Improving Programming Support for Hardware Accelerators Through Automata Processing Abstractions

by

Kevin A. Angstadt

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Doctoral Committee:

Professor Westley Weimer, Chair
Assistant Professor Reetuparna Das
Assistant Professor Jean-Baptiste Jeannin
Professor Kevin Skadron, University of Virginia

If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.

— Donald Knuth

Ein neues Tor ins Unglaubliche und ins Mögliche, ein neuer Tag, an dem alles geschehen konnte, wenn man es nur wollte.

— Tove Jansson, *Muminvaters wildbewegte Jugend*

Kevin A. Angstadt

angstadt@umich.edu

ORCID iD: 0000-0002-0104-5257

ACKNOWLEDGMENTS

Research is an inherently collaborative endeavor, and throughout this long journey, I have been supported by some phenomenal individuals. I would like to acknowledge, with brevity unbefitting of their contributions, many of those who helped me reach this moment in my life.

First, I would like to thank my advisor, Westley Weimer, who has patiently guided me through my journey of doctoral studies from start to finish. Wes continually challenged me to step outside my academic comfort zone while helping me learn the skills necessary to be a successful researcher. Wes's knowledge of program analysis techniques played a key role in the success of the work described in this dissertation. I may have needed to take graduate-level Programming Languages two and a half times, but I got there eventually. I am also seriously indebted to Wes's efforts to support my mentorship and teaching interests. I would not be the educator and scholar I am today without Wes. Finally, I must thank Wes for always being a good sport when it comes to terrible puns—even in our most overworked hours testing and implementing quadcopter-based software systems.

Next, I would like to thank Kevin Skadron, my former advisor and continued mentor. Kevin is the reason this dissertation exists; he got me hooked during a visit to the University of Virginia when he described a new project involving an

experimental processor. Little did I know that it would be the start of a six-year journey into understanding how to best leverage finite automata to program new kinds of hardware! I am also thankful for Kevin's ability to always consider the "big picture" and to remind me of its importance.

None of this work would be possible without the expertise and experience of my collaborators. Thank you to Jack Wadden, Tommy Tracy II, Matt Casias, Arun Subramaniyan, Xiaowei Wang, Elaheh Sadredini, Reza Rahimi, Vinh Dang, Ted Xie, Nathan Brunelle, Chunkun Bo, Dan Kramp, Reetuparna Das, Stephanie Forrest, Jean-Baptise Jeannin, Mircea Stan, and Lu Feng for all that they have taught me while we conducted research together. To my computer architecture colleagues, thank you for your patience whenever I would ask ignorant questions. To my software engineering and programming languages colleagues, thank you for your patience whenever I would ask ignorant questions.

I am also very much appreciative of my officemates over the years: Jonathan Dorn, Kevin Leach, Yu Huang, Madeline Endres, Colton Holoday, Zohreh Sharafi, Jamie Floyd, Kate Highnam, Hammad Ahmad, Fee Christoph, Yirui Liu, Ryan Krueger, Xinyu Liu, and Martin Kellogg. I truly enjoyed our conversations over the years and their willingness to teach me about their research.

I also wish to acknowledge my teaching colleagues and mentors. While teaching is not an explicit part of the doctoral experience, it was a significant portion of *my* experience. Teaching is what has kept me motivated to finish. I am indebted to Amir Kamil, Dave Paoletti, Marcus Darden, Mark Sherriff, Luther Tychonievich,

Ed Harcourt, and Patti Frazer Lock (among others) for all that they have done to help my development as an educator over the years.

I would not have been able to complete this degree without the unconditional love and support of my family (Mom, Dad, Mike, Charlotte, Linnea, Shay, my grandparents, and my aunts and uncles). I might not have always been able to communicate clearly to them what my research is about, but they've stuck by me nonetheless. I'm fortunate to be able to celebrate my successes and overcome my setbacks with them by my side. To my family: thank you, and I love you.

Thank you to my friends (Katja, Elaine, Nya, Terry, Liz, Joey, Tristan, Erin, Samyukta, Christabel, and Clara, among others) for tolerating my quirkiness. They have made this six-year journey significantly more fun and tolerable.

Finally, thank you to you, the reader. The fact that you are reading this now means that my efforts were not for naught.

If I have forgotten to thank you, I apologize. It has more to do with fatigue than anything else.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODE LISTINGS

LIST OF ALGORITHMS

LIST OF ACRONYMS

μJ	Microjoule
AL	Stack Action Lookup
ANML	Automata Network Markup Language
ANOVA	Analysis of Variance
AP	Automata Processor
API	Application Programming Interface
ARM	Association Rule Mining
ART	Aligned Rank Transform
ASIC	Application-Specific Integrated Circuit
ASPEN	Accelerated in-SRAM Pushdown ENgine
ATR	Automata-to-Routing
AUC	Area Under Curve
BRAM	Block RAM

BSD	Berkeley Software Distribution
C-BOX	Control Box
CA	Cache Automaton
CEGIS	Counterexample-Guided Inductive Synthesis
CFG	Context-Free Grammar (or Control-Flow Graph)
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
DFA	Deterministic Finite Automaton
DNA	Deoxyribonucleic Acid
DOM	Document Object Model
DPDA	Deterministic Pushdown Automaton
DRM	Disjoint Report Merging
DSL	Domain-Specific Language
EPLD	Erasable Programmable Logic Device
ER	Entity Resolution
FF	Flip-Flop

FIS	Frequent Itemset
FPGA	Field-Programmable Gate Array
GB	Gigabyte
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HDPDA	Homogeneous Deterministic Pushdown Automaton
HLS	High-Level Synthesis
HMD	Hardware-based Malware Detector
IDS	Intrusion Detection System
ILA	Integrated Logic Analyzer
IM	Input Match
IO	Input/Output
IP	Intellectual Property
IRB	Institutional Review Board
ISA	Instruction Set Architecture

JSON	JavaScript Object Notation
KB	Kilobyte
LLC	Last Level Cache
LOC	Lines of Code
LUT	Look-up Table
MARTINI	Memory Address Representation To INfer Intrusions
MAT	Minimally Adequate Teacher
MB	Megabyte
MISD	Multiple Instruction, Single Data
MNCART	MNRL Network Computation and Research Testbed
MNRL	MNRL Network Representation Language
NFA	Non-Deterministic Finite Automaton
NM	Nanometer
NS	Nanosecond

OS	Operating System
PAC	Probably Approximately Correct
PAL	Programmable Array Logic
PANDA	Platform for Architecture-Neutral Dynamic Analysis
PCRE	Perl-Compatible Regular Expressions
PDA	Pushdown Automaton
PJ	Picojoule
PLD	Programmable Logic Device
PLY	Python Lex-Yacc
PS	Picosecond
QEMU	Quick EMUlator
RAM	Random Access Memory
RF	Random Forest
ROC	Receiver Operating Characteristic
RTL	Register-Transfer Level
SAX	Simple API for XML

SM	Stack Match
SPM	Sequential Pattern Mining
SRAM	Static RAM
ST	State Transition
STE	State Transition Element
SU	Stack Update
TDP	Thermal Design Power
TOS	Top of Stack
TPU	Tensor Processing Unit
UML	Unified Modeling Language
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VIO	Virtual Input/Output
VPR	Versatile Place and Route
W	Watt
XML	Extensible Markup Language

ABSTRACT

The adoption of hardware accelerators, such as Field-Programmable Gate Arrays, into general-purpose computation pipelines continues to rise, driven by recent trends in data collection and analysis as well as pressure from challenging physical design constraints in hardware. The architectural designs of many of these accelerators stand in stark contrast to the traditional von Neumann model of CPUs. Consequently, existing programming languages, maintenance tools, and techniques are not directly applicable to these devices, meaning that additional architectural knowledge is required for effective programming and configuration.

Current programming models and techniques are akin to assembly-level programming on a CPU, thus placing significant burden on developers tasked with using these architectures. Because programming is currently performed at such low levels of abstraction, the software development process is tedious and challenging and hinders the adoption of hardware accelerators.

This dissertation explores the thesis that theoretical 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 hardware representations that enable high-performance execution on hardware accelerators. We adopt a principled hardware/software co-design methodology to develop a

programming model providing the key properties that we observe are necessary for success, namely performance and scalability, ease of use, expressive power, and legacy support.

First, we develop a framework that allows developers to port existing, legacy code to run on hardware accelerators by leveraging automata learning algorithms in a novel composition with software verification, string solvers, and high-performance automata architectures. Next, we design a domain-specific programming language to aid programmers writing pattern-searching algorithms and develop compilation algorithms to produce finite automata, which supports efficient execution on a wide variety of processing architectures. Then, we develop an interactive debugger for our new language, which allows developers to accurately identify the locations of bugs in software while maintaining support for high-throughput data processing. Finally, we develop two new automata-derived accelerator architectures to support additional applications, including the detection of security attacks and the parsing of recursive and tree-structured data. Using empirical studies, logical reasoning, and statistical analyses, we demonstrate that our prototype artifacts scale to real-world applications, maintain manageable overheads, and support developers' use of hardware accelerators. Collectively, the research efforts detailed in this dissertation help ease the adoption and use of hardware accelerators for data analysis applications, while supporting high-performance computation.

