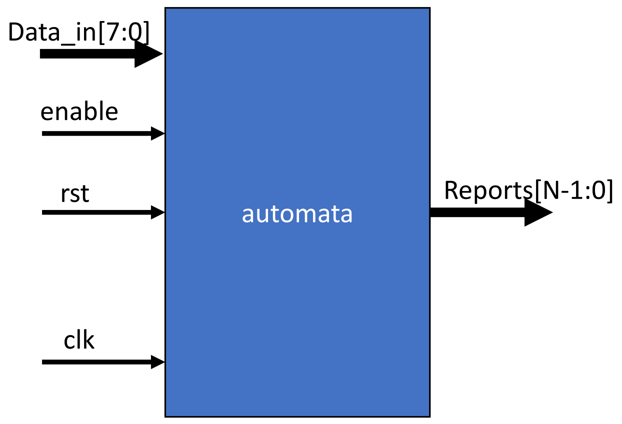

Figure 5.4: The standard RTL interface for finite automata has an 8-bit input and an N-bit output, where N is the number of reporting elements.

input stream one byte at a time until at least one bit in the report vector is high. At this time, one or both of the RAGGs, depending on the location of the report signal(s), will send a request to the arbiter, and the arbiter will send a stall signal to the automata module until all RAGG requests are de-asserted.

RAGGs will assert their request line if at least one of the report signals in their 64-bit `Reports_in` input is high. This request will stay high until the arbiter grants the request, at which point the request line will be forced low. A circuit diagram of the RAGG logic can be seen in Figure 5.6.

The arbiter will issue one RAGG grant at a time until no more RAGGs request their data to be offloaded (i.e. all of the requests have been serviced). When a RAGG's request is granted by the arbiter, its 64-bit data chunk is passed through the report multiplexer and appended with 64 bits of metadata containing the timestamp and the RAGG ID. The metadata field is used by the CPU-side code to determine which automaton states were activated and where in the input stream they were activated.

Each time a valid 128-bit report data + metadata "report packet" is created, it is sent to a 4x128-bit shift register that serves as a buffer to the cache coherent interconnect bus. Once this shift register has been filled with four entries, the HARP system will send the entire 512-bit buffer through CCI-P to be stored back into the CPU's memory system.

5.3.3 HARP Integration

Two finite state machines handle streaming input and output; the input stream feeds into the automaton's 8-bit data input while the output stream accepts 512-bit packets from the reporting architecture. The input FSM requests one 64B cacheline at a time from the CCI-P bus and stores the responses into a FIFO. The input controller will continue requesting cachelines until either the FIFO is full or the end of the data stream has been reached.

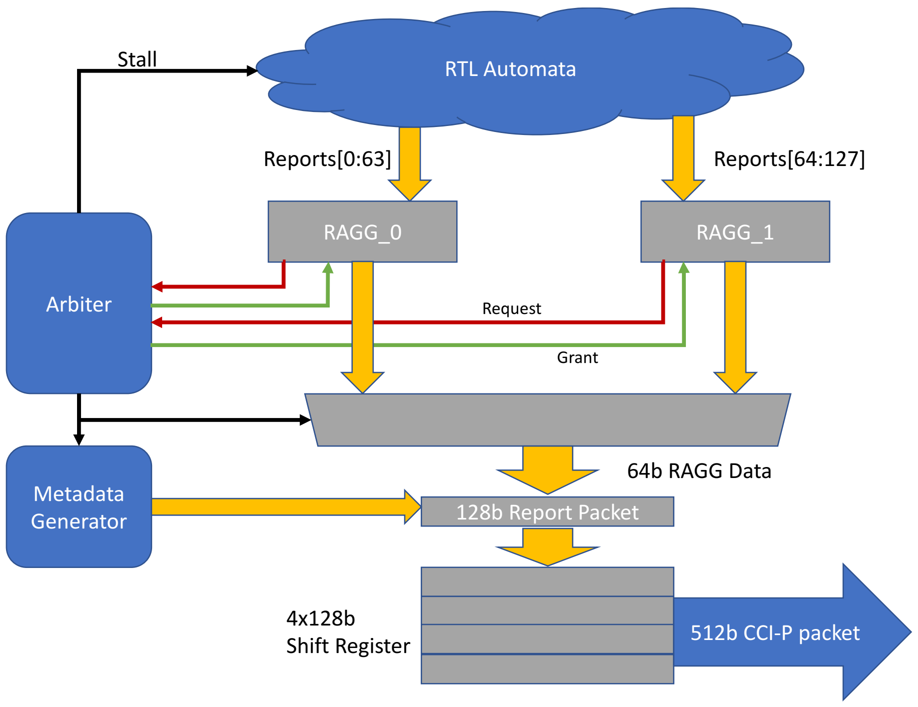

Figure 5.5: System topology for the reporting architecture.

