CSE PhD: Improving Programming Support for Hardware Accelerators Through Automata Processing Abstractions

We propose designing programming models and maintenance tools for hardware accelerators using finite automata as an intermediate representation. We focus on maintaining *performance* and *scalability* while improving *ease of use*, *expressive power*, and *legacy support* with our tools.

Background and Motivation. Hardware accelerators are becoming more commonplace for accelerating general-purpose computation due to increasing rates of data collection and increased pressure for real-time analyses. While the performance of these devices show great promise, support for programming and maintenance tasks tend to be low-level or nonexistent, which places a significant burden on developers and slows the adoption of this technology. Recent efforts to accelerate applications with finite automata (DFAs and NFAs) on hardware accelerators have been quite successful, and we propose leveraging this work as an intermediate representation to design novel high-level programming models and maintenance tools.

Intellectual Merit. We propose four research efforts:

1. **High-Level Languages for Automata Processing:** We will design a high-level programming language, RAPID, for accelerating sequential pattern-matching applications. We propose extending a C- or Java-like language with three domain-specific parallel control structures and creating algorithms to compile the language to a set of finite automata.
2. **Interactive Debugging for High-Level Languages and Accelerators:** To aid developers with debugging-related maintenance tasks, we propose developing a high-throughput interactive debugger that captures the low-level execution context on hardware accelerators and bridges the semantic gap with high-level RAPID programs.
3. **Expressive Power of In-Memory Automata Accelerators:** Processing of tree-structured or recursively-nested data is intrinsic to many applications, but is not directly supported by finite automata. We will therefore design a new, in-cache accelerator architecture (and associated compiler) that supports the richer computational model of pushdown automata.
4. **Adapting Legacy Code for Execution on Hardware Accelerators:** We propose combining insights from automata learning and software model checking to design algorithms that learn functionally-equivalent automata from legacy code for execution with accelerators.

Evaluation. We will compare the *performance* and *scalability* of applications written in our proposed RAPID language with expert-crafted baselines for a collection of real-world benchmarks. We will measure the *ease of use* of our proposed debugging system by conducting an IRB-approved human study that measures accuracy and duration for indicative fault localization tasks. To evaluate our proposed pushdown automata architecture, we will consider both *expressive power* as well as *performance* using real-world case studies. Our evaluation of *legacy support* for our automata learning algorithms is the most speculative and will measure performance for a benchmark suite of legacy code as well as characterize approximation quality for learned automata.

Broader Impact. There is a growing need for effective programming models for hardware accelerators as devices become more prevalent in general-purpose computing; this work seeks ease the adoption of such devices through easy-to-use and performant tools. We also seek to train the next generation of researchers and engineers by actively pursuing mentorship opportunities for underrepresented undergraduates.

Improving Programming Support for Hardware Accelerators Through Automata Processing Abstractions

Ph.D. Dissertation Proposal
Kevin A. Angstadt
`angstadt@umich.edu`

December 4, 2018

1 Introduction

Writing software for hardware accelerators can be a difficult and error-prone process. However, the confluence of several factors, including the rapid growth of data collection [32], demands for real-time analyses by business leaders [29], and the lessening impact of Dennard Scaling and Moore's Law [76], have all led to the increased use of hardware accelerators, such as Field-Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), Google's Tensor Processing Unit (TPU) [52], Micron's D480 Automata Processor (AP), and others for general-purpose computing. Such accelerators trade off general computing capability for increased performance on very specific workloads; however, these devices require additional architectural knowledge to effectively program and configure.

While present in industry for prototyping and application-specific deployments for quite some time, reconfigurable architectures, such as FPGAs, are now becoming commonplace in everyday computing as well. In fact, FPGAs are in use in Microsoft datacenters and are also widely available through Amazon's cloud infrastructure [3, 25, 54, 67].

Current programming models are akin to assembly-level development on traditional CPU architectures. While these hardware solutions provide high throughputs, programming them can be challenging. Consequently, programs written for these accelerators are tedious to develop and challenging to write correctly. Additionally, these low-level representations do not lend themselves well to debugging and maintenance tasks. We hypothesize that this low level of abstraction places a high burden on developers and is a key barrier for the adoption of hardware accelerators. Higher levels of abstraction for programming FPGAs have been achieved with languages such as OpenCL [84] and frameworks such as Xilinx's SDAccel [101]; however, these models still require low-level knowledge of the underlying architecture to allow for efficient implementation and execution of applications [91, 109].

We argue that a successful programming model must satisfy the following criteria:

- **Performance and Scalability.** Maintaining the performance gains provided by hardware accelerators is critical and is achieved by minimizing the overhead introduced by high-level programming models and tools.
- **Ease of Use.** Tools must aid developers in effectively writing and maintaining software for hardware accelerators by providing familiar abstractions and a shallow learning curve.
- **Expressive Power.** The underlying computational model must be rich enough to support the applications that developers wish to accelerate with dedicated hardware.
- **Legacy Support.** Programming models must support the adaptation of existing software to execute efficiently on hardware accelerators while placing a minimal burden on developers.

We propose building programming models and software maintenance tools for hardware accelerators using finite automata as an intermediate representation. The overarching hypothesis of the proposed dissertation is:

> Finite automata provide a suitable abstraction for bridging the gap between high-level programming models and maintenance tools familiar to developers and the low-level representations that execute efficiently on hardware accelerators.

Our approach leverages the following insights. First, finite automata are a good fit for representing a diversity of applications. Recently, researchers have successfully developed new algorithms using the automata processing abstraction to accelerate analyses across many domains, including: natural language processing [108], network security [73], graph analytics [72], high-energy physics [99], bioinformatics [70, 71, 90], pseudo-random number generation and simulation [95], data-mining [97, 98], and machine learning [89]. Second, finite automata maintain compact state, which allows for program introspection tools to monitor a relatively small number of signals to capture the execution of a program. Third, finite automata can be mapped efficiently to reconfigurable architectures [31, 102], allowing for scalability and performance. Finally, we observe that support for the execution of existing software on hardware accelerators can make use of legacy source code as an oracle for counterexample-guided approaches [7, 27] to learning functionally-equivalent automata.

The expected main contributions of the proposed dissertation are the following:

1. A high level programming language, RAPID, for accelerating sequential pattern matching applications on hardware accelerators.
2. A high-throughput, interactive debugging system for RAPID programs for maintenance tasks on FPGAs and Micron's D480 Automata Processor.
3. An in-cache accelerator and associated optimizing compilation algorithms for execution of deterministic pushdown automata.
4. An automata synthesis system for porting legacy source code to execute on hardware accelerators by learning functionally-equivalent automata.

The remainder of this proposal is as follows. In Section 2, we discuss background material and related research efforts. Next, we describe our proposed research and associated technical approaches in Section 3, and in Section 4, we detail our evaluation of the proposed research. In Section 5, we present results of preliminary experiments for our first three efforts. Finally, we finish with a discussion of the broader impact in Section 6, our proposed schedule of research in Section 7, and concluding thoughts in Section 8.

2 Background and Related Work

In this section, we present definitions, background material, and related work in the context of our proposed research efforts.

2.1 Finite Automata

Deterministic and non-deterministic finite automata (DFAs and NFAs) provide useful models of computation for identifying patterns in a string of symbols. A DFA, formally, is defined as a five-tuple, $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, and $F \subseteq Q$ is the set of accepting states. The finite alphabet defines the allowable symbols within the input string. The transition function takes, as input, the currently active state and a symbol, and the function returns a new active state.