CHAPTER 1

Introduction

Hardware accelerators are currently experiencing a resurgence in adoption for data processing pipelines. These devices often consist of custom-designed silicon that trades off general computing capability for increased performance on very specific workloads. The confluence of several factors is driving this increased use. In particular, there is a rapid growth of data collection (a fivefold increase over the next five-year 2020-2025 period according to one report [177]), and business leaders believe that real-time analysis of this data is critical for their success [62, 67]. On the technical front, Dennard Scaling and Moore's law, which describe scaling trends in semiconductor development, either no longer hold or have significantly decreased impact [194]. Consequently, a reinvigorated study of these devices is vital as their need in industry increases.

Accelerators are varied in their design and usage, and types include Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) as well as more esoteric accelerators, such as Google's Tensor Processing Unit (TPU) [117] and Micron's D480 Automata Processor (AP). 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 [187]. In fact, FPGAs are in use in Microsoft datacenters and are also widely available through Amazon's cloud infrastructure [5, 51, 125, 175]. These devices, however, require additional architectural knowledge to effectively program and configure.

Current programming models are akin to assembly-level development on traditional CPU architectures, in which developers must specify their application using minute operations that are device-specific. While these hardware solutions provide high throughputs [95, 181], programming them can be challenging. Consequently, programs written for these accelerators are tedious to develop and challenging to write correctly [57]. Additionally, these low-level representations do not lend themselves well to debugging and maintenance tasks, which are key challenges as it is estimated that developers spend roughly 80–90% of their time on these activities [247]. Put in other words, current programming models lack *sufficient abstraction* from the underlying hardware. Abstraction, as defined by Patterson and Hennessy, refers to hiding low-level details of a system to enable development of complex hardware or software systems to help cope with design complexity [172]. We hypothesize that this lack 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 [208] and frameworks such as Xilinx's SDAccel [248]; however, these models still require low-level knowledge of the underlying architecture to allow for efficient implementation and execution of applications [223, 268].

1.1 APPROACH

To reap the benefits of the performance of hardware accelerators, while enabling higher levels of abstraction and ease of maintenance, 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 (of both the programming model and the hardware) must be sufficiently rich 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.

In this dissertation, we adopt a principled *hardware/software co-design* approach to developing a programming model that meets these requirements [213]. By doing so, we recognize that both the software development process *and* the hardware architectural designs of accelerators are evolving in response to each other.

Consequently, we develop new software tools for supporting hardware accelerators as well as new architectural designs that better support these tools. To do so, we leverage the theoretical findings from *automata theory* [199] to bridge the gap between these two sides. Automata model computation mathematically as transitions between discrete states following a predefined function. The overarching thesis of this work 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 in this dissertation leverages several key 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 [267], network security [184], graph analytics [183], high-energy physics [240], bioinformatics [185, 186, 220], pseudo-random number generation and simulation [232], data-mining [238, 239], and machine learning [221]. Second, finite automata maintain compact state, which admits debugging on accelerators by minimizing and supporting the capture of relevant program state. Third, finite automata can be mapped efficiently to reconfigurable architectures [75, 252], allowing for scalability and performance. Finally, we observe that support for the execution of existing software on hardware accelerators can leverage recent

results from automata theory and software maintenance to construct and execute functionally equivalent automata [10, 60].

1.2 CONTRIBUTIONS

In this dissertation, we introduce new programming models, software maintenance tools, and architectural designs to improve programming support for current and future hardware accelerators. Our contributions include two programming models, a software debugger for accelerator-based applications, and two new hardware accelerator designs for common applications. We briefly describe each in turn.

1.2.1 *Adapting Legacy Code for Execution on Hardware Accelerators*

First, we focus on the task of porting existing, legacy source code for execution on FPGAs and other hardware accelerators. As companies and individuals adopt hardware accelerators into their application workflows, they will need to port existing code to these new devices. Ultimately, we wish to reduce the burden on developers tasked with porting legacy code.

We develop AUTOMATASYNTH, an algorithm for accelerating a particular, relevant class of functions (known as Boolean string kernels) found in extant source code. Our approach uses a novel combination of techniques and approaches from software engineering, machine learning, formal methods, and high-performance automata processing architectures to learn the behavior of a program and construct

a behaviorally equivalent FPGA hardware description. We also formally prove the correctness and termination of AutomataSynth for our target class of functions. For programs that do not meet these criteria, AutomataSynth is able to produce an approximate hardware description.

1.2.2 *High-Level Languages for Automata Processing*

After establishing the feasibility of porting extant code for execution on hardware accelerators, we next focus on supporting development of new applications. We observe that one common technique used in hardware accelerator application design is to quickly scan the data for "interesting" regions (the definition of interesting varies between applications), and return to these regions to perform a more thorough analysis later, reducing the amount of data being processed by a complex algorithm. The initial scan can often be re-phrased as a pattern-searching problem, in which many searches are conducted against a single stream of data. A *pattern* defines a sequence of data that should be identified within another collection of data.

As such, we present RAPID, a new programming model that supports high-level representation of pattern-searching algorithms while maintaining the performance benefits of hardware accelerators. To provide familiar abstractions, we extend a C- or Java-like language with domain-specific parallel control structures to support common pattern-searching paradigms. We also develop compilation algorithms to lower programs written in RAPID to an automata-based representation. The

language supports execution on many hardware platforms, including FPGAs, GPUs, CPUs, and the Micron D480 Automata Processor, and this is achieved through adapting existing automata-based architectures to work with the RAPID language. Further, RAPID programs use 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.

1.2.3 *Interactive Debugging for High-Level Languages and Accelerators*

Next, we design an interactive debugger to help developers maintain code written in high-level languages for hardware accelerators. Debuggers are a vital part of a developer's arsenal of maintenance tools [190]. Although debugging support for CPUs is mature and fully featured (e.g., including standard tools [206], successful technology transfer [24] and annual conferences [1]), the throughput of automata processing applications on CPUs is typically orders of magnitude slower than execution on hardware accelerators [166, 231], making CPUs too slow for effective debugging of automata processing. Unfortunately, current debugging techniques are limited or nonexistent for hardware accelerators. For example, debugging of FPGA designs is typically conducted at extremely low levels of abstraction, such as monitoring individual voltages of hardware elements in the device [21, 128, 219, 246].

We develop a high-throughput, interactive debugger for the RAPID programming language. Our approach bridges the semantic gap between low-level hard-

ware signal inspection available on accelerators and the high levels of abstraction in a programming language using finite automata as an intermediate representation. We focus on the implementation of two key debugging operations: setting breakpoints to stop program execution, and the inspection of program variables [190]. In addition to supporting a traditional notion of breakpoints in RAPID programs, we also introduce a novel breakpoint scheme where breakpoints are set on streams of data. We argue that this new form of breakpoint aids debugging of the class of applications commonly represented in RAPID. To reduce latency in our system (i.e., the time taken between executing a statement or expression in the program and being able to view updated variable values), we also develop a combined hardware accelerator and CPU-software simulation design. 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.

1.2.4 *Architectural Support for Common Applications*

Finally, we develop accelerator architectures to improve the performance of vital applications, such as the parsing of tree-structured and recursively nested data as well as the detection of security policy violations [38, 135]. To enable accelerated parsing of data (e.g., data stored in common text-based serialization formats such as XML and JSON), we develop ASPEN, an Accelerated in-SRAM Pushdown ENgine that implements a more expressive computational abstraction than

previous automata processors. For monitoring of security policies, we develop MARTINI, a simplified automata processing architecture designed to store a Memory Address Representation To INfer Intrusions. Both leverage recent advances in high-performance automata processing architectures and can be implemented by repurposing a portion of the cache subsystem in a modern processor. By doing so, we leverage existing, suitable hardware resources to minimize latency in data processing pipelines and to gain access to internal CPU state.

To support parsing of data, we observe that a computational model more expressive than finite automata—notably deterministic pushdown automata (DPDA)—is necessary [199]. We thus develop a novel, five-stage architecture for execution of DPDA. To support direct adaptation of a large class of legacy parsing applications, we design a compiler that supports existing grammars used to define many common languages and introduce two key optimizations for improving the runtime of parsers on ASPEN.

We restrict the expressive power in MARTINI to minimize hardware resources while providing support for detection of security violations. MARTINI monitors sequences of abstracted memory accesses to validate system behavior. This approach supports the rapid detection of a variety of low-level anomalous behaviors and attacks not otherwise easily discernible at the software level. In particular, our architecture is capable of detecting attacks that exploit internal CPU state, such as Spectre and Meltdown vulnerabilities in Intel processors, discovered in 2018 and 2019 [130, 142].

1.3 METHODOLOGY

In this dissertation, we develop programming models and maintenance tools primarily suited for *logic-based* or *spatial-reconfigurable* accelerators, such as FPGAs and the Micron D480 AP. Additionally, we develop new logic-based architectures to support these models. In many cases, our models may also be used with more traditional von Neumann architectures, such as CPUs and GPUs.

Our evaluation focuses primarily on measuring the extent to which our languages, tools, and architectures satisfy the criteria for successful programming models: performance and scalability, ease of use, expressive power, and legacy support (as described in Section 1.1). As our contributions and criteria are varied, we employ a variety of evaluation approaches, including simulations, empirical studies of real hardware and software, human subjects studies, and formal proofs. In particular, we strive to align our individual methodologies with the metrics of interest.

In general, we evaluate our prototypes using real-world applications. Whenever possible, we strive to use existing benchmark suites and previously published implementations, thereby admitting direct comparison with previous results. For cases where no such benchmarks exist, we develop rigorous protocols for selecting benchmark applications, often based on the mining of open-source repositories of source code (e.g., GitHub).