Meanwhile, the automata processing engine reads one byte at a time from the FIFO, only pausing when the reporting architecture's arbiter sends a stall signal. On the output side, another finite state machine continuously checks the 4x128b shift register to see whether it is full.

Once all four entries in the shift register are populated, the FSM will export the contents to the CCI-P bus and empty the shift register. Since there are three channels available to HARP's CCI-P controller (1 QPI and 2 PCI-Express) that can be automatically selected based on availability, it is possible to simultaneously request a line read while also writing the 512b output buffer. Figure 5.7 shows the overall system architecture inside of the FPGA including all of the automata processing, reporting architecture, and I/O elements.

Both the baseline and partitioned ANMLZoo benchmarks described in Table 5.1 will be integrated into a full HARP system and run through a full place and route flow. The RTL compilation results of each benchmark will provide two data points each: 1) the end-to-end compilation time for a full HARP automata accelerator system; and 2) the resource utilization. To run the place and route flow, we use Intel Quartus Prime Pro v16.0 on a CentOS 7 Linux distribution. Our compilation machines have 6-core Intel Core i7 5820k with 32 GB of 2133 MHz DDR4 RAM.

To estimate the performance of this system, a cycle-accurate performance model has been developed which can read in report traces generated by VASim and output an estimate of the total number of cycles needed to process a given input file. Since the reporting architecture adds stall cycles every time at least one

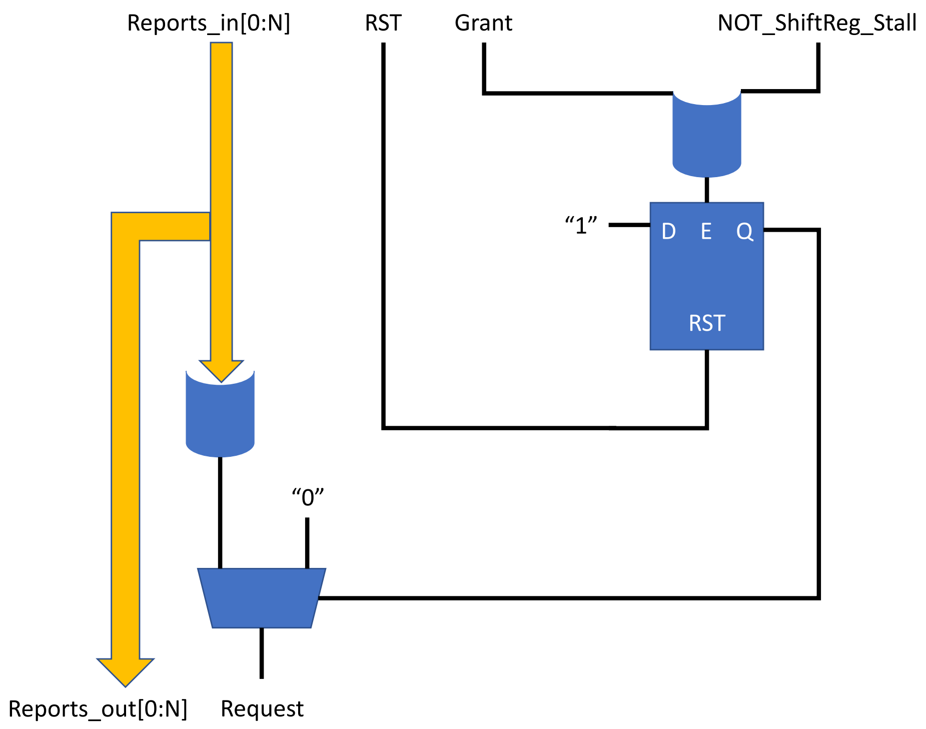

Figure 5.6: Microarchitecture of the report aggregator (RAGG).

RAGG's request line is high, it is important to quantify this overhead.