An NFA modifies this five-tuple to be $(Q, \Sigma, Q_{start}, \delta, F)$, where $Q_{start} \subseteq Q$ is a set of initial states and $\delta : 2^Q \times \Sigma \rightarrow 2^Q$ is the transition function.[1] NFAs have the same representative power as DFAs, but have the advantage of being more spatially compact [78]. In this proposal, we use an alternate form of NFAs known as *homogeneous* NFAs. These automata restrict the possible transition rules such that all incoming transitions to a state must occur on the same symbol [23]. Because all transitions to a state occur on the same symbol, we can label states with symbols rather than labeling the transitions. We refer to these combined states and labels as *state transition elements* (STEs), following the nomenclature adopted by Dlugosch et al. [31]. Figure 1 depicts an NFA and a behaviorally-equivalent homogeneous NFA.

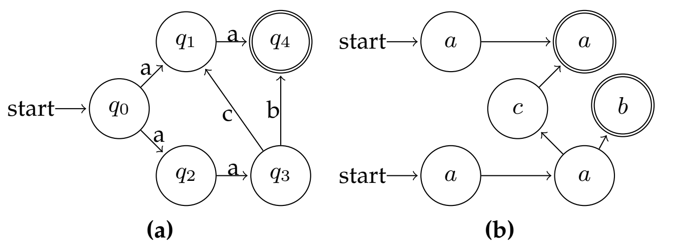

Figure 1: A behaviorally-equivalent NFA and homogeneous NFA (both accept exactly `aa`, `aab`, and `aaca`). Note that there is a singleton start state in (a) (i.e., $Q_{start} = \{q_0\}$), but there are two start states in (b).

2.2 Accelerating Automata Processing

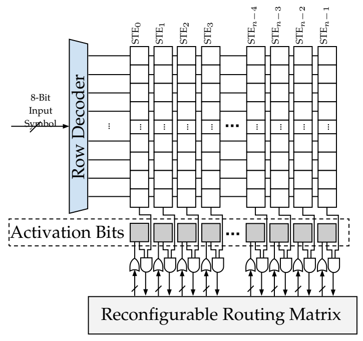

Figure 2: AP architecture. STEs are stored in a memory array, and edges are encoded in a reconfigurable routing matrix.

As improvements in semiconductor technology have slowed, but demand for increased throughput for complex algorithms remains, there is a trend in hardware design toward specialized accelerator architectures [68, 77]. A recent body of work studies the acceleration of finite automata (NFA and DFA) processing across multiple architectures. Becchi et al. have developed a set of tools and algorithms for efficient CPU-based automata processing [17]. Several regular-expression-matching and DFA-processing application-specific architectures have also been proposed [35, 38, 87, 94]. Some (e.g., UDP [36]) incorporate regular expression matching into an extract-transform-load pipeline, supporting a richer set of applications. There have been several efforts to develop memory-centric architectures for automata processing, such as Micron's D480 Automata Processor (AP) [31], Parallel Automata Processor (PAP) [85], Subramaniyan et al.'s Cache Automaton (CA) [86], and Xie et al.'s REAPR [102].

We briefly describe two architectures that accelerate automata processing applications: Micron's D480 AP and commodity FPGAs. These architectures are indicative of those we target in our proposed research efforts detailed in Section 3.

[1]NFAs traditionally support ε-transitions between a source and target state *without* consuming a symbol. These are not present in our definition of an NFA. An ε-transition may be removed by duplicating all incident transitions to the source state on the target state [31].

The AP is a hierarchical, memory-derived architecture for execution of homogeneous NFAs developed by Dlugosch et al. [31]. The processing core of the AP consists of a DRAM array and a reconfigurable routing matrix, representing the STEs and edges respectively. The architecture is depicted in Figure 2. In-cache automata processing architectures, such as Subramaniyan et al.'s Cache Automaton (CA) [86], implement a similar datapath.

Field-Programmable Gate Arrays are reconfigurable fabrics of look-up tables (LUTs), flip-flops (FFs), and block RAMs (BRAMs). LUTs can be configured to compute arbitrary logic functions and are connected together with memory using a reconfigurable routing fabric. This model allows FPGAs to form arbitrary circuits, which can be useful for prototyping. Xie et al. propose an AP-style processing model for NFA execution in which LUTs are used in place of columns of memory, flip-flops are used to store the activation bits for STEs, and transition signals are propagated through the FPGA's reconfigurable routing matrix [102]. Because of the general nature of the AP-style datapath, this proposal assumes such an architecture unless otherwise noted.

2.3 Software Maintenance

Software maintenance tasks are varied and account for a significant proportion of developer effort [65, 75]. In our proposed dissertation, we focus on the task of debugging, including aiding fault localization. *Fault localization* is an aspect of debugging that attempts to implicate particular statements or expressions as the likely source of undesirable behavior [104]. The development of debugging tools has a lengthy history [43, 56, 74, 106], and software debuggers are commonplace in development toolchains [59]. Ungar et al. argue that immediacy is important for debugging tasks [92]. There has also been significant effort devoted to improving the efficiency of debugging tools, such as quickly transferring control when a *breakpoint* is reached [53] and efficiently supporting large numbers of watchpoints [107]. Breakpoints allow a developer to begin interacting with a debugger by specifying locations in the program source at which to pause execution for inspection. These approaches provide debugging support for general purpose processors and languages. The technique proposed in this work is in the same spirit: we will provide immediacy for debugging big data pattern-matching applications through low-overhead breakpoints on specialized hardware and interactive, step-through program inspection.

Previous research has considered debugging for specialized hardware, including support for distributed sensor networks [81] and energy-harvesting systems [28]. Hou et al. developed a debugger for general-purpose GPU programs which leverages automatic dataflow recording to allow users to analyze errors after program execution [48]. Similarly, there are approaches for debugging FPGA applications [5, 40]; however, these techniques typically focus on inspection of the underlying hardware description, rather than programs written in high-level languages. We propose further developing the area of debugging for specialized processors by designing a technique for inspecting source-level program state during program execution on automata processing engines.

2.4 Program Synthesis and Verification

Program synthesis is a holistic term for automatically generating software from some input description. Recent efforts have focused on different applications of synthesis, such as sketching [4, 79, 80], programming by example [42], and automated program repair [61, 62]. Many of these approaches employ *counterexample-guided inductive synthesis* (CEGIS) to produce a final solution [80]. CEGIS is an iterative technique that constructs candidate solutions that are tested (typically via formal methods) for equivalence. A *counterexample*, or model of undesirable behavior, is provided if the candidate solution is incorrect, and this is used to begin the next iteration of synthesis. Further, there have been many efforts to synthesize state machines from held-out oracles or example data [83, 93], including Angluin's L^* and associated algorithms [7, 21], which learn finite automata from a held-out teacher. There has also been related efforts to understand the the number of queries of an oracle needed to learn a language as well as approximate learning of languages [6, 7, 39, 50].

Program verifiers and software model checkers prove that a program adheres to a provided specification or produce counterexamples that violate the specification [18]. These tools typically interleave the control flow graph (CFG) and a given specification automaton and explore the resulting graph to determine if any path leads to an error state in the specification. There has been significant research and engineering effort applied to making these techniques scalable and applicable to real-world applications [15, 19, 27, 45, 60].

We propose combining and adapting insights from CEGIS, automata learning, and software model checking to synthesize behaviorally-equivalent automata from legacy source code in Section 3.4.