1.4 SUMMARY AND ORGANIZATION

To summarize, this dissertation makes the following contributions:

1. AutomataSynth, an automata synthesis system for porting legacy source code to execute on hardware accelerators by learning functionally equivalent automata.

2. A high-level programming language, RAPID, for accelerating sequential pattern matching applications on hardware accelerators.

3. A high-throughput, interactive debugging system for RAPID programs for maintenance tasks on FPGAs and the Micron D480 AP.

4. An in-cache accelerator and associated optimizing compilation algorithms for execution of deterministic pushdown automata, such as those used for parsing of serialized data formats.

5. An in-cache accelerator for monitoring memory accesses to detect security policy violations, including many attacks not easily discernible at the software level.

The remainder of this dissertation is organized in the following manner. In Chapter 2, we provide relevant background material on the formalisms and techniques used in the remainder of this dissertation, including finite automata models, common automata-based accelerator designs, programming models, and maintenance tools. Chapter 3 introduces our algorithm for porting legacy code

to hardware accelerators. In Chapter 4 we develop RAPID, a new programming model for representing pattern-searching algorithms, and then develop a high-throughput, interactive debugger for the language in Chapter 5. Next, we develop new architectural designs to support the execution of deterministic pushdown automata as well as detect security policy violations in Chapter 6. Finally, in Chapter 7 we summarize our results and lay out proposed directions of future exploration of related and emerging research challenges.

CHAPTER 2

Background

Prior to commencing our exploration of improving programming support for hardware accelerators, we introduce key concepts and formalisms used heavily throughout the remainder of the chapters. First, we formally define finite automata, which we will use as an abstraction and intermediate representation of computation (Section 2.1). Next, we describe common architectural approaches for accelerating automata computation (Section 2.2). Then, we introduce several extant programming models related to our efforts (Section 2.3) and describe debugging (a typical software maintenance task) with a particular emphasis on hardware accelerators (Section 2.4). Finally, we conclude our presentation of background material with a discussion of two target application areas (Section 2.5).

2.1 FINITE AUTOMATA

We employ several automata-based models of computation to support performant execution of code on hardware accelerators. In this subsection, we describe these

models, introduce notation used in this dissertation, and summarize relevant properties of each model. Readers are encouraged to refer to a theory of computation reference (e.g., Sipser [199]) for a more thorough handling of these computational models.

2.1.1 *Deterministic and Non-Deterministic 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.

A DFA processes input data through the repeated application of the transition function with each subsequent symbol in the input string. After the application of the transition function, a single state within the DFA becomes active. If an accepting state is active after all input characters have been processed, the DFA *accepts* the input (i.e., the input matches the pattern encoded by the DFA).

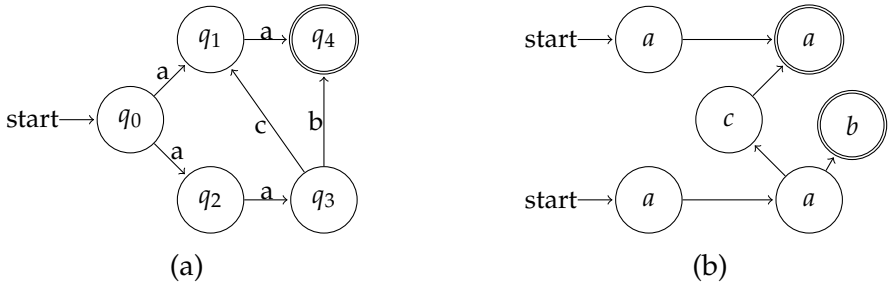


Figure 2.1: A behaviorally equivalent NFA and homogeneous NFA (both accept exactly `aa`, `aab`, and `aaca`). Note that there is a singleton start state in (a) (i.e., $Q_{start} = \{q_0\}$), but there are two start states in (b).

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 $\delta : 2^Q \times \Sigma \to 2^Q$ is the transition function.[1] Note that non-determinism in terms of finite state machines does not refer to stochastic non-determinism, but rather refers to the transition function, which given a set of active states and symbol, returns a new set of active states. This allows for multiple transitions to occur for every symbol processed, effectively forming a tree of computation. NFAs have the same representative power as DFAs but have the advantage of being more spatially compact [199].

In this dissertation, we use an alternate form of DFAs and NFAs known as *homogeneous* DFAs and NFAs. These automata restrict the possible transition rules such that all incoming transitions to a state must occur on the same symbol. 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

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.

states and labels as *state transition elements* (STEs), following the nomenclature adopted by Dlugosch et al. [75]. An STE accepts the symbols in its label, which we refer to as the *character class* of the STE. Figure 2.1 depicts an NFA and a behaviorally equivalent homogeneous NFA.

Additionally, we relax the definition of machine acceptance. Instead of accepting if an accepting state is active at the end of input, whenever an accepting state is active, we *report* the relative offset in the input stream. This allows for pattern-recognition in streams of data symbols by supporting multiple matches in a sequence of input data.

2.1.2 *Deterministic Pushdown Automata*

Pushdown automata (PDAs) extend basic finite automata by including a stack memory structure. A PDA is represented by a 6-tuple, $(Q, \Sigma, \Gamma, \delta, S, F)$, where Γ is the finite alphabet of the stack, which need *not* be the same as the input symbol alphabet. The transition function, δ, is extended to consider stack operations. The transition function for a PDA considers the current state, the input symbol, and the top of the stack and returns a new state along with a stack operation (one of: push a specified symbol, pop the top of the stack, or no operation). Note that PDA are, by definition, non-deterministic, meaning that multiple transitions while processing a single input character. Consequently, the state of the stack memory

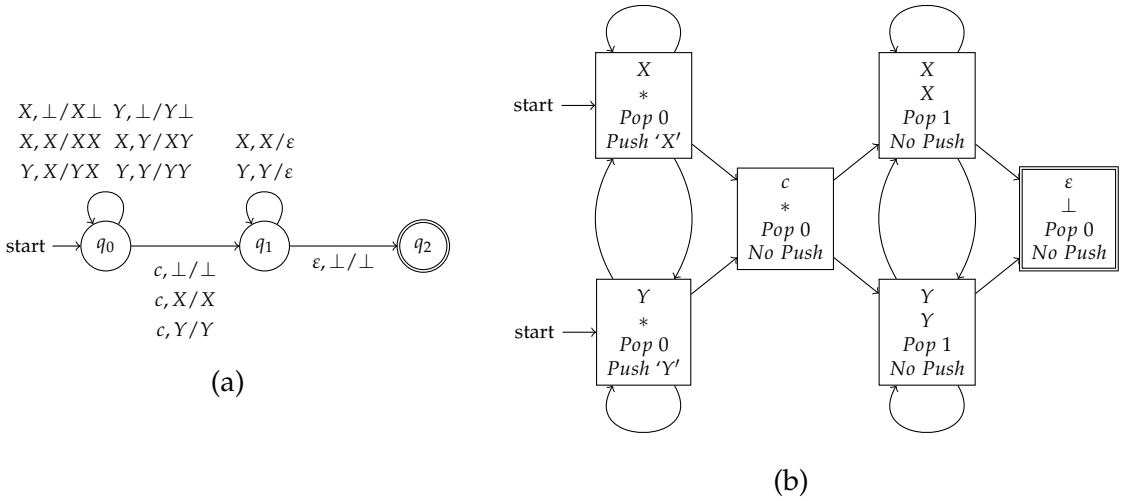


Figure 2.2: A behaviorally equivalent DPDA (a) and hDPDA (b) for recognizing odd-length palindromes with a given center character. For simplicity, we consider strings formed from $\Sigma = \{X, Y\}$ with center character c. Transition rules for the DPDA (a) are written as "$a, b/c$", where a is the matched input symbol, b is the matched stack symbol, and c is the top of the stack after a push or ε for a pop. Note that $\perp$ is a special symbol to represent the bottom of the stack. The hDPDA (b) lists (in order) the input symbol match (ε for no match), stack symbol match ($*$ is a wildcard match), number of symbols to pop, and symbol to push.

may *diverge*. That is, the transition function induces a tree of computation in which each branching point creates a duplicate copy of the stack memory.

In Chapter 6, we restrict attention to *deterministic pushdown automata* (DPDAs), which limit the transition function to only allow a single transition for any valid configuration of the DPDA and an input symbol. This restriction prevents stack divergence, a property we leverage for efficient implementation in hardware. Some transitions perform stack operations without considering the next input symbol, and we refer to these transitions as *epsilon-* or *ε-transitions*. To maintain

determinism, all ε-transitions take place before transitions considering the next input symbol.

Unlike basic finite automata, where non-deterministic and deterministic machines have the same representative power (any NFA has an equivalent DFA and vice versa), DPDAs are strictly *weaker* than PDAs [199, Theorems 2.52 and 2.57]. DPDAs, however, are still powerful enough to parse most programming languages and serialization formats (as described in Chapter 6) as well as other common tasks, such as mining for frequent subtrees within a dataset [17].