5.4 Results

5.4.1 Spatial Resource Reduction

Figure 5.8 shows the total number of LUTs used for each of the ANMLZoo benchmarks for the optimized kernels (blue) versus the partitioned ones (orange). In many cases, the *total* number of LUTs is not drastically reduced (on average just 8%, with a maximum of 20%), but this is mainly due to the fact that the automata in ANMLZoo are relatively small compared to the high overhead associated with the various I/O circuitry needed for HARP (just I/O circuitry occupies roughly 18% of available LUT resources).

Examining the total number of *automata states* before and after partitioning gives a clearer picture of the true benefit of this technique. Figure 5.2 shows the total number of automata states that can be offloaded to the CPU using the proposed partitioning technique. In most cases, a large fraction of the states can be pruned from the FPGA-side automata engine.

A similar analysis can be applied to FPGA resource utilization by examining only the *automata-related* LUT usage, which can be easily done by simply subtracting the number of LUTs needed for a bare-bones I/O kernel from the total number of LUTs needed for each benchmark. Figure 5.9 shows the results of this

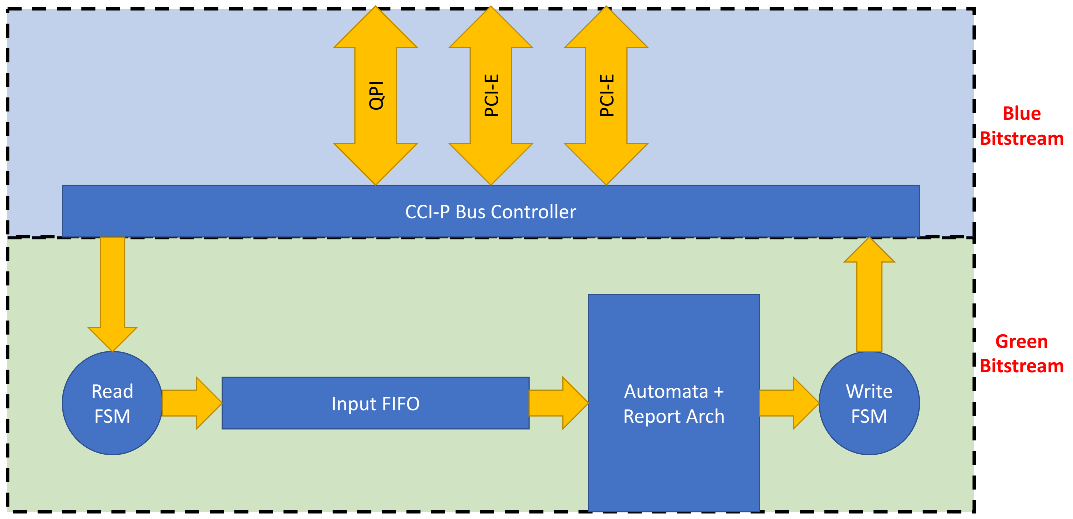

Figure 5.7: System design for the automata accelerator and reporting architecture inside of the HARP platform.

comparison. On average, the partitioning approach results in **over 60% reduction** in LUTs required to implement nearly the same functionality as the baseline benchmark application. In the case of Dotstar, over 98% of LUT resources can be removed with this technique. Comparing Figure 5.9 with Figure 5.2, the two plots nearly mirror each other except in the case of Brill. For that particular benchmark, the likely explanation for the mismatch between the high LUT reduction rate and the relatively low state reduction rate is that the states left over by the partitioning algorithm have redundant behavior, so the FPGA logic optimizer chooses to remove the circuitry that implements those states.

5.4.2 Compilation Time Reduction

A direct consequence of reducing requisite spatial resources for these automata benchmarks is *reduced compilation time*. This can be readily observed in Figures 5.10 and 5.11, which represent the total compilation time (baseline versus partitioned) and the percentage improvement due to partitioning, respectively.

On average, the compilation time is reduced by **48%**, a nearly 2x improvement in productivity! Interestingly, Random Forest, a benchmark that did not benefit very much from partitioning in terms of states offloaded or LUT reduction, sees a 55% decrease in compile time, which indicates that even if partitioning cannot necessarily improve the resource utilization of an application, it can at least reduce the compilation time.