3 Proposed Research and Technical Approach

Our goal is to design new programming models and maintenance tools that ease the adoption and use of hardware accelerators in data processing pipelines. We propose using finite automata as an intermediate representation of computation, and will design high-level languages and tools as well as hardware backends that support this abstraction. Concretely, we propose four research efforts for this dissertation, designing the following:

1. A high-level programming language, RAPID, for sequential pattern search applications.
2. A high-speed, interactive debugging system for the RAPID language that supports breakpoints and variable introspection.
3. An in-cache accelerator and associated optimizing compiler for execution of deterministic pushdown automata.
4. A system for porting existing software for use on hardware accelerators by automatically learning functionally-equivalent finite automata.

The remainder of this section is dedicated to describing each research effort and detailing our proposed technical approaches, but does not describe our proposed evaluation. A discussion of metrics and experiments may be found in Section 4.

3.1 Research: High-Level Languages for Automata Processing

The goal of this research thrust is to establish the feasibility of compiling an imperative, high-level programming language to a set of finite automata for execution on hardware accelerators. We propose extending a simple C- or Java-like language with domain-specific parallel control structures and developing algorithms to lower programs written in this language to an automata-based representation. We hypothesize:

> A high-level programming language will improve the conciseness of representing an algorithm while maintaining the performance of hand-crafted applications for hardware accelerators.

The work in this subsection was performed in collaboration with other researchers and is described in detail in [11, 12].

Approach. The output of this research effort is an initial prototype of the RAPID programming language, including a compiler that generates a set of finite automata from an input program. We propose supporting execution of RAPID programs on CPUs, GPUs, FPGAs, and Micron's D480 AP. The AP provides direct hardware support for processing automata [31]; for the remaining architectures, we propose adapting existing automata processing engines and execution cores to work with the RAPID language.

RAPID programs will be written in a combination of imperative and declarative styles, and we will design code abstractions similar to functions or procedures that allow for efficient reuse while mapping naturally to pattern-matching problems and the underlying automata computational model, which we refer to as *macros* and *networks*. A *macro* uses sequential control flow to define an algorithm for matching patterns in an input data stream. Macros in RAPID are similar to C-style macros and macros in low-level Automata Processor programs. The *network* contains a list of macros that are instantiated in parallel, allowing for simultaneous recognition of many patterns in data streams (e.g., [73, 97, 98, 108]).

We propose accelerating RAPID programs on a hardware accelerators by leveraging insights from *staged computation*, which is an approach for performing a portion of computation at compile time and leaving "holes" that are filled in at runtime to handle support for dynamic inputs [41, 66]. With respect to RAPID, imperative statements in the code will be executed at compile time to aid in generating finite automata, while declarative statements related to the input data stream will then be executed at runtime on the hardware accelerator.

We propose three parallel control structures (`foreach`, `some`, and `whenever`) to facilitate common pattern-matching tasks. These allow the concise specification of multiple, simultaneous comparisons against a single data stream and provide high-level support for variable-offset sliding window comparisons that are integral to many pattern-recognition problems (e.g., [37, 96, 108]).

Our proposed control structures, code abstraction, compilation strategy will allow RAPID programs to be written at a high level of abstraction while also achieving high throughputs when executed on hardware accelerators.

3.2 Research: Interactive Debugging for High-Level Languages and Accelerators

Although debugging support for CPUs is mature and fully-featured (e.g., including standard tools [82], successful technology transfer [15] and annual conferences [51]), the throughput of automata processing applications on CPUs is typically orders of magnitude slower than execution on hardware accelerators [63, 96], making CPUs too slow for effective debugging of automata processing. Unfortunately, current debugging techniques are limited or nonexistent for hardware accelerators. For architectures where there is no instruction stream, such as in FPGAs, the traditional method of inserting breakpoints is not available. Instead, debugging on FPGAs is often performed at the signal level using logic analyzers or scan chains [14, 55, 88, 100], exposing low-level state to software. The AP also provides no explicit debugging support, but does expose low-level state through APIs.

We propose designing an interactive, source-level debugger by building upon low-level signal inspection on hardware accelerators. We hypothesize that:

The automata abstraction reduces the program state that must be monitored at the signal level on hardware accelerators while still allowing for program semantics to be lifted to higher levels of abstraction and meaningfully supporting debugging.

Approach. In this research effort, we propose designing a prototype debugging system on top of the RAPID programming language described in the previous subsection. Our debugging system will include support for breakpoints and program data inspection. We propose supporting debugging on both the AP and Xilinx FPGAs without modifying the underlying accelerators. While we focus our presentation on RAPID, the general techniques we develop for exposing state from low-level accelerators to provide debugging support lay out a general path for providing such capabilities for other accelerators and languages. Our approach will leverage a combined hardware accelerator and CPU-software simulation system design to allow for both high-speed data processing as well as interactive debugging.

Hardware accelerators already contain much of the low-level hardware support needed to inspect the state of executing automata. For example, Micron's Automata Processor contains context-switching hardware resources, which are often left unused. Additionally, FPGA manufacturers provide logic analyzer APIs to inspect the values of signals during data processing. We will repurpose these hardware features to transfer control from the execution context on the accelerator to an interactive debugger on the host system.

We propose lifting captured automata state to the semantics of the source-level program through a series of mappings generated at compile time. We will design the RAPID compiler (proposed in Section 3.1) such that the mapping from source-level expressions to architecture-level automata states is traceable; our approach will also be applicable to any high-level programming language for which such a mapping from expressions to hardware resources may be inferred.

Finally, we propose a new form of breakpoints for data-processing applications. Setting breakpoints on particular expressions in a program is not directly supported by the automata processing paradigm. Instead, we will set and trigger breakpoints on input data, pausing execution after processing a fixed number of bytes. We can leverage these pauses to transparently provide the abstraction of more traditional breakpoints set on lines of code.

3.3 Research: Expressive Power of In-Memory Automata Accelerators

Processing of tree-structured or recursively-nested data is intrinsic to many computational applications. Data serialization formats such as XML and JSON are inherently nested (with opening and closing tags or braces, respectively), and structures in programming languages, such as arithmetic expressions, form trees of operations. Unfortunately, the expressive power of finite automata is not adequate for recognizing these recursively-nested structures [78].

We observe that *deterministic pushdown automata* (DPDA) provide a general-purpose computational model for processing tree-structured data. Pushdown automata extend basic finite automata with a stack. State transitions are determined by both the next input symbol and also the top of stack value. Determinism precludes stack divergence (i.e., simultaneous transitions never result in different stack values) and admits efficient hardware implementation. While somewhat restrictive, DPDAs are powerful enough to parse most programming languages and serialization formats. We hypothesize that:

An in-cache accelerator architecture supporting pushdown automata computation will support a rich class of applications, such as XML parsing, and allow for improved performance over state-of-the-art baselines.

Approach. We propose designing ASPEN, an Accelerated in-SRAM Pushdown ENgine that is a realization of deterministic pushdown automata in Last Level Cache (LLC). Our design is based on the insight that much of DPDA processing can be architected as LLC SRAM array lookups without involving the CPU. By performing DPDA computation in-cache, ASPEN avoids conventional CPU overheads such as random memory accesses and branch mispredictions. Execution of a DPDA with ASPEN will be divided into five stages: (1) input symbol match, (2) stack symbol match, (3) state transition, (4) stack action lookup, and (5) stack update, with each stage making use of SRAM arrays to encode matching and transition operations.