For hardware efficiency, we extend the definition of homogeneous finite automata to DPDAs. In a *homogeneous DPDA* (hDPDA), all transitions to a state occur on the same input character, stack comparison, and stack operation. Concretely, the *homogeneous property for DPDAs* states that for any $q, q', p, p' \in Q$, $\sigma, \sigma' \in \Sigma$, $\gamma, \gamma' \in \Gamma$, and op, op' that are operations on the stack, if $\delta(q, \sigma, \gamma) = (p, op)$ and $\delta(q', \sigma', \gamma') = (p', op')$, then

$$p = p' \Rightarrow \left(\sigma = \sigma' \wedge \gamma = \gamma' \wedge op = op'\right). \tag{2.1}$$

This restriction on the transitions function does not limit computational power, but may increase the number of states needed to represent a particular computation. It is possible to characterize this increase as follows.

Claim 1. *Given any DPDA $A = (Q, \Sigma, \Gamma, \delta, S, F)$, the number of states in an equivalent hDPDA is bounded by $O(|\Sigma||Q|^2)$.*

Proof. We consider the worst case: A is fully connected with $|\Sigma| \cdot |Q|$ incident edges to each state and each of these incoming edges performs a different set of input/stack matches and stack operations. Therefore, we must duplicate each node $|\Sigma|(|Q| - 1)$ times to ensure the homogeneity property as defined in Equation (2.1). For any node $q \in Q$, we add $|\Sigma| \cdot |Q|$ copies of q to the equivalent hDPDA, one node for each of the different input/stack operations on incident edges. Therefore, there are at most $|\Sigma| \cdot |Q| \cdot |Q| = |\Sigma||Q|^2$ vertices in the equivalent hDPDA. □

In practice, DPDAs tend to have a fixed alphabet (e.g., ASCII) and are not fully connected, resulting in less than quadratic growth. Even in the worst case, hDPDAs do not significantly increase the number of states (cf. the exponential NFA to DFA transformation[199, Theorem 1.39]). Figure 2.2 provides an example of equivalent DPDA and hDPDA for odd-length palindromes with a known middle character.

2.2 ACCELERATING AUTOMATA PROCESSING

As improvements in semiconductor technology have slowed while demand for increased throughput for complex algorithms remains, there is a trend in hardware design toward specialized accelerator architectures [176, 195]. For example, the use of GPUs and Field-Programmable Gate Arrays (FPGAs) to accelerate general-purpose computation has become commonplace [187]. A recent body of work studies the acceleration of finite automata (NFA and DFA) processing across

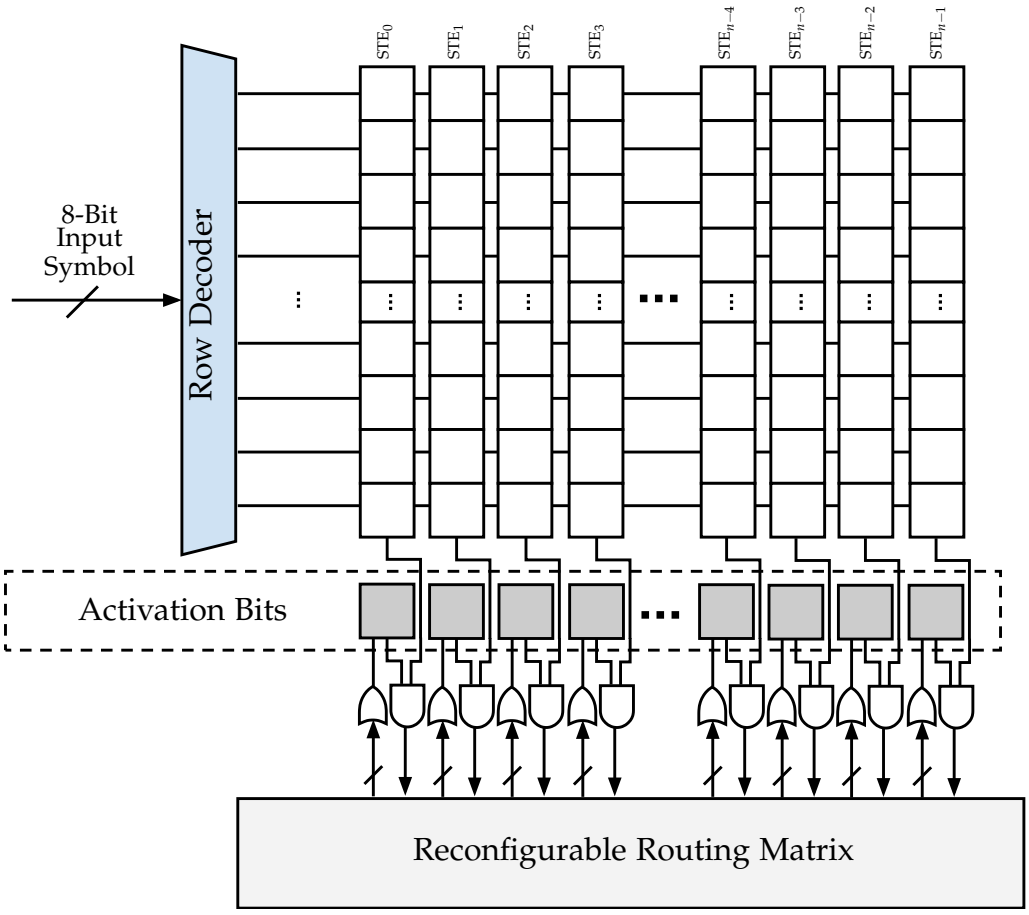


Figure 2.3: Overview of AP architecture. STEs are stored in a memory array, and edges are encoded in a reconfigurable routing matrix.

multiple architectures. Becchi et al. have developed a set of tools and algorithms for efficient CPU-based automata processing [27]. Several regular-expression-matching and DFA-processing ASIC designs have also been proposed [79, 91, 146, 212]. Some (e.g., [80]) incorporate regular expression matching into an extract-transform-load pipeline, supporting a richer set of applications. In this work, we focus on three architectures that accelerate *automata processing* applications:

the Micron D480 AP, commodity FPGAs, and the Cache Automaton. While we focus on these particular architectures, automata processing engines have been developed for other hardware platforms, including CPUs and GPUs [12, 111]. Readers may refer to Appendix A for more details.

2.2.1 *Micron's D480 AP*

The AP is a hierarchical, memory-derived architecture for direct execution of homogeneous non-deterministic finite automata developed by Dlugosch et al. at Micron Technology (a large producer of computer memory) [75]. A homogeneous state (and its transition symbols) is referred to as a *State Transition Element (STE)*. The processing core of the AP consists of a DRAM array and a hierarchical, reconfigurable routing matrix, representing the STEs and edges respectively. The architecture is depicted in Figure 2.3.

A single column of the memory array is used to represent an STE. For the NFA given in Figure 2.1, six columns of memory are needed. The transition symbol(s) of an STE are encoded in the rows of the memory array; each row represents a different symbol in the alphabet. At runtime, a decoded input symbol drives a single row in the DRAM, and the architecture simultaneously computes the subset of STEs that match the input. STEs that match and are active (determined by an additional activation bit stored with each column) generate an output signal that passes through the reconfigurable routing matrix to set the activation bits of downstream STEs.

The ability to record locations in input data where patterns are matched is supported by connecting accepting STEs to special *reporting elements*. When an accepting STE activates, the reporting element generates a *report*, which encodes information about which STE generated the signal and the offset in the data stream where the report was made (as defined in Section 2.1.1). Reports are collected on the AP through a series of buffers and caches before being copied back to the host system (i.e., the AP supports an offload model similar to GPU programming). Because the AP allows for the execution of many NFAs in parallel and because a single NFA may contain multiple reporting STEs, Dlugosch et al. extend the definition of an NFA to contain a *labeling function* that maps nodes to unique labels. We represent labeled NFAs as $(Q, \Sigma, \delta, S, F, id)$, where id is the labeling function. In Chapter 5, we leverage this mapping information to lift hardware runtime state to the semantics of the user-level program.

In addition to STEs, there may be additional special purpose elements. For example, the current generation AP contains saturating counters and combinational logic. These elements connect to the STEs via transition edges and allow for aggregation and thresholding of transitions between STEs. While these elements do not necessarily add any expressive power over traditional NFAs, the use of counters and logic often reduces the overall size of automata. This allows the AP architecture to be flexible. Future implementations might contain additional special purpose elements. In Chapter 4, we use these elements to aid in generating efficient automata from a high-level programming language.

2.2.2 *Cache Automaton*

In addition to the DRAM-based AP, a recent research effort by Subramaniyan et al. developed an SRAM-based automata processor, known as the *Cache Automaton* (CA) [209]. The CA repurposes SRAM-based memory arrays typically found in the *Last Level Cache* (LLC) [81] of modern CPUs to perform automata computations. The theoretical underpinnings of the design closely mirror those of the AP: STEs are stored in arrays and are connected with a reconfigurable routing matrix (see Figure 2.3). Subramaniyan et al. note two primary advantages of a cache-based design. First, SRAM allows for higher clock frequencies than DRAM (i.e., one iteration of state updating can be made faster in a cache). Second, caches are integrated on a processor die, which typically implies improved overall performance due to improved and optimized logic and interconnect performance. However, the disadvantage of embedding the CA in the cache of a processor is *capacity*: the CA supports approximately an order of magnitude fewer STEs than the AP in practice.