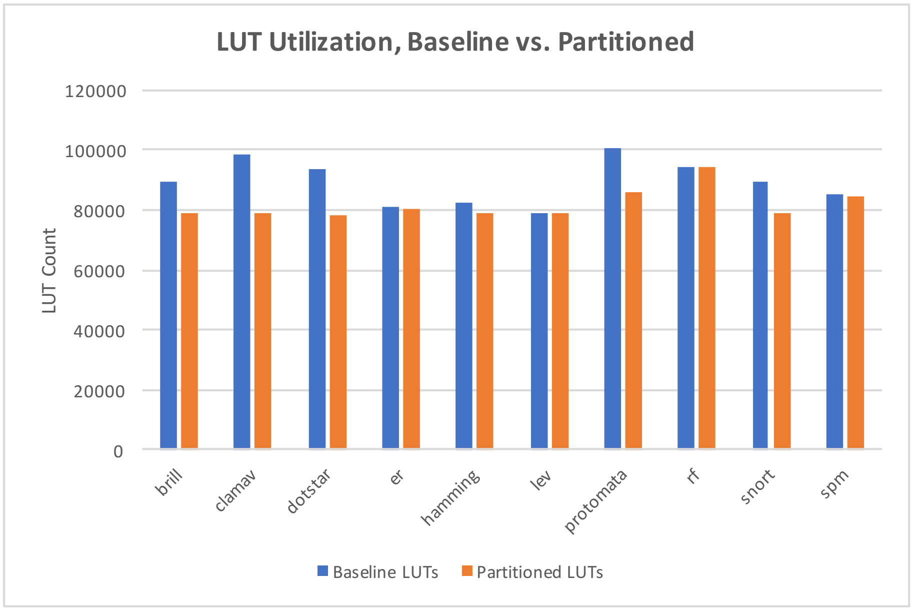

Figure 5.8: The total number of LUT resources needed to compile each ANMLZoo benchmark. The blue bars represent the LUT count for the baseline optimized automata graphs while the orange bars represent the LUT count for the partitioned graphs. The reduction in LUTs necessary ranges from 0.3% (Random Forest) to 20% (ClamAV) with an average of 8.15%.

5.4.3 Communication/Synchronization Overheads

Figure 5.12 shows the communication overhead associated with the baseline automata compared to the partitioned ones. For most applications, partitioning does not add a significant overhead compared to the baseline - only as high as 6% in the case of Brill. Some benchmarks can actually *benefit* from the partition as groups of states with high report activity are off-loaded to the CPU, thus saving precious cycles that the reporting architecture does not need to stall for. This phenomenon occurs in Entity Resolution but is most pronounced in SPM, where the partitioned automaton runs over 20% faster than the baseline. Future work may explore partitioning strategies that minimize only *reporting behavior* rather than activation behavior, as this chapter's approach does.

5.5 Discussion

For most applications, the partitioning technique described in this chapter is able to reduce the number of automata states implemented on the FPGA significantly, in some cases up to 97%. The net effect of this optimization is that the FPGA is now relatively underutilized - peak LUT resource utilization for the partitioned applications barely reaches above 20% of the FPGA's total system resources, including area overhead attributed to I/O and reporting architecture circuitry. Theoretically, users could accelerate much