To scale to large DPDAs with thousands of states, ASPEN will adopt a hierarchical architecture while still processing one input symbol in one cycle. Further, ASPEN will support processing of hundreds of different DPDAs in parallel, as any number of LLC SRAM arrays can be re-purposed for DPDA processing.

To support direct adaptation of a large class of legacy parsing applications, we propose designing a compiler for converting existing grammars for common parser generators to DPDAs executable by ASPEN. We propose two key optimizations for improving the runtime of parsers on ASPEN. First, the architecture will support popping a reconfigurable number of values from the stack in a single cycle, a feature we call *multipop*. Second, our compiler will implement a state merging algorithm that reduces chains containing ε-transitions. Both of these optimizations are designed to reduce stalls in input symbol processing.

We have published, in collaboration with other researchers, a prototype architecture in [8]. We demonstrated that it is possible to implement a DPDA-based architecture with low overhead in LLC. Our proposed research will expand on this existing work by exploring additional use-cases for in-memory automata processing accelerators, such as their application to security problems.

3.4 Research: Adapting Legacy Code for Execution on Hardware Accelerators

As companies and individuals adopt hardware accelerators into their application workflows, they will need to port existing code to these new programming models. We wish to reduce the burden on developers tasked with porting legacy code. We hypothesize that:

> An algorithm that learns a set of finite automata from a legacy source code kernel, using a combination of automata learning and formal methods, can correctly synthesize a functionally-equivalent kernel computation, reduce the manual annotation and refactoring efforts of human developers, and efficiently represent real-world applications.

Approach. The output of this research effort is an algorithm that, given an input function deciding a regular language (written in a high-level language), produces a set of finite automata with functionally-equivalent behavior. At a high level, we propose employing a counterexample-guided inductive synthesis approach in this algorithm (see Section 2.4). We observe that an Angluin-style learning algorithm, such as L^*, falls within this category and learns a state machine that represents a language, L, using queries to a *minimally adequate teacher* [7]. For learning a finite automaton, this teacher must answer (1) *membership* queries (e.g., is string $s \in L$?) and (2) *equivalence* queries (e.g., is the language, L', of the candidate machine equivalent to L?).

Our insight is that the legacy kernel can be used as the basis for the minimally adequate teacher in our algorithm. For membership queries, we will execute the function on the given input and return the result. We will apply formal methods to answer equivalence queries, which we note is equivalent to the test input generation problem: *given the kernel and a candidate automaton, are there any inputs for which the outputs differ*? If no inputs satisfy this constraint, then we conclude that the automaton is functionally equivalent to the kernel code. We propose using software model checking and counterexample-guided abstraction refinement [27] to find these inputs. While these refinement semi-algorithms may not converge, previous work provides evidence that convergence is frequently achieved in a few iterations on methods of similar scale [16].

Because of the speculative nature of this research, we take several steps to mitigate risk. First, we will initially assume that input kernel functions decide a regular language (i.e., could be re-written as a regular expression). Time permitting, we will extend our algorithm to support approximate solutions for non-regular languages. We will study how user-provided input examples can guide our automata synthesis algorithm and the resulting quality of the approximation. Additionally, the software model checking algorithms we propose to use can be challenging to scale to real-world applications. We will allow developers to annotate their legacy code to aid the model checker and explore approaches to minimizing the annotation burden. We may also explore alternate approaches that attempt to directly infer automata from the control- or data-flow graphs of the legacy kernels.

4 Proposed Experiments and Evaluation

In this section we outline our proposed experiments and evaluation of the technical approaches detailed in Section 3. Taken collectively, the proposed research improves the programming support for—and performance of—hardware accelerators by leveraging an automata processing abstraction. Our evaluation will be conducted with respect to our stated goals of *performance*, *scalability*, *ease of use*, *expressiveness*, and *legacy support*.

4.1 Experiments: High-Level Languages for Automata Processing

This subsection outlines the experimental plan for evaluating the feasibility of compiling a high-level language (RAPID) to a set of finite automata for execution on hardware accelerators. Much of the work for this research effort has previously been published, and the techniques for evaluating the efficiency, scalability and expressiveness of the RAPID language follow those in our preliminary work [11,12].

To perform our evaluation, we will employ a benchmark suite of at least five real-world applications previously accelerated by automata processing [20,71,96,98,108]. By selecting from the recent literature, we ensure that we evaluate RAPID on indicative applications and baselines.

We will evaluate the **expressiveness** of the RAPID programming language by re-implementing the benchmark suite using our proposed language. To be successful, RAPID must be able to represent real-world applications; we will measure the number of benchmarks we can successfully implement. The RAPID versions of the benchmarks will then be used to conduct further experiments.

Due to the lock-step execution of automata on reconfigurable hardware accelerators, runtime performance of loaded designs is linear in the length of a given input stream. Therefore, we focus on evaluating the spatial **performance** of RAPID programs. On hardware accelerators, automata require physical hardware resources (i.e., STEs on the AP and LUTs and registers on FPGAs) for each state, in addition to signal routing resources. We will measure the overhead incurred by compiling RAPID benchmarks to hardware accelerator configurations compared with hand-crafted baselines. We wish to minimize the quantity of additional resources required by RAPID; keeping these overheads under 15% will demonstrate the success of our approach. We derive our success metric from the upper bound for overheads presented by Cong et al. for high-level synthesis on FPGAs [30].

Scalability, with regard to a programming language, measures how program size grows with respect to the size of the application being represented. We propose measuring total lines of code (LOC) for our benchmarks. For the hand-crafted, optimized baselines, we will measure the LOC needed to generate the automata, or—where the automata were built by hand—LOC in a commercial automata serialization format, which is roughly equivalent to the number of actions taken within a design tool. To be successful, RAPID must use significantly fewer LOC to represent a benchmark than the baseline implementation. Ultimately, a successful program representation will remain constant in size—or grow sublinearly—as application instances grow in size.

4.2 Experiments: Interactive Debugging for High-Level Languages and Accelerators

We propose using human studies to evaluate the **ease of use** of our interactive debugging tools for the RAPID language. Ultimately, we wish to understand how our technique affects a developer's abilities to localize faults in pattern-matching applications. We will formulate the study as an online survey that presents participants with a sequence of fault localization tasks, similar to the study in [64]. Participants will be shown ten RAPID programs (adapted from real-world applications) containing indicative bugs along with associated input data exposing these bugs. For half of these programs, we will also provide our debugging system for the participants to use. We will record the time needed for participants to identify the location of the bug in the program and record their responses. After conducting the study, we will analyze our collected data to identify if there is a statistically significant improvement in localization accuracy and/or time taken. Our proposed debugging system will be successful if there is a demonstrable improvement in either accuracy or time. We have already received IRB approval for the study described above.

Further, we propose evaluating the **scalability** of our prototype FPGA-based debugger implementation. We will measure both the time and space overheads of adding debugging support to applications in the ANMLZoo benchmark suite, a collection of diverse automata-based applications [96]. Our technique will be successful if the additional hardware resources do not exceed the capacity of a server-grade FPGA and if the runtime performance does not degrade below that of automata execution on CPU-based software engines.