Subramaniyan et al. introduced several novel components to improve performance and space efficiency and support the execution of automata in cache. These include: (1) a sense-amplifier[2] cycling technique to more quickly read all columns of data in an SRAM array, (2) an eight-transistor-based SRAM array-based implementation for a compact and programmable switching architecture for the routing

2 A sense-amplifier is the electrical component in a memory system responsible for detecting the bits being read out of a memory array and converting the voltages to match the rest of the circuit [97].

matrix,[3] and (3) a hierarchical switch and wire-routing topology leveraging the existing design of LLC cache ways[4] to support tens of thousands of states in an NFA.

In Chapter 6, we leverage many of the optimizations proposed by Subramaniyan et al. in the development of two automata-derived architectures.

2.2.3 *Field-Programmable Gate Arrays*

Field-Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits containing both combinational logic (e.g., logical gates) and memory [172]. These sub-units are typically laid out in a regular, repeated pattern, often referred to as a *fabric*, of reconfigurable 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 matrix. This architectural model allows FPGAs to dynamically form arbitrary circuits, which can be useful for prototyping logic circuits.

FPGAs have been in use for decades and were introduced as an improvement over earlier programmable devices, such as *programmable array logic* (PAL), *programmable logic devices* (PLDs), *erasable programmable logic devices* (EPLDs) [25]. Their use has been varied over the years, including rapid system prototyping,

3 SRAM arrays traditionally use six transistors to store one bit [198]. Eight transistors may be used to increase redundancy and improve signals, and this property enables the reuse as a switching architecture.

4 Last Level Caches are typically hierarchically subdivided into manageable units. The first level of subdivisions are referred to as *slices* [81]. Slices are further subdivided into *ways*.

communications processors, digital signal processing, industrial control systems, wearables, fully custom computing machines, and dynamically reconfigurable systems [180]. There have also been efforts to combine traditional CPUs with FPGAs, which can take many forms. For example, some FPGAs include processing units as discrete components in the fabric (i.e., *hard cores*) [228]. More common are *soft cores*, which implement a traditional microprocessor using combinational logic and memory (e.g., Xilinx MicroBlaze [154], ARM Cortex-M1 [64], and the Oracle OpenSPARC T1 [167]). Finally, there is an effort to manufacture hybrid devices that combine CPUs and FPGAs into a single package [107].

Prior work has investigated implementing finite automata processing on FPGAs [28, 119, 196, 204, 241, 256]. Because automata can be thought of as circuits—where each state transition element is a specialized logic gate—they can be naturally implemented in an FPGA fabric. Recently Xie et al. combined these recent advances with optimized input/output circuitry to support integration of high-performance automata processing into data processing pipelines [252]. The aptly named Reconfigurable Engine for Automata Processing (REAPR) configures an FPGA to operate very similarly to the AP-style processing model (see Section 2.2). LUTs are used in place of columns of memory to determine input symbol matches each clock cycle (logically, one LUT is assigned to each STE). Flip-flops are then used to store the activation bits for STEs, and transition signals are propagated through the FPGA's reconfigurable routing matrix.

Although FPGAs are a natural and successful fit for acceleration of automata processing and have been the subject of significant study [28, 119, 252], the ability

to port software to FPGAs while maintaining performance remains an open research problem [268]. In this dissertation, we leverage the recent advances in automata processing to support porting existing software to FPGAs (Chapter 3) as well as maintaining the performance of design choices across architectures (Chapter 4).

2.3 PROGRAMMING MODELS

Next, we the consider current landscape of relevant programming models. We begin by introducing models most relevant for automata processing and hardware-accelerator-based computation. We also briefly discuss programming models that we use as springboards for porting legacy code to accelerators in Chapter 3.

2.3.1 *Automata Representations and Regular Expressions*

The most direct programming model for automata processing is to develop DFAs and NFAs. Traditionally, NFAs are often represented as a directed graph with states as vertices and the transition function encoded as edges. While capable of specifying search patterns, NFAs are difficult to write and maintain. Text-based NFA formats, such as the XML-based Automata Network Markup Language (ANML) and Becchi's transition table representation [27], are extremely verbose. For example, a toy example for measuring the pairwise difference of characters between an input string and a fixed five-character string requires 62 lines of

ANML to represent [155]. Maintenance tasks on this code are also cumbersome: changing such an automaton to compare against a string of length 12 requires modification of 65% of the code. NFAs can be challenging and tedious to write correctly, especially for developers lacking familiarity with automata theory. In research areas such as program verification, the task of specifying automata is often automated [7].

Regular expressions are another common option for representing a search pattern as matched by an NFA or DFA and are defined recursively by Sipser using the following rules [199]:

1. a for some a in the alphabet Σ (i.e., match a single character),
2. ε (i.e., match the empty string),
3. $\emptyset$ (i.e., match nothing),
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions (i.e., match the *union* of R_1 and R_2),
5. $(R_1 R_2)$, where R_1 and R_2 are regular expressions (i.e., match the *concatenation* of R_1 and R_2), and
6. (R_1^*), where R_1 is a regular expression (i.e., match the *Kleene closure*, which is zero or more occurrences of R_1).

We will use this definition in Chapter 4 to demonstrate the expressive power of the RAPID language. Regular expressions suffer from similar maintainability challenges as explicit automata representations. For many of our target applications,

such as motif searches, particle tracking, and rule mining, the regular expression representing the search is non-intuitive and may simply be an exhaustive enumeration of all possible strings that should be matched (much in the same way an overfit machine learning classifier might directly encode a lookup table of the training data [131]). Additionally, programming of regular expressions can be extremely error-prone due to variations in regular expression syntax, which leads to high rates of runtime exceptions [205].

Therefore, both of these programming model fail to meet our required ease of use criterion (see Section 1.1).

2.3.2 *Languages for Streaming Applications*

Streaming applications process a sequence of data received in real time. Common examples include radio receivers and software routers. Automata processing can be viewed as a streaming application because input symbols are processed in real time to update the automaton's active states.

Languages for streaming applications have been studied in great detail. StreamIt [215], an exemplar of this class of languages, provides structures for stream pipelining, splitting and joining, and feedback loops. StreamIt objects may peek and pop from the input stream, store input, and perform computations before outputting a result. Automata processing, however, does not readily admit this computational model. Finite automata have no inherent memory (see Section 2.1.1) and cannot generally peek at the input stream. Many of the operations allowed by StreamIt

are thus not applicable in our domain, and it is not evident how to extend the StreamIt model to describe complex automata nor non-deterministic execution. Ultimately, StreamIt targets a different computational abstraction, and is not directly applicable.

2.3.3 *Non-Deterministic Languages*

Non-determinism is a useful formalism for identifying patterns in parallel within a data stream. In a state machine, non-determinism arises when multiple states are active simultaneously, allowing for parallel exploration of input data. Several existing languages contain non-deterministic control structures to facilitate these types of operations.

Dijkstra's Guarded Command Language [74] introduces non-deterministic alternative and repetitive constructs. These constructs are predicated with a Boolean *guard* that must be true for the encapsulated statements to execute. The alternative construct chooses arbitrarily one command with a satisfied guard and executes it. In the repetitive construct, the program loops, choosing one command with a satisfied guard to execute, until no guards are satisfied. Rather than proposing a concrete syntax, the Guarded Command Language presents guiding formalisms for supporting non-determinism. We develop a programming model that provides similar constructs in Chapter 4, with a particular focus on identifying patterns in streams of data.

An additional non-deterministic programming language is Alma-0 [19], a declarative extension of Modula-2. Alma-0 supports the use of Boolean expressions as statements, an `ORELSE` statement allowing for execution of multiple paths through the program, and a `SOME` statement that is the non-deterministic dual of a traditional for-loop. In Alma-0, `ORELSE` and `SOME` are defined via backtracking. Execution is single threaded: when an `ORELSE` statement or a `SOME` statement is encountered, the program will choose a single option to execute. If an exploration fails, the program backtracks to the last choice point, restoring all program state, and attempts a different option. These additional constructs provide natural extensions to traditional languages to support parallel processing of data and thus satisfy our stated requirement for ease of use. In Chapter 4 we introduce a programming model that leverages these constructs for abstractly representing automata computation.

2.3.4 *Programming Models for Portability*

The holy grail of programming for heterogeneous environments is to "write once and run anywhere." Research into portability dates back decades and has its origins in attempts to support multiple mainframe computer architectures. For example, the Parallel Programming Language (PPL) was a strongly typed language that abstracted away from machine-dependent types to support multiple architectural targets [236]. More recently, the focus has been on supporting portability across different coprocessors.

The OpenCL language boasts support for CPUs, GPUs, FPGAs, and other microprocessors [208]. The language provides an abstract notion of computational devices and processing elements, which allow for task- and data-parallel applications to be executed in heterogeneous environments. While the same code can be run on multiple types of hardware, code written for one architecture rarely performs well on another architecture. To execute efficiently on both GPUs and FPGAs, for example, a developer must often write two copies of the application, crafting the code to make use of the particular strengths of each platform. Our goal in this dissertation is to avoid this rewriting step, allowing the developer to write an application using a computational abstraction that performs well *across architectures*.

High-level constructs, such as Map-Reduce, have been demonstrated to provide portability across architectures [102, 258]. We also make use of high-level constructs, but constructs in our language are more specific to sequential pattern identification tasks.