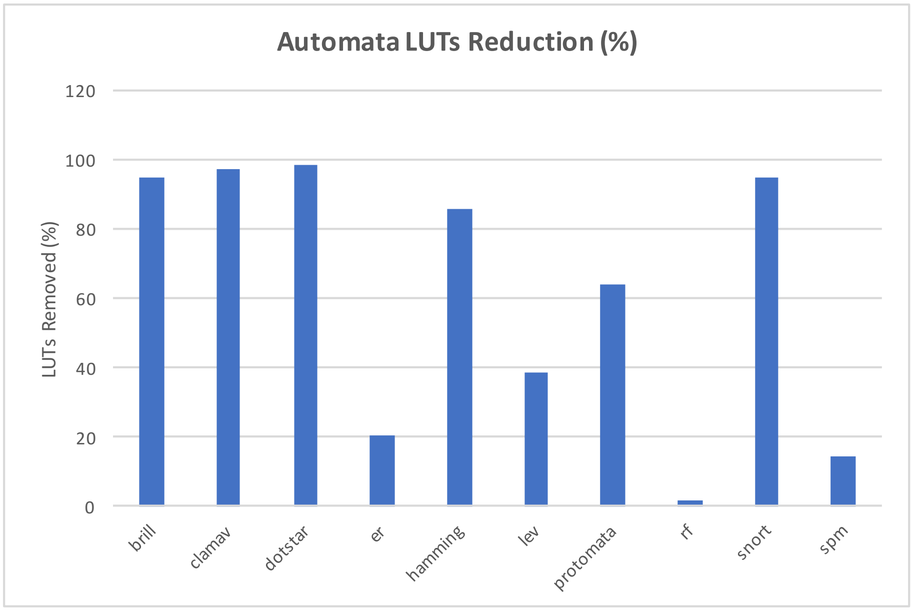

Figure 5.9: The percentage of strictly automata-related LUTs removed. This figure of merit is simply the total LUT count for a given benchmark minus the number of LUTs needed for a basic I/O kernel.

larger automata applications (compared to what can typically fit on just one FPGA) thanks to the work proposed in this thesis. This partitioning technique effectively combines the strengths of CPU and FPGA automata processing: large automaton size and high computational throughput, respectively.

A secondary effect of reduced spatial resource utilization is reduced compilation times. Compiling automata accelerators for an FPGA platform can be extremely time intensive due to the aggressive logic optimizations performed by FPGA place and route tools. For most ANMLZoo applications, compiling an accelerator to implement a whole benchmark can take up to four hours! By reducing the amount of logic required to implement automata kernels, it is possible reduce the overall compilation time by an average of **48%**, effectively allowing developers to iterate twice as fast while prototyping their systems. This is also a boon for any compilation time-sensitive applications such as sequential pattern mining [7] or network intrusion detection systems [3] [4], since the performance and/or efficacy of these applications is directly tied to how quickly an existing automaton design can be updated.

5.6 Conclusion

In this chapter, a novel hybrid automata *co*-processing architecture was explored on the Intel Xeon+FPGA (“HARP”) platform. In order to take advantage of the tight coupling between HARP’s CPU and FPGA, this work proposed running the most computationally intensive part of an automaton, or the “hot set”, on a high-performance Intel Arria 10 FPGA while the CPU computes the “cold set.” This approach has several

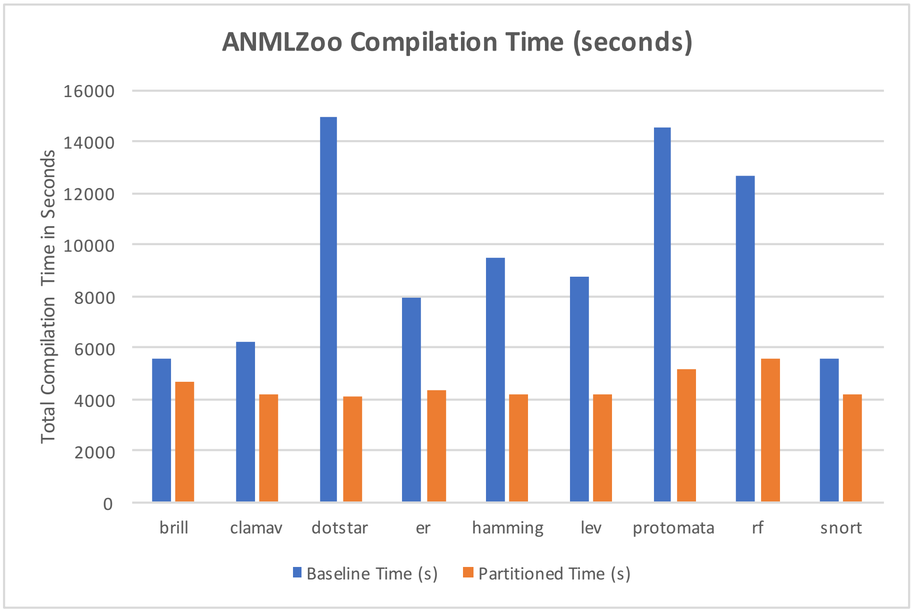

Figure 5.10: The total number of seconds needed to compile the optimized baseline graphs (blue) versus partitioned graphs (orange).

key benefits. First, it greatly reduces the number of states run on the FPGA (up to 98% are removed), thus also greatly reducing the LUT resource utilization. Second, because there are now fewer circuit elements to synthesize and place & route, the total end-to-end compilation time is greatly reduced (on average 48% and up to 72%). Lastly, some partitions (see Entity Resolution and Sequential Pattern Mining) actually offload frequently reporting states to the CPU, where reporting overhead is not as significant compared to FPGA, and therefore can reduce the communication overhead associated with reporting.