4.3 Experiments: Expressive Power of In-Memory Automata Accelerators

For this research effort, we will evaluate the expressive power, scalability, and performance of our proposed in-cache hardware accelerator architecture and associated optimizing compiler. We will collect a suite of context-free grammars for common programming languages and serialization formats (e.g., [33, 34, 44]). We will use these to test the **expressive power** and **scalability** of our compiler. We will measure the hardware resources (e.g., STEs and SRAM arrays) needed to configure our proposed architecture to recognize inputs conforming to each grammar. Further, to evaluate the architecture, we will seek a collection of real-world applications that stress the hardware resources, including benchmarks that contain few large, highly-connected pushdown automata (e.g., XML parsing), as well as benchmarks with many small automata (e.g., packet filtering or signature-based intrusion detection). Such benchmarks support evaluation of the **scalability** of our datapath, including the reconfigurable routing matrix (see Section 2.2) and stack memory. Additionally, we will measure the **performance** (both in terms of modeled application throughput and energy) of these applications in comparison with state-of-the-art implementations for CPUs and other hardware accelerators. We will be successful if ASPEN outperforms the state-of-the-art for indicative applications, such as XML parsing, while also not exceeding power thresholds for modern CPUs.

4.4 Experiments: Adapting Legacy Code for Execution on Hardware Accelerators

To evaluate our new finite automata synthesis algorithms, we will seek a collection of existing kernel functions that map strings to Boolean values. For example, we may use applications from the ANMLZoo benchmark suite [96] or kernels used to evaluate string decision procedures (e.g., [47]). We will measure the annotation burden (in terms of the number and/or complexity) required to apply our algorithms to legacy code. Initially, we wish to determine whether it is possible for our proposed algorithm to successfully generate behaviorally-equivalent finite automata. As such, these experiments measure the **legacy support** provided by our technical approach. Further, we will measure the **scalability** of our algorithms by measuring the time needed to automatically synthesize a set of finite automata and the size (measured in number of states) of the resulting automata. To be successful, our algorithms must be able to run over the weekend [49, 69] and the resulting automata should not exceed the capacity of commercial FPGAs. Because our algorithms may produce approximate solutions for some applications, we propose measuring the accuracy of the synthesized automata using test cases (either provided as part of the original software or synthetically generated using a test-input generation tool [22,57,105]). We will be successful if our approach of using example inputs improves the accuracy of our approximate results for the given test data.

5 Preliminary Results

In this section, we present preliminary results from ongoing research. We focus on results for the proposed research on programming models (Section 3.1), debugging tools (Section 3.2), and in-cache automata accelerators (Section 3.3).

5.1 Results: High-Level Languages for Automata Processing

We evaluate RAPID against hand-crafted designs and parameter settings for five real-world benchmark applications *Association Rule Mining (ARM)* [98], *Brill part-of-speech tagging (Brill)* [108], *Exact match DNA search (Exact)* [20], *DNA string search with gaps (Gappy)* [20], and *MOTOMATA* [71]. The authors of the *ARM* [98] and *Brill* [108] benchmarks provided us with their automated scripts for generating automata. We recreated the remaining benchmarks, using algorithms and specifications published in previous work.

Table 1 lists design statistics for the benchmarks. We compare the lines of code needed to generate the automata. For *ARM*, the RAPID code requires six times fewer lines to represent, and *Brill* requires about half of the lines of the hand-crafted solution. For the *Gappy*, *Exact*, and *MOTOMATA* benchmarks, we present the lines of code in a commercial automata format, which is roughly equivalent to the number of actions taken within the design tool. In all cases, the RAPID program is significantly more compact than the automata it generates.

Table 1: Comparison between RAPID and hand-crafted code with respect to lines of code (LOC) and space utilization on AP and FPGA targets. Lower values for AP States, FPGA LUTs and FPGA registers indicate a smaller footprint; lower values for AP MBRA indicate less stress on the routing network.

Benchmark		Source LOC	Automata LOC	STEs	AP STEs	AP MBRA	AP Clk	FPGA LUTs	FPGA Reg
ARM	H	118	301	79	58	20.8%	1	73	76
	R	18	214	58	56	20.8%	1	83	65
Brill	H	1,292	9,698	3,073	1,514	65.4%	1	201	1483
	R	688	10,594	3,322	1,429	52.6%	1	358	1360
Exact	H	–[†]	193	28	27	4.2%	1	6	25
	R	14	85	29	27	4.2%	1	28	27
Gappy	H	–[†]	2,155	675	123	77.1%	1	73	123
	R	30	2,337	748	399	70.8%	1	52	399
MOTOMATA	H	–[†]	587	150	149	75.0%	0.5	114	148
	R	34	207	53	72	75.0%	1	85	60

R – RAPID *H – Hand-coded* [†] Automata were directly hand-coded

As an approximation for the size of the resulting automaton, we measure the number of STEs generated and the number of STEs loaded to the AP after placement and routing. For most benchmarks, RAPID-generated automata contain fewer device STEs, taking up less space on the device.

In Table 1, we also present the performance of RAPID programs compared to hand-crafted automata based on placement and routing statistics for the AP. The placement and routing tools modify the original automaton to better match the architectural design of the AP. The final count of STEs is given by *AP STEs*. Mean BR allocation (*AP MBRA*) is a metric provided by the AP SDK that approximates the routing complexity of the design. Here, a lower number is better, signifying lower congestion within the routing matrix. RAPID does not appear to introduce any additional routing congestion. The *AP Clk* column indicates whether the clock cycle of the AP must be reduced to accommodate a design. In one instance (the RAPID *MOTOMATA* program), the clock cycle must be halved due to a limitation in signal propagation between counters and combinatorial elements in the current generation AP. However, the RAPID version is four times more compact. Although this is a performance loss for a single instance, it is a net performance gain for a full problem, which will fill the AP board: four times as many instances execute in parallel at half the speed, for a net improvement factor of two. Although RAPID provides a higher level of abstraction than automata, the final device binaries are more compact, using fewer resources on the AP.

Finally, we evaluate the space efficiency of the FPGA engines our tools produce. We synthesize our designs for a Xilinx Kintex UltraScale XCKU060. Table 1 also lists the number of LUTs and registers needed to implement the hardware description of the benchmark. Lower numbers indicate smaller footprints for the circuits, which allows for more widgets to be run in parallel. As with the AP results, RAPID programs do not incur significant space overheads on the FPGA.

These preliminary results demonstrate that our proposed RAPID programming language is **expressive** enough—and scales—to support real-world applications, while incurring **performance** overheads well within our 15% threshold.

5.2 Results: Interactive Debugging for High-Level Languages and Accelerators

In this subsection we evaluate our debugging system using a human study by presenting participants with code snippets and asking them to localize seeded defects (see Section 2.3). Our IRB-approved human study[2] was formulated as an online survey that presented participants with a sequence of fault localization tasks. We presented each participant with ten randomly-selected and ordered fault localization tasks from a pool of twenty; stimuli were presented following the design described in Section 4.2.

Participants were all voluntary and predominantly from the University of Virginia, including intermediate and upper-level undergraduate students, graduate students, and members of the D480 AP professional development team. In total, 61 users participated in our survey each completing ten fault localization tasks, resulting in over 600 individual data points.

To measure the effect of debugging information on programmer performance, we used the following metrics: accuracy and time taken. We defined accuracy as the number of correctly-identified faults. We manually assessed correctness after the completion of the survey, taking into account both the marked fault location and justification text provided. Using Wilcoxon signed-rank tests, we did not observe a statistically significant difference in time taken to localize faults ($p = 0.55$); however, we determined that there is a statistically significant increase in accuracy when participants were given our debugging information ($p = 0.013$). Mean accuracy increased from 45.1% to 55.1%, meaning participants were 22% more accurate when using our tool. Thus, our proposed debugging system improves **ease of use** by increasing fault localization accuracy with no loss of time.[3]

5.3 Results: Expressive Power of In-Memory Automata Accelerators

In this subsection, we evaluate ASPEN on real-world applications with indicative workloads. First we evaluate the generality of ASPEN and our proposed optimizations by compiling several parsers for the architecture. Second, we evaluate runtime performance for one motivating application.