2.3.5 *Languages for Programming FPGAs*

Adopting hardware accelerators into existing application workflows requires porting code to these new programming models. Unfortunately, porting legacy code remains difficult. The primary programming model for FPGAs remains *Hardware Description Languages* (HDLs) such as Verilog and VHDL [173, 216]. HDLs have a level of abstraction akin to assembly-level development on traditional

CPU architectures, allowing for the specification of circuits by specifying logical formulas and their connections to memory and each other. While these hardware solutions provide high throughputs, these languages and their abstractions are not a part of computer science curricula. For example, the Joint Task Force on Computing Curricula at the Association of Computing Machinery (ACM) and IEEE Computer Society recommend in its 2013 curriculum that a computer science degree program includes only three lecture hours of "digital logic and digital systems" [116]. HDLs are listed as a topic, but none of the learning outcomes specify familiarity with HDLs—let alone competency. Therefore, we should not expect software developers entering the workforce to have the necessary skills to port code to HDLs.

High-Level Synthesis (HLS) allows development for FPGAs at a much higher level of abstraction than HDLs [161]. Indeed, HLS has been demonstrated to reduce the time to develop FPGA designs [134]. Most tools support programs written in C-like languages, suggesting that HLS would be amenable for adapting and accelerating legacy code bases. However, the performance of designs constructed using HLS can be unimpressive, requiring significant optimization [223, 268]. HLS tools may also not support all features of the language (e.g., dynamic data structures), meaning that legacy code must be refactored before the approach is applicable.

In Chapter 3, we present an alternative to HLS that decouples the existing design and implementation of legacy code from the final design produced for an FPGA. In doing so, we avoid many of the limitations of HLS techniques.

2.3.6 *State Machine Learning Algorithms*

In this dissertation, we employ state machining learning algorithms as an alternative to HLS. We briefly summarize learning of state machines here, detailing the most relevant instance in Section 3.1.1. These algorithms are a subset of *model learning* in learning theory and have been the subject of study for several decades [9, 207, 225, 269]. The most common approach is to use *active learning* in which the model is learned by performing experiments (tests) on the software or system to be learned. State machine learning has been applied to the domains of internet banking [2], network protocols [72, 82], legacy systems [148, 189], and describing machine learning classifiers [245].

Most efforts have focused on developing suitable algorithms for learning finite automata [10, 41]. More recent advances simplify the internal data structures of the algorithms, reduce the number of tests necessary to learn a model, or combinations thereof [113, 114, 123, 179]. Learning an equivalent state machine from software remains challenging, and most approaches employ some form of approximation [10, 137].

In Chapter 3, we apply this body of model learning research to the problem of adapting legacy source code for efficient execution on hardware accelerators. Our approach attempts to learn a model that is fully equivalent to the original program using software verification techniques but may also produce approximate results in some situations.

2.3.7 *Program Synthesis*

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 [6, 200, 201], programming by example [94], and automated program repair [152, 164]. Many of these approaches employ *counterexample-guided inductive synthesis* (CEGIS) to produce a final solution [201]. 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 begins the next iteration of synthesis. We note that CEGIS is largely equivalent to the techniques used in the learning theory community for model learning.

A related body of research focuses on extracting program behavior from legacy code for acceleration using *domain-specific languages* (DSLs), an approach referred to as *verified lifting*. Examples, include extracting stencil computations [118, 153], database queries [55], and sparse and dense linear algebra calculations [89]. By targeting DSLs, verified lifting can leverage known properties of the given problem domain to aid extraction and acceleration. For generality in this dissertation, we intentionally limit the domain-specific assumptions leveraged by our approach.

2.4 MAINTENANCE TOOLS

Software maintenance tasks are varied and account for a significant proportion of developer effort [174, 191]. In this section, we describe efforts to support and evaluate the common maintenance task of debugging. Further, we introduce the maintenance task of *verification*, which we leverage in a framework for porting legacy code to hardware accelerators.

2.4.1 *Debugging on Hardware Accelerators*

In this 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 [259]. The development of debugging tools has a lengthy history [96, 136, 188, 262], and software debuggers are commonplace in development toolchains [149]. Ungar et al. argue that immediacy is important for debugging tasks and developed a step-through debugger [224]. There has also been significant effort devoted to improving the efficiency of debugging tools, such as quickly transferring control when a breakpoint is reached [124] and efficiently supporting large numbers of watchpoints [265]. These approaches provide debugging support for general purpose processors and languages. The technique presented in this work is in the same spirit: we 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 [203] and energy-harvesting systems [61]. 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 [104]. Similarly, there are approaches for debugging FPGA applications [8, 92]; however, these techniques typically focus on inspection of the underlying hardware description, rather than programs written in high-level languages. Debugging of high-level synthesis (HLS) designs has focused on monitoring trace registers and using record-replay techniques to expose program state for segments of single-threaded applications [90, 159]. Our work further develops the area of debugging for specialized processors by presenting a technique for inspecting source-level program state during program execution on highly parallel automata processing engines.

2.4.2 *Understanding the Importance of Debugging*

Human studies have shed light on debugging and the role of automated tools. Weiser found that programmers inspect "program slices" when debugging, which may not be textually contiguous but follow data and control flow [244]. Ko and Myers demonstrated that their debugging tool, Whyline, allowed study participants to perform debugging tasks more quickly [129]. Fry and Weimer found

that localization accuracy is not uniform across various bug types [86]. Romero et al. found that debugging performance is related to balanced use of available information in programming systems that provide multiple representations of state [182]. Our results in Chapter 5 complement these findings by demonstrating our debugging system improves fault localization accuracy for the domain of pattern-matching automata processing applications.

2.4.3 *Software Verification*

Software maintenance encompasses more than just debugging, including activities such as validating that a program provides pre-specified functionality and meets pre-defined requirements.

Program verifiers and *software model checkers* prove that a program adheres to a specification or produce counterexamples that violate the specification [35]. These tools typically interleave the control flow graph (CFG) and a 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 applications [34, 60, 151]. Of particular relevance here are bounded or iterative techniques that address recursive control flow [37, 115], which typically unroll loops a fixed number of times before checking if an error state is reachable in the straight-line portion of the CFG. Most closely related to our work has been the use of bounded model checking to

verify string-processing web applications; however, this work often focused on secure information flow rather than constraints over strings [106]. There are also theoretical results on the decidability of straight-line programs on strings, which naturally arise in bounded model checking [140].

These techniques have been employed to verify operating system drivers [24], validating communication protocols [82], and finding bugs in concurrent data structures [169], among others. In Chapter 3, we combine software verification with model learning techniques to port legacy code to FPGAs.

2.5 Applications Benefiting from Acceleration

The principle of hardware/software co-design suggests that choices made when designing hardware should be influenced by the target applications. In this section, we introduce two application domains, data parsing and computer security, that we use as motivation for proposing new, automata-based accelerator architectures in Chapter 6.

2.5.1 *Parsing of XML Files*

As data continues to be collected and processed at ever-increasing rates, it is paramount that software be able to efficiently read stored data. Such data is often stored in structured text files, often formatted in *Extensible Markup Language* (XML) or *JavaScript Object Notation* (JSON) [77, 98].

Parsing, or syntactic analysis, is the process of validating and reconstructing tree (nested) data structures from a sequence of input *tokens* [4]. In natural language, this process relates to validating that a sequence of words forms a valid sentence structure, and for a programming language, a parser will verify that a statement has the correct form (e.g., a conditional in C contains the correct keywords, expressions, and statements in the correct order). In this dissertation, we focus primarily on the task of parsing XML files, which is common to many applications [135, 145]. Parsing XML produces a special tree data structure called the *Document Object Model* (DOM) [98].

Parsers are typically implemented as the second stage of a larger pipeline [190]. In the first stage, a *lexer*, tokenizer, or scanner reads raw data and produces a list of tokens (i.e., a lexer converts a stream of characters into a stream of words), which are passed to the parser. The parser produces a tree from these input tokens, which can be further validated and processed by later pipeline stages. For example, an XML parser will validate that tags are properly nested, but a later stage in the pipeline performs semantic checks, such as verifying that text in opening and closing tags match.

Conventional software-based parsers exhibit complex input-dependent data and control flow patterns resulting in poor performance when executed on CPUs. Figure 2.4 (b) shows two state-of-the-art open-source XML parsers, Expat [58] and Xerces [18], which can require approximately 6–25 branch instructions to process a single byte of input depending on the *markup density* of the input XML file (i.e., ratio of syntactic markup to document size). These overheads result from nested

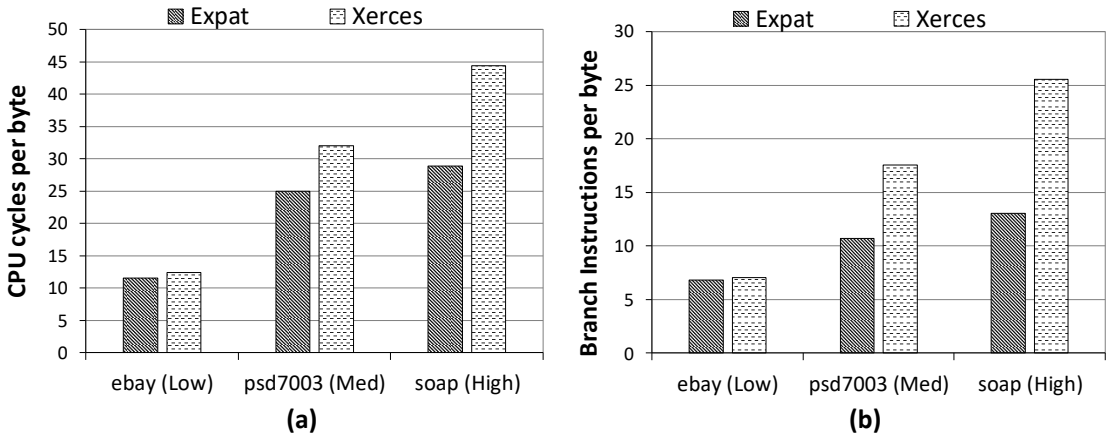


Figure 2.4: Conventional parser performance. (a) CPU cycles per byte. (b) Branch instructions per byte

switch-case statements that determine the next parsing state. Furthermore, as the parser alternates between markup processing and processing of variable-length content (e.g., free-form strings), there is little data reuse, leading to high cache miss rates (approximately 100 L1 caches misses per kB for Xerces). As a result of both high branch misprediction and cache miss rates, software parsers take about 12–45 CPU cycles to parse a single input byte as depicted in Figure 2.4 (a). In Chapter 6, we demonstrate that is possible to avoid these overheads by applying principles from in-memory automata processing architectures (Section 2.2) to develop an architecture that supports DPDA computation (Section 2.1.2).