These experiments show that automata partitioning has a wide variety of benefits, and that Intel HARP is an excellent platform to host this type of co-processor architecture given the tight coupling between the host CPU and the accelerator. The experimental results from this chapter indicate that this technique can reduce circuit area (thus increasing maximum automaton capacity for the HARP FPGA), reduce compilation time, and even improve the overall run time compared to a system which only uses the FPGA for automata processing.

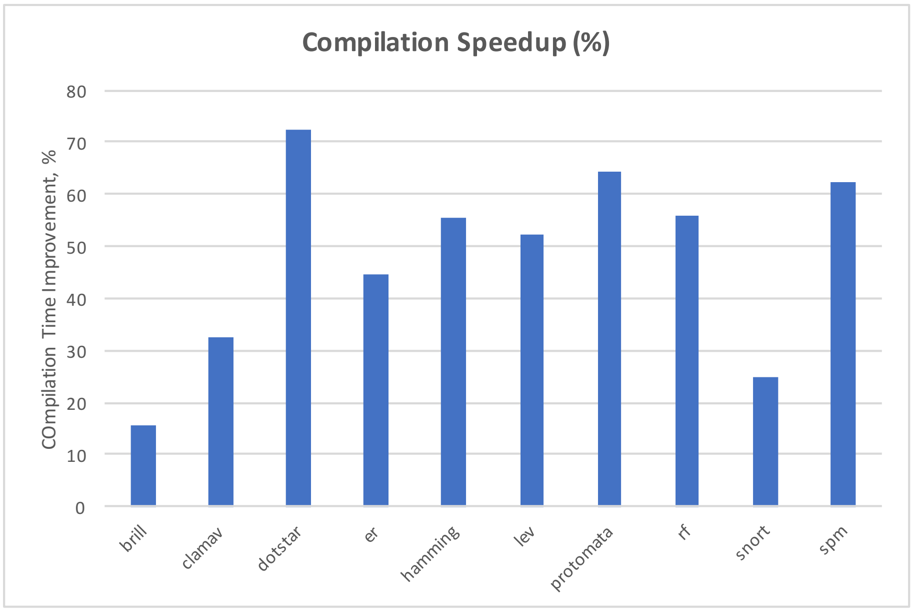

Figure 5.11: The amount (in percentage) that automata graph partitioning can reduce total compilation time. This ranges from 15.6% (Brill) to 72% (Dotstar).

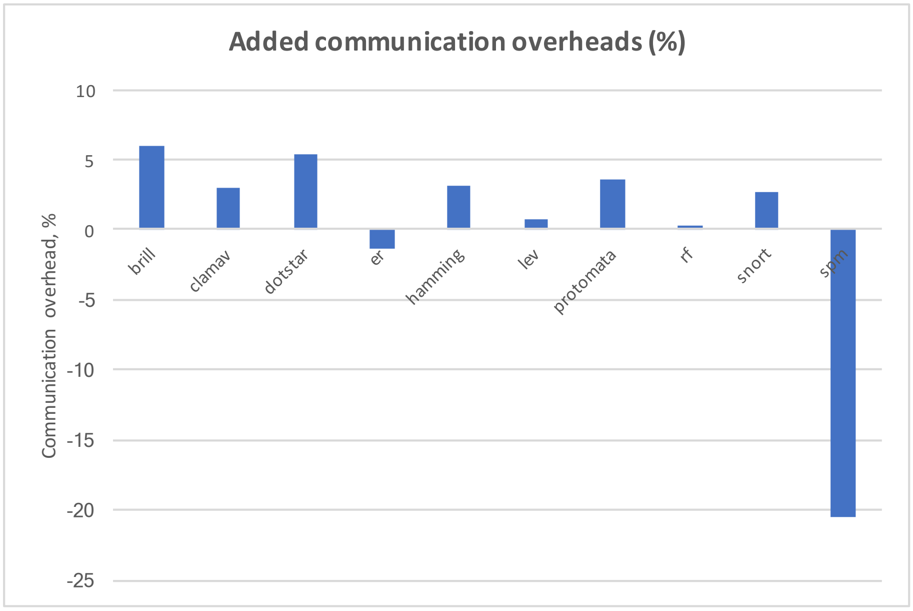

Figure 5.12: In most cases, the partitioned graphs will not require significantly more cycles to run than the baseline automata. In some cases such as Entity Resolution and SPM, the runtime may actually be reduced because some frequently reporting states have been allocated to the host CPU.