5.3.1 Parsing Generality

We demonstrate compilation of four different languages: *Cool*, an object oriented programming language [2]; *DOT*, the language used by the GraphViz graph visualization tool [34]; *JSON* [33]; and *XML* [44]. We selected these benchmarks because grammar specifications were readily available. Importantly, no modification to existing legacy grammars was necessary to support compilation to ASPEN. The architecture is general-purpose enough to support these diverse applications, and our prototype compiler supports a large class of existing parsers. The grammar specifications ranged from 19–61 productions.

We measured the average time across ten runs of our compiler and optimizations. Compilation of all grammars, including optimization, is well below 5 seconds, meaning that compilation of grammars is not a significant bottleneck with ASPEN. With both our multipop and epsilon reduction optimizations enabled, we observe a 47%, on average, decrease in the number of states. The number of epsilon states is reduced by 65% on average. Reducing epsilon states increases the

[2]University of Virginia IRB for Social and Behavioral Sciences #2016-0358-00. (This study was conducted before transferring to The University of Michigan.)

[3]For additional analyses and a discussion of threats to validity, please see [24].

runtime performance of parsing on ASPEN. These preliminary results give us confidence that the ASPEN compiler **scales** to—and has the **expressive power** to represent—real-world grammars.

5.3.2 Performance of XML Parsing

We evaluate ASPEN against the baseline XML tools Expat (v.2.0.1) [26], a non-validating parser, and Xerces-C (v.3.1.1) [13], a validating parser and part of the Apache project. We assume that input data is already loaded into main memory when modeling performance. Our 23 XML benchmarks are derived from Parabix [58], Ximpleware [103] and the UW XML repository [1].

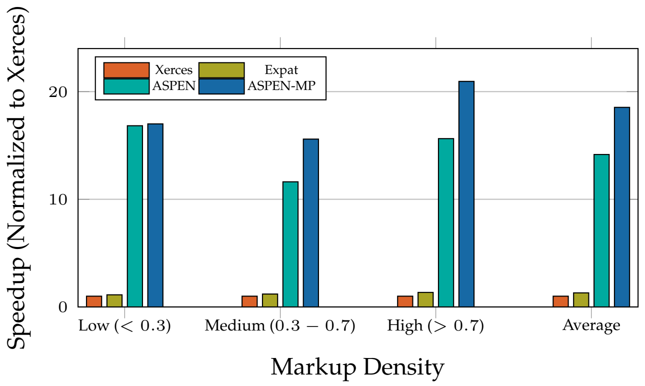

Figure 3: Aggregate speedup (relative to Xerces) of ASPEN over CPU-based parsers. Performance of ASPEN was modeled for 23 benchmarks. Markup density measures the frequency at which tokens appear in the input. Performance of ASPEN with and without multipop (MP) is shown.

Figure 3 compares ASPEN's performance against Expat and Xerces; speedups are normalized to Xerces. We evaluate two DPDA configurations: (1) ASPEN-MP has both multipop and epsilon merging optimizations enabled and (2) ASPEN, which only enables epsilon merging. We group our XML datasets based on markup density which is an indirect measure of XML document complexity. There is a noticeable trend in performance benefits of ASPEN-MP over ASPEN as markup density increases. As the density increases, tokens are generated more frequently, and ε-transition stalls are less likely to be masked by the tokenization stage of the pipeline. ASPEN-MP reduces the number of stalling cycles during parsing, thus improving performance with high markup density. Additionally, the speedup of ASPEN over Expat and Xerces increases with the markup density of the input XML document, because the CPU-based parsers perform more poorly as token density increases. ASPEN-MP achieves 30% improvement in both performance and energy over ASPEN. Overall, averaged across the datasets evaluated, ASPEN-MP takes 704.5 ns/kB. When compared with Expat, a 14.1× speedup is achieved. ASPEN-MP also achieves 18.5× speedup over Xerces. In summary, ASPEN outperforms state-of-the-art CPU-based XML parsers, and our proposed multipop optimization produces additional **performance** gains.

6 Broader Impact

The ultimate goal of this proposed dissertation is to improve the programmability of—and ease the adoption of—hardware accelerators in data processing pipelines. To be successful, we need not only new technologies and approaches, but also to educate and mentor the next generation of programmers, scientists, and engineers.

Tools to Promote Adoption. In an effort to ease the adoption of hardware accelerators and to promote further—and reproducible—research on the automata abstraction, we will make the prototype tools developed as part of the proposed research efforts freely available. We have previously developed and released MNCaRT, a unified ecosystem for automata processing accelerator

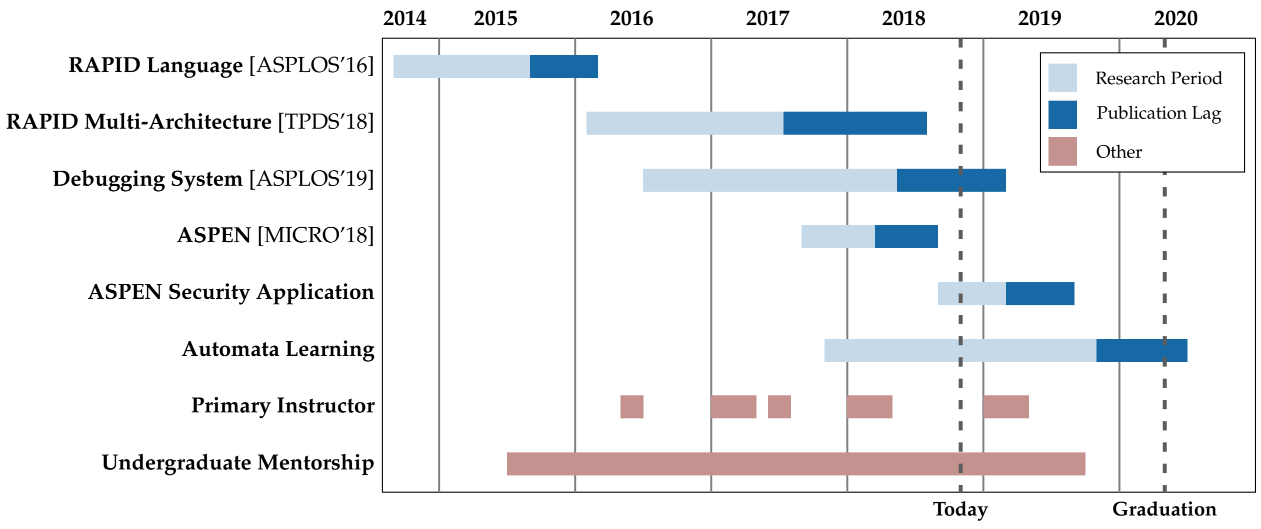

Figure 4: Proposed dissertation work schedule.

research [9,10]. We will extend this collection of tools with the new artifacts we develop as part of the research effort.