2.5.2 *Architectural Side-Channel Attacks*

The security of software and hardware systems is the subject of much study [38]. *Side-channel* attacks are a particularly insidious form of security vulnerabilities

in a system. Side-channel attacks steal information from a system indirectly by monitoring properties of the system, such as power dissipation, high-frequency sounds, or timing to infer program state and data [44, 59, 88]. Of particular interest in recent years have been so-called *architectural side-channel attacks*, which leverage properties of—or design flaws in—commodity computational hardware [242].

Architectural side-channel attacks, such as Spectre [130] and Meltdown [142], use cache timing to leak information in memory. Such attacks can exploit side effects of branch prediction and speculative execution to read or affect arbitrary memory locations. The key problem is that changes to the state of the cache persist even if the CPU discards instructions that are speculatively executed. As a result, a malicious program can execute a controlled sequence of memory accesses and then leverage its knowledge of the cache structure to read arbitrary locations in memory that are cached by other programs. In addition to Spectre and Meltdown, other cache side-channel attacks include: Foreshadow [227], Flush+Reload [257], Evict+Time [168], Prime+Probe [143], and Nailgun [165]. Since this class of attacks relies on hardware vulnerabilities, they are OS-independent and challenging to patch efficiently in software.

To defend against Spectre and Meltdown, CPU speculation features can be disabled, but the performance impact is high [150, 237] and doing so does not address other extant hardware vulnerabilities. Further, novel microarchitecture redesigns are non-trivial and costly in terms of time and resources and may expose new—or overlook existing—hardware vulnerabilities.

In Chapter 6, we develop a custom processor unit for accelerating the detection of side-channel attacks, providing additional system security with minimal overhead.

2.5.3 *Runtime Intrusion Detection Systems*

Broadly, an *intrusion detection system* (IDS) is tasked with classifying a sequence of inputs as being normal or anomalous according to some detection technique [139]. Anomaly-based approaches have the advantage of being able to detect *zero-day attacks* (i.e., previously unreleased or undocumented attacks), and they can be customized to particular operating environments, IDS efforts most relevant to this dissertation use *n*-grams of system calls to detect misbehaving Unix processes [84, 202]. *Systems calls* are special functions that allow a program to interact with the operating system [45]. Forrest et al. found that modeling sequences of these calls could accurately detect software-level attacks. Unfortunately, system calls do no capture the low-level behavior exploited by many hardware side-channel attacks (see Section 2.5.2), and can therefore not be directly applied our use case in this dissertation.

A *hardware-based malware detector* (HMD) monitors micro-architectural traces and raises alerts about anomalous behavior (e.g., [122]). HMDs can detect side-channel attacks that leave no system call traces and can potentially be secured against a compromised OS [263]. For example, Demme et al. used performance counters as the data source for an HMD [73], though there are concerns about using

performance counters in this domain [69]. Similarly, Wei et al. proposed a power anomaly detection system for embedded systems which can detect side-channel attacks, including Spectre, with high accuracy [243]. However, this method targets embedded systems that run fixed jobs with consistent behavior.

In Chapter 6, we combine the concepts of n-gram-based monitoring with HMDs to detect attacks (including Spectre and Meltdown) with minimal system overhead. We design a microarchitecture that monitors n-grams of memory access sequences, which we model as finite automata.

2.6 CHAPTER SUMMARY

We introduce several key concepts and describe how they apply to the work presented in this dissertation. We describe several theoretical models from automata theory. Then, we consider several architectural approaches to accelerating their execution. Next, we explore various programming models and describe their relationship with automata-based computation. We introduce concepts from various software maintenance tools, which we both implement and leverage in this dissertation. Finally, we describe two application domains (security and data parsing) that we use as case studies motivating further architectural development.

In the next chapter, we introduce a new programming model to help developers port existing code to run on hardware accelerators.

CHAPTER 3

Acceleration of Legacy String Functions

As hardware accelerators begin to be adopted in industry it will be necessary to port extant software to these new platforms. In Section 2.3.5, we described HLS, the state-of-the-art approach for porting legacy code to FPGAs, which unfortunately lacks support for some language features, and typically still requires significant refactoring to produce performant FPGA code.

In this chapter, we present AutomataSynth,[1] a new approach for executing code (including legacy programs and automata computations) on FPGAs and other hardware accelerators. Unlike HLS, which statically analyzes a program to produce a hardware design, AutomataSynth both dynamically observes and statically analyzes program behavior to synthesize a *functionally equivalent* hardware design. Our approach is based on several key insights. First, state machines provide an abstraction that has successfully accelerated applications across many domains [183–186, 220, 221, 232, 238, 240, 267] and admit efficient implementations in hardware [75, 146, 252], but typically require rewriting applications. Second,

1 https://github.com/kevinaangstadt/automata-synth

there is an entire body of work on query-based learning of state machines (e.g., see Angluin for a classic survey of computational learning theory [11]), but these algorithms commonly rely on unrealistic oracle assumptions. Third, we observe that the combination of software model checking (e.g., [33, 35]) and recent advances in string decision procedures (e.g., [71, 120, 217, 222]) can be used in place of oracles for certain classes of legacy code kernels, such as those that recognize regular languages.

While AutomataSynth is based on a general approach for synthesizing hardware designs from high-level source code, we focus in this chapter specifically on synthesizing Boolean string kernels (functions that return true or false given a string). We accelerate these string kernels using automata processing, which requires representing functions as finite automata [75, 146, 252]. We demonstrate how software model checking, using a novel combination of bounded model checking with incremental loop unrolling augmented with string decision procedures, can answer oracle queries required by Angluin-style learning algorithms, resulting in a framework to iteratively infer automata corresponding to legacy kernels.

We focus our evaluation of AutomataSynth on *scalability* and *legacy support*. As such, we develop a benchmark suite of string kernels mined from public repositories on GitHub and measure the correctness of the automata generated by AutomataSynth as well as the time required to synthesize and size of the resulting automata. Our evaluation demonstrates that our approach is viable for small functions and exposes new opportunities for improving current-generation tools.

We identify four key challenges associated with using state-of-the-art methods to compile legacy kernels to FPGAs and suggest paths forward for addressing current limitations.

In summary, we present the following scientific contributions in this chapter:

- AutomataSynth, a framework for accelerating legacy string kernels by learning equivalent state machines. We extend an Angluin-style learning algorithm to use a combination of iterative bounded software model checking and string decision procedures to answer oracle queries.

- A proof that AutomataSynth terminates and is correct (i.e., relatively complete with respect to the underlying model checker) for kernels that recognize regular languages. The proof leverages the minimality of machines learned by L* and the Pumping Lemma for regular languages.

- An empirical evaluation of AutomataSynth on 18 indicative kernels mined from public GitHub repositories. We learn 13 exactly equivalent models and 2 near approximations.

In the remainder of this chapter, we introduce our formulation of the state machining learning problem in Section 3.1. Then we detail a composition of formal methods and software verification techniques for solving this problem and prove, formally, the correctness of our approach in Section 3.2. Next, we describe our experimental methodology in Section 3.3 present an empirical evaluation of AutomataSynth in Section 3.4 before finally concluding with a discussion of open challenges for learning-based approaches in Section 3.5.

3.1 LEARNING STATE MACHINES FROM LEGACY CODE

We present AUTOMATASYNTH, a framework for learning functional behavior models for off-the-shelf, legacy code implementing regular languages and synthesizing hardware descriptions from those models. Our approach extends Angluin's L* algorithm [10] by (1) using bounded software model checking with incremental unrolling to implement one of its assumptions, (2) using software testing to implement another of its assumptions, and (3) transforming learned models into homogeneous DFAs for hardware synthesis.

3.1.1 *L* Primer*

Dana Angluin's foundational L* algorithm was popularized in 1987 [10]. Because many of our framework decisions (such as how to implement its required queries and counterexamples in a legacy source code context) and results (such as correctness and termination arguments) depend on the steps and invariants of her algorithm, we sketch it here in some detail. We claim no novelty in this subsection and readers familiar with L* can proceed to Section 3.1.2.

At its core, the L* algorithm relies on a *minimally adequate teacher* (MAT) to answer two kinds of queries about a held-out language, L. First, the MAT must answer *membership* queries, yielding a Boolean value indicating if the queried string is a member of L. Second, the MAT must answer *conjecture* or *termination*

queries.[2] Given a candidate regular language A, typically expressed as a finite state machine, the MAT responds with `true` if $A = L$ or responds with a *counterexample* string for which A and L differ. (Note that automata learning is used in applications where L is not a DFA, and thus this query is typically not resolved by standard DFA equivalence checking.)