Chapter 6

Conclusion

This thesis has shown that FPGAs are an ideal platform to host automata processing applications. By exploiting the straightforward translation from the spatial distribution of automata state machine nodes to digital circuit elements, it is possible to transform the $O(n * m)$ work associated with simulating non-deterministic finite automata (NFAs) into just $O(n)$ runtime while still only using $O(m)$ circuit elements. FPGAs enable greater time- and space- efficiency for NFA processing than traditional von Neumann CPU architectures.

The fourth chapter of this thesis proposed REAPR, a tool that generates RTL code and rudimentary I/O architecture for an FPGA backend. Synthesizing the ANMLZoo benchmark suite [26] using REAPR for the Xilinx SDAccel platform confirmed the initial hypothesis that FPGAs are well-suited for automata processing; the spatial representation of nondeterministic finite automata naturally maps to the spatial representation of circuit elements in reconfigurable hardware. Thanks to this compatability, FPGAs can offer significant performance and efficiency boosts for automata processing workloads compared to traditional von Neumann computer architectures. Additionally, experiments in the fourth chapter confirmed the hypothesis that synthesizing a 2D-mesh application (Levenshtein distance) on a fabric with a 2D-mesh routing topology (FPGAs) will have superior routing and capacity performance than doing so on the Micron Automata Processor, which has a tree-like routing topology. 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 and 5.8x speedup over the estimated worst-case performance of a naive reporting protocol. 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.

The fifth chapter of this thesis optimized the resource utilization and compile time associated with the original REAPR system by sharing the NFA processing workload between the Xeon host CPU and a tightly-coupled cache-coherent FPGA on the Intel HARP platform. Using this workload sharing approach, the overall resource utilization was lowered by **98%** while the compile time was reduced by an average of **78%**. These results indicate that for automata processing, it is not entirely necessary to have a large state-of-the-art FPGA to still preserve high performance for automata processing. Future computer systems purpose-built for automata processing may choose to include a small *integrated FPGA* (much like an integrated GPU on desktop/laptop processors) to process the ”hot set” of an automata kernel, thus obviating the need for a large top-of-the-line FPGA such as an Arria 10.

Chapter 7

Future Work

7.1 FPGA Automata Processing Overlay Architecture

One of the major limitations to both of the works proposed in this thesis (REAPR on SDAccel, REAPR on HARP) is the compilation time. Without any kind of optimizations, accelerator systems generated by REAPR for Xilinx SDAccel take on the order of 5-15 hours to compile. Even after drastically pruning the number of states using the profiling method described in the HARP chapter, compile times are *still* around one or two hours. This leads to greatly reduced productivity and is especially detrimental for any iterative or turnaround-sensitive application. Iterative algorithms such as sequential pattern mining rely on using the outputs from the previous run to reconfigure the behavior of the automaton for the upcoming run. With the current way that REAPR is designed, every iteration would incur a several-hour penalty to completely rerun the RTL compilation flow, which is unacceptable (for reference, CPUs can reconfigure automaton behavior almost instantaneously - just change the contents of some pointers). For other turnaround-sensitive applications such as network intrusion detection systems or antivirus file scanning, the time difference between discovering the patch for a virus and deploying the patch is critical. A latency of several hours is simply unacceptable for this type of scenario; CPU NIDS engines can almost instantaneously add a new regular expression to their database and be up-and-running in mere microseconds.

To address these concerns, it is prudent to develop a layer of abstraction on top of the existing FPGA infrastructure that has a smaller and more coarse-grained design space than an FPGA that allows a spatial reconfigurable automaton kernel to be compiled in seconds or minutes rather than hours. To this end, there is already existing work that has built the "automata-to-routing (ATR)" tool [32] that can place-and-route automata designs onto a parameterizable spatial reconfigurable fabric relatively quickly. Using the tools and intuitions from ATR, it will be relatively straightforward to implement the RTL for an automata overlay architecture's switch boxes, connection boxes, logic cells, etc. and integrate it with a platform such as Xilinx SDAccel or Intel HARP. The main challenges to this overlay approach will be timing and capacity tradeoffs; implementing FPGA-like routing on top of existing FPGA routing will surely incur huge area overhead.