Mentorship and Diversity. To train the next generation of researchers and engineers, we will involve undergraduate students in the proposed research agenda. Mentorship is a first-order desire for this dissertation. We are particularly interested in attracting and mentoring students from genders and groups typically underrepresented in computer science. Such undergraduate researchers have contributed to the preliminary results, including Matthew Casias (UVA '19), first author on a peer-reviewed publication detailing interactive debugging for automata processing and hardware accelerators [24]. We have mentored an additional five undergraduate students (including two females as well as students with limited computing background) on projects related to autonomous vehicle resiliency [46], the use of automata processing in kinesiology, and modeling hard disk failures. For work proposed in this dissertation, we will recruit students from the University's Undergraduate Research Opportunity Program (UROP) and the Girls Encoded Explore Computer Science Research program as well as targeted invitations to students in courses we have taught. These activities are synergistic with our recent efforts to organize a speaker series highlighting the diverse research, careers, and backgrounds within the computer science discipline. We successfully proposed and received funding—with 50% matching departmental support—through the Rackham Faculty Allies Diversity Grant program.

7 Schedule

Figure 4 outlines the research timeline for the proposed dissertation. We anticipate completing the proposed research efforts in 1.5 years, with an expected graduation in May 2020. We have completed the research described in Section 3.1 to develop a high-level language for automata processing. The results of this effort have been published in [11,12]. We have developed a working prototype and conducted an initial human study for our research on interactive debugging (Section 3.2), and a manuscript detailing our findings has been accepted for publication [24]. For the expressive power of in-cache automata processors (Section 3.3), we have completed design of a prototype compiler and architecture, which has been published in [8]. We are currently conducting additional research on the use of our prototype architecture for use in intrusion detection systems, and we expect to complete this work by April 2019. Finally, we have begun very preliminary work on the synthesis of automata from legacy source code (Section 3.4), and we anticipate completing this effort by December 2019.

Our proposed schedule includes considerable slack for teaching and mentorship related to these projects (see Section 6), as well as research and engineering for projects outside the scope of this dissertation. Such projects include collaborative research (with multiple university and industry partners) on increasing the resiliency of autonomous vehicle systems through the use of online monitoring, binary transformations, redundant hardware, and automated program repair. We leave open the possibility of additional teaching activities and course design, occurring concomitantly with the proposed dissertation research.

We have previously targeted venues such as the *International Symposium on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), the *International Symposium on Microarchitecture* (MICRO), and *Transactions on Parallel and Distributed Systems* (TPDS). We propose targeting similar venues for the remaining work, as well as other top-tier venues in architectures, software engineering, and programming languages such as the *International Conference on Software Engineering* (ICSE), the *Symposium on Principles of Programming Languages* (POPL), and the *International Symposium on High-Performance Computer Architecture* (HPCA).

8 Conclusion

Data is being collected at increasing rates, and the demand from business and research leaders for real-time analyses often necessitates the use of dedicated hardware accelerators in processing pipelines. However, these accelerators are often very difficult to program and debug. In this proposal, we present a multi-faceted approach to easing the adoption of hardware accelerators through the use of finite automata. We hypothesize that automata processing provides a suitable abstraction for new programming models on reconfigurable hardware accelerators.

We propose four research efforts to simultaneously maintain the *scalability* and *performance* of hardware accelerators, while also improving their *ease of use*, *expressive power*, and *legacy support*. First, we propose developing a high-level language that compiles to finite automata to accelerate sequential pattern-matching applications. Second, we propose a technique for high-speed, interactive debugging on reconfigurable accelerators that bridges the semantic gap between low-level automata state and a high-level programming language. Third, we propose extending the expressive power of automata processing hardware architectures to support deterministic pushdown automata computations, such as parsing. Fourth, we propose a CEGIS- and formal methods-based approach to automatically synthesizing finite automata from legacy source code. Together, these components help ease the adoption and use of hardware accelerators for data analysis applications, while supporting high-performance computation.

References

[1] XML Data Repository. `http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html`.

[2] A. Aiken. Cool: A portable project for teaching compiler construction. *SIGPLAN Not.*, 31(7):19–24, July 1996.

[3] G. Alonso. FPGAs in data centers. *Queue*, 16(2):60:52–60:57, Apr. 2018.

[4] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, pages 1–8, Oct 2013.

[5] H. Angepat, G. Eads, C. Craik, and D. Chiou. NIFD: Non-intrusive FPGA debugger – debugging FPGA 'threads' for rapid HW/SW systems prototyping. In *International Conference on Field Programmable Logic and Applications*, pages 356–359, Aug 2010.

[6] D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76 – 87, 1981.

[7] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.

[8] K. Angstadt, A. Subramaniyan, E. Sadredini, R. Rahimi, K. Skadron, W. Weimer, and R. Das. Cache automaton. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51. IEEE, 2018. To appear.

[9] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron. MNCaRT: An open-source, multi-architecture automata-processing research and execution ecosystem. *IEEE Computer Architecture Letters*, 17(1):84–87, Jan 2018.

[10] K. Angstadt, J. Wadden, W. Weimer, and K. Skadron. MNRL and MNCaRT: An open-source, multi-architecture state machine research and execution ecosystem. Technical Report CS2017-01, University of Virginia, 2017.

[11] K. Angstadt, J. Wadden, W. Weimer, and K. Skadron. Portable programming with RAPID. *IEEE Transactions on Parallel and Distributed Systems*, 2018. *To appear*.

[12] K. Angstadt, W. Weimer, and K. Skadron. RAPID programming of pattern-recognition processors. In *Architectural Support for Programming Languages and Operating Systems*, pages 593–605, 2016.

[13] Apache Software Foundation. Xerces C++ XML parser. `http://xerces.apache.org/xerces-c/`.

[14] Z. K. Baker and J. S. Monson. In-situ FPGA debug driven by on-board microcontroller. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 219–222, April 2009.

[15] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, pages 1–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[16] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 103–122, Berlin, Heidelberg, 2001. Springer-Verlag.

[17] M. Becchi. Retular expression processor. `http://regex.wustl.edu`, 2011. Accessed 2017-04-06.

[18] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, Oct 2007.

[19] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 189–198, Austin, TX, 2010. FMCAD Inc.

[20] C. Bo, K. Wang, Y. Qi, and K. Skadron. String kernel testing acceleration using the Micron Automata Processor. In *Workshop on Computer Architecture for Machine Learning*, 2015.

[21] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In *International Joint Conference on Artificial Intelligence*, 2009.

[22] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[23] P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1):75–90, 2000.

[24] M. Casias, K. Angstadt, T. Tracy II, K. Skadron, and W. Weimer. Debugging support for pattern-matching languages and accelerators. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19. ACM, 2019. To appear.

[25] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 7:1–7:13, Piscataway, NJ, USA, 2016. IEEE Press.

[26] J. Clark. The Expat XML parser. `http://expat.sourceforge.net`.

[27] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, Sept. 2003.

[28] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *Architectural Support for Programming Languages and Operating Systems*, pages 577–589, 2016.

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

[30] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[31] 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, 2014.

[32] DNV GL. Are you able to leverage big data to boost your productivity and value creation? `https://www.dnvgl.com/assurance/viewpoint/viewpoint-surveys/big-data.html`, 2016.

[33] ECMA Technical Committee 39. The JSON Data Interchange Format. Technical Report ECMA-404 1st Edition, ECMA, Oct. 2013.

[34] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies. Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[35] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien. Fast support for unstructured data processing: the unified automata processor. In *International Symposium on Microarchitecture*, pages 533–545, 2015.

[36] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. In *International Symposium on Microarchitecture*, pages 55–68. ACM, 2017.

[37] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP '96, pages 120–, Washington, DC, USA, 1996. IEEE Computer Society.

[38] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. HARE: hardware accelerator for regular expressions. In *International Symposium on Microarchitecture*, pages 1–12, 2016.