7.2 Integrated FPGAs for Consumer-Grade High-End Desktop (HEDT) Systems

The results from the HARP chapter of this thesis indicate that at least for automata processing, it is possible to capture most of the circuit activity in just a small percentage of the area. The remainder of the circuit, or the "cold set", can be computed on the CPU when an activation signal propagates from the FPGA to the host CPU. While this approach may not apply to *all* accelerator systems, it is certainly possible that many hardware accelerators do not necessarily need the full might of an Arria 10 FPGA to achieve high throughput. A recent paper using the Rodinia GPU benchmark suite as an FPGA high-level synthesis benchmark suite [33] indicates that several of the Rodinia applications (NW, HotSpot3D, pathfinder) fit in roughly 20% of the logic resources on an Intel Stratix V FPGA, the largest Intel/Altera FPGA of the last generation. Since the Cyclone V, Intel's smallest family of FPGAs, contains between 10% and 50% of the logic capacity of the Stratix V, it seems plausible that a small FPGA can perhaps even reside in the same die as a consumer-grade Core i5 or i7 CPU, much like how existing Intel Core processors already have integrated GPUs for accelerating graphics workloads. Incorporating small high-performance FPGAs into desktop processors could potentially improve the throughput of broad range of applications, not just graphics, for a small chip area and power consumption overhead. Similar to HARP, these integrated reconfigurable processors could share a coherence domain with the main CPU's last level cache and be programmed with a software engineer-friendly language like OpenCL.

Bibliography

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

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

[3] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the USENIX Large Installation Systems Administration Conference (LISA)*, 1999.

[4] Vern Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[5] ClamAV. ClamAV Rules. Available at https://www.clamav.net/.

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

[7] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Sequential Pattern Mining with the Micron Automata Processor. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2016.

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

[9] Michael H.L.S. Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. Using the Automata Processor for fast pattern recognition in high energy physics experiments—a proof of concept. *Nuclear Instruments and Methods in Physics Research*, 2016.

[10] Reetinder Sidhu and Viktor K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact Architecture for High-throughput Regular Expression Matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 30–39, New York, NY, USA, 2008. ACM.

[12] 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*, 25(12):3088–3098, Dec 2014.

[13] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea R. Stan. Reapr: Reconfigurable engine for automata processing. *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2017.

[14] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology, 2006.

[15] Pascal Caron and Djelloul Ziadi. Characterization of glushkov automata. *Theoretical Computer Science*, 233(1):75 – 90, 2000.

[16] Russell Tessier. *Fast Place and Route Approaches for FPGAs*. PhD thesis, Massachussetts Institute of Technology, 1999.

[17] Xilinx. Sdaccel development environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[18] Algo-Logic. Intel xeon + fpga. http://algo-logic.com/Intel-Xeon-FPGA.

[19] Intel. Intel cci: Core cache interface. https://01.org/sites/default/files/downloads/opae/cci-p-mpf-overview.pdf.

[20] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, pages 1:1–1:12, New York, NY, USA, 2007. ACM.

[21] Tran Trung Hieu and N. T. Tran. A memory efficient fpga-based pattern matching engine for stateful nids. In *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 252–257, July 2013.

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

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

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

[25] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. iNFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[26] Jack Wadden, Vinh Dang, Nathan Brunelle, Tom Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[27] Jack Wadden. Virtual Automata Simulator (VASim). Available at https://github.com/jackwadden/vasim/.

[28] Tommy Tracy, Mircea R. Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabriel Robins. Nondeterministic finite automata in hardware - the case of the levenshtein automaton. 2015.

[29] Micron Technologies. Micron Automata Pocessor: Developer Portal. Available at http://micronautomata.com/.

[30] Intel. Open programmable acceleration engine. https://opae.github.io/.

[31] Jack Wadden, Kevin Angstadt, and Kevin Skadron. Characterizing and mitigating output reporting bottlenecks in spatial-reconfigurable automata processing architectures. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2018.

[32] Jack Wadden, Samira Khan, and Kevin Skadron. Automata-to-Routing: An Open Source Toolchain for Design-Space Exploration of Spatial Automata Processing Architectures. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[33] A. Podobas, H. R. Zohouri, N. Maruyama, and S. Matsuoka. Evaluating high-level design strategies on fpgas for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2017.