[39] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447 – 474, 1967.

[40] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging FPGA circuits. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 41–50, March 2001.

[41] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1-2):147–199, Oct. 2000.

[42] S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Proceedings of the 8th International Joint Conference on Automated Reasoning*, pages 9–14, Berlin, Heidelberg, 2016. Springer-Verlag.

[43] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. VIDA: Visual interactive debugging. In *International Conference on Software Engineering*, pages 583–586, May 2009.

[44] E. R. Harold and W. S. Means. *XML in a Nutshell, Third Edition*. O'Reilly Media, Inc., 2004.

[45] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[46] K. Highnam, K. Angstadt, K. Leach, W. Weimer, A. Paulos, and P. Hurley. An uncrewed aerial vehicle attack scenario and trustworthy repair architecture. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 222–225, June 2016.

[47] P. Hooimeijer and W. Weimer. StrSolve: solving string constraints lazily. *Automated Software Engineering*, 19(4):531–559, Dec 2012.

[48] Q. Hou, K. Zhou, and B. Guo. Debugging GPU stream programs through automatic dataflow recording and visualization. In *SIGGRAPH Asia*, pages 153:1–153:11, 2009.

[49] S. Hou, L. Zhang, T. Xie, and Jia-Su Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *IEEE International Conference on Software Maintenance*, pages 257–266, Sept 2008.

[50] F. Howar, B. Steffen, and M. Merten. From ZULU to RERS. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 687–704, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[51] IEEE Computer Society. *2017 IEEE International Workshop on Program Debugging (IWPD), Symposium on Software Reliability Engineering Workshops, ISSRE Workshops*, Toulouse, France, Oct 2017. IEEE.

[52] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[53] P. B. Kessler. Fast breakpoints: Design and implementation. In *Programming Language Design and Implementation*, pages 78–84, 1990.

[54] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 107–127, Berkeley, CA, USA, 2018. USENIX Association.

[55] G. Knittel, S. Mayer, and C. Rothlaender. Integrating logic analyzer functionality into VHDL designs. In *2008 International Conference on Reconfigurable Computing and FPGAs*, pages 127–132, Dec 2008.

[56] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[57] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 475–485, New York, NY, USA, 2018. ACM.

[58] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. D. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *International Symposium on High Performance Computer Architecture*, pages 373–384, 2012.

[59] N. Matloff and P. J. Salzman. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, San Francisco, CA, USA, 2008.

[60] K. L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[61] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[62] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE.

[63] M. Nourian, X. Wang, X. Yu, W. Feng, and M. Becchi. Demystifying automata processing: GPUs, FPGAs, or Micron's AP? In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 1:1–1:11, New York, NY, USA, 2017. ACM.

[64] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209, 2011.

[65] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[66] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transaction Programming Languages and Systems*, 21(2):324–369, Mar. 1999.

[67] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[68] B. Reagen, R. Adolf, Y. S. Shao, G. Y. Wei, and D. Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *International Symposium on Workload Characterization*, pages 110–119, Oct 2014.

[69] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

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

[71] I. Roy and S. Aluru. Finding motifs in biological sequences using the Micron Automata Processor. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, pages 415–424, 2014.

[72] I. Roy, N. Jammula, and S. Aluru. Algorithmic techniques for solving graph problems on the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, pages 283–292, May 2016.

[73] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru. High performance pattern matching using the automata processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, pages 1123–1132, 2016.

[74] E. Satterthwaite. Debugging tools for high level languages. *Software: Practice and Experience*, 2(3):197–217, 1972.

[75] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[76] J. M. Shalf and R. Leland. Computing beyond Moore's law. *IEEE Computer*, 48(12):14–23, Dec 2015.

[77] Y. S. Shao, B. Reagen, G. Y. Wei, and D. Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *International Symposium on Computer Architecture*, pages 97–108, June 2014.

[78] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2nd edition, 2006.

[79] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 136–148, New York, NY, USA, 2008. ACM.

[80] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[81] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global views of distributed program execution. In *Conference on Embedded Networked Sensor Systems*, pages 141–154, 2009.

[82] R. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB*. Free Software Foundation, 2002.

[83] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems*, SFM 2011, pages 256–296, Bertinoro, Italy, 2011. Springer Berlin Heidelberg.

[84] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[85] A. Subramaniyan and R. Das. Parallel automata processor. In *International Symposium on Computer Architecture*, pages 600–612, New York, NY, USA, 2017. ACM.

[86] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50, pages 259–272, New York, NY, USA, 2017. ACM.

[87] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch. HAWK: hardware support for unstructured log processing. In *International Conference on Data Engineering*, pages 469–480, 2016.

[88] A. Tiwari and K. A. Tomko. Scan-chain based watch-points for efficient run-time debugging and verification of FPGA designs. In *Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference, 2003.*, pages 705–711, Jan 2003.

[89] T. Tracy II, Y. Fu, I. Roy, E. Jonas, and P. Glendenning. Towards machine learning on the Automata Processor. In *Proceedings of ISC High Performance Computing*, pages 200–218, 2016.

[90] T. Tracy II, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins. Nondeterministic finite automata in hardware—the case of the Levenshtein automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[91] L. D. Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio. The role of CAD frameworks in heterogeneous FPGA-based cloud systems. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 423–426, Nov 2017.

[92] D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *Communications of the ACM*, 40(4):38–43, Apr. 1997.

[93] F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, Jan. 2017.

[94] J. van Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *International Symposium on Microarchitecture*, pages 461–472, 2012.

[95] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron. Generating efficient and high-quality pseudo-random behavior on automata processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 622–629, Oct 2016.

[96] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *International Symposium on Workload Characterization*, IISWC '16, pages 1–12, Sept 2016.

[97] K. Wang, E. Sadredini, and K. Skadron. Sequential pattern mining with the Micron Automata Processor. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 135–144, New York, NY, USA, 2016. ACM.

[98] K. Wang, M. Stan, and K. Skadron. Association rule mining with the Micron Automata Processor. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, 2015.

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

[100] T. Wheeler, P. Graham, B. E. Nelson, and B. Hutchings. Using design-level scan to improve FPGA design observability and controllability for functional verification. In *Proceedings of*

the 11th International Conference on Field-Programmable Logic and Applications, FPL '01, pages 483–492, London, UK, UK, 2001. Springer-Verlag.

[101] L. Wirbel. Xilinx SDAccel: A unified development environment for tomorrow's data center. Technical report, The Linley Group, 2014.

[102] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. REAPR: Reconfigurable engine for automata processing. In *27th International Conference on Field Programmable Logic and Applications*, FPL '17, pages 1–8, Sept 2017.

[103] XimpleWare. Ximpleware XML dataset. `http://www.ximpleware.com/xmls.zip`.

[104] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: Fault localization using time spectra. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 81–90, New York, NY, USA, 2008. ACM.

[105] M. Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl`, 2014. Accessed 2018-11-30.

[106] P. T. Zellweger. *Interactive Source-level Debugging for Optimized Programs (Compilation, High-level)*. PhD thesis, University of California, Berkeley, 1984.

[107] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Compiler Construction*, pages 147–162, Berlin, Heidelberg, 2008.

[108] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron. Brill tagging on the Micron Automata Processor. In *Proceedings of the 9th IEEE International Conference on Semantic Computing*, pages 236–239, 2015.

[109] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *High Performance Computing, Networking, Storage and Analysis*, pages 35:1–35:12, 2016.