These queries are used to construct an *observation table* that can be transformed directly into a DFA. This table may be defined as a 3-tuple, (S, E, T), where S is a nonempty, finite, prefix-closed[3] set of strings over Σ; E is a nonempty, finite, suffix-closed set of strings over Σ; and T is a function mapping $((S \cup S \cdot \Sigma) \cdot E)$ to $\{\texttt{true}, \texttt{false}\}$. (S, E, T) may be visualized as a two-dimensional array where rows are indexed by a value $s \in S \cdot \Sigma$, columns are indexed by a value $e \in E$, and entries are equal to $T(s \cdot e)$. For ease of notation, Angluin defines $row(s)$ to be a finite function, f, mapping values from E to $\{\texttt{true}, \texttt{false}\}$ defined as $f(e) = T(s \cdot e)$. Informally, $row(s)$ denotes the values in a particular row of the observation table.

An observation table must be both *closed* and *consistent* before a DFA may be correctly constructed. A table is *closed* if for every $t \in S \cdot \Sigma$, there exists an $s \in S$ such that $row(t) = row(s)$. A table is *consistent* if, for all $s_1, s_2 \in S$ where $row(s_1) = row(s_2)$, $row(s_1 \cdot a) = row(s_2 \cdot a)$ for all $a \in \Sigma$. These properties ensure that there is a valid transition out of each state in the DFA (closed) and that transitions on any character remain the same regardless of the characters already

2 These are also called equivalence queries, but we avoid this term to prevent confusion with similar uses of the term in software verification.

3 A set is prefix-closed if $\forall s \in S$, every prefix of s is also a member of S. Suffix-closure is defined similarly.

processed (consistent). Given a closed and consistent observation table, a DFA over the alphabet Σ may be constructed as follows:

$$
\begin{aligned}
Q &= \{row(s) \mid s \in S\}, \\
q_0 &= row(\varepsilon), \\
F &= \{row(s) \mid s \in S \wedge T(s) = \texttt{true}\}, \\
\delta(row(s), a) &= row(s \cdot a).
\end{aligned}
$$

Each unique row in the observation table becomes a state in the candidate automaton, and outgoing transitions from a state are defined by the row indexed by the current row's prefix concatenated with the transition character.

Pseudocode for the L* algorithm is provided in Algorithm 3.1. A closed, consistent observation table is constructed using membership queries. Then, the table is transformed into a candidate automaton for a termination query. If the MAT responds with a counterexample, the counterexample and its prefixes are added to the observation table. The process repeats until the MAT responds to a termination query in the affirmative. The final automaton is the learned language.

3.1.2 AutomataSynth *Problem Description*

In this subsection, we formalize the problem of learning a state machine from a legacy string kernel.

Algorithm 3.1: Angluin's L* Learner [10]

Data: MAT for held-out language, L
Result: A DFA, M, representing the held-out language, L
Initialize S and E to $\{\varepsilon\}$;
Ask membership queries for ε and each $a \in \Sigma$;
Construct initial observation table (S, E, T);
repeat
  while (S, E, T) is not closed or not consistent **do**
    if (S, E, T) is not consistent **then**
      Find $s_1, s_2 \in S$, $a \in \Sigma$, $e \in E$ such that
       $row(s_1) = row(s_2) \wedge T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$;
      Add $a \cdot e$ to E;
      Extend T to include the new suffix with membership queries;
    end
    if (S, E, T) is not closed **then**
      Find $s_1 \in S$, $a \in \Sigma$ such that $row(s_1 \cdot a) \neq row(s)$ for all $s \in \Sigma$;
      Add $s_1 \cdot a$ to S;
      Extend T to include the new prefix with membership queries;
    end
  end
  Construct DFA, M from (S, E, T);
  Make termination query with M;
  if MAT responds with counterexample t **then**
    Add t and all prefixes to S;
    Extend T to include the new prefixes using membership queries;
  end
until the MAT responds with `true` to the termination check on M;
return DFA M

AutomataSynth operates on a function that takes one string argument and returns a Boolean value:

```
kernel : string -> bool
```

We assume that the source code for this function is provided and that the function halts and returns a value on all inputs (i.e., `kernel` is an algorithm). If `kernel` recognizes a regular language, AutomataSynth returns a state machine, M, with equivalent behavior to `kernel`. That is, for all $s \in \Sigma^*$, $M(s) =$ `kernel`(s). For runs which exceed a resource budget or expose incompleteness in the underlying

theorem prover (including functions that are non-regular), our prototype implementation alerts and provides *approximate* equivalence, where $M(s) = \texttt{kernel}(s)$ when the length of s is less than an arbitrary fixed length (see Section 3.2).

In Section 3.2.4, we present a formal proof that our framework produces an equivalent DFA for input kernels that recognize regular languages. Our empirical evaluation in Section 3.4 demonstrates that real-world legacy string kernels either recognize regular languages, or our tool can produce an approximation of the original function. We discuss the challenges associated with supporting a broader class of functions in Section 3.5.1.

3.1.3 *Using Source Code as a MAT*

We extend Angluin's L* algorithm to learn a DFA representation of a legacy string kernel. To succeed, we must construct a MAT that can answer membership and termination queries about an input string kernel. While the L* algorithm provides a framework for query-based DFA learning, the original work does not define any one mechanism for implementing the teacher. Our proposed MAT implementation leverages the source code of the target function.

MEMBERSHIP QUERIES. We observe that a membership query for a string, s, may be implemented by executing the legacy kernel on s. The result returned by the function is the answer to the query. For languages akin to C employing integers, we interpret Boolean values in the standard way (i.e., 0 is `false` and all

other values are `true`). While direct and intuitive in theory, we note that there are several challenges in practice. Following the C standard, many runtime systems assume that strings are null-terminated (i.e., a null character must only occur as the final character in a string). In practice, however, we find that legacy string kernels will sometimes allow null characters in other positions. This often occurs when the length of the input string is known *a priori*. While seemingly innocuous, this deviation from the standard limits the runtime mechanisms by which the legacy kernel may be invoked. We found that compiling the kernel to a shared object and then invoking the function dynamically provided the best stability in our experiments.

TERMINATION QUERIES. At the heart of our problem formulation is the challenge that a legacy string kernel does not admit a direct means for answering termination queries. A recent survey of model learning indicates that testing for equivalence queries [225]; however, our initial efforts found testing alone to be unsuitable for termination queries in this domain. Our insight is that verification strategies from software model checking may be used to test for equivalence between the kernel and a candidate automaton. Traditionally in verification, equivalence would be proven using bisimulation or interleaving of the automaton and the source kernel. However, this formulation presupposes that the "state transitions" are directly encoded in the source code and can be aligned with the state transitions in the candidate automaton. We do not make this assumption in our problem definition in Section 3.1.2, and we prefer an approach that does not

require manual annotation. Indeed, we do not even assume that the states of the equivalent automaton are visited "in order" during the execution of the legacy kernel.

We verify an alternate reachability property that places additional constraints on the input string. In particular, we observe that a counterexample $t \in \Sigma^+$ is in either $L(\texttt{kernel})$ or $L(M)$ but not in both, and thus will always satisfy the constraint $t \in L(\texttt{kernel}) \oplus L(M)$, where $\oplus$ is the symmetric difference operator. Therefore, we ask the software verifier to prove that there is no execution of `kernel` on t where `kernel` returns `true` and $t \notin L(M)$ *or* `kernel` returns `false` and $t \in L(M)$. To test this reachability property, we use a novel combination of bounded model checking with incremental loop unrolling augmented with a string constraint solver. We discuss the implementation of this verification task in depth in Section 3.2.

Software verifiers are *relatively complete* (e.g., [22]), meaning that there are certain programs that cannot be fully verified due to limitations in the underlying SMT solvers. Verifiers often return an answer in three-valued logic: `true` in our application means that the kernel and candidate automaton were proved equivalent, allowing for termination; `false` in our application means that the property was not satisfied, and there is a counterexample to provide to the L* algorithm; and `unknown` in our application means that the verifier was unable to prove equivalence, but also does not provide a counterexample. In the case of an `unknown` answer from the verifier, we halt our algorithm and warn the user

that the resulting automaton is *approximate*; there may be inputs for which the automaton returns an incorrect answer.

3.1.4 *Synthesizing Hardware Descriptions from Automata*

Once a state machine has been learned using the L* algorithm with our custom MAT, the kernel is now amenable to acceleration. There has been a significant effort to accelerate automata using FPGAs [252] and other custom ASICs (e.g., GPUs [231, 261] and Micron's AP [75]). We convert the learned automaton to a hardware description and synthesize the design for loading onto an FPGA. Other execution platforms are possible [12], but we focus on FPGAs in this work due to their widespread deployment.

3.1.5 *System Architecture*

Figure 3.1 depicts the high-level system architecture of our framework. The L* learner (shown to the left) queries a MAT (shown to the right) consisting of the legacy source code, software model checker, SMT solver, and string decision procedure. The legacy string kernel is used by the MAT to answer membership queries. Termination queries are transformed by a mapper into a software verification problem that searches for string that distinguish the language of a candidate automaton from the target language implemented in the kernel. The output of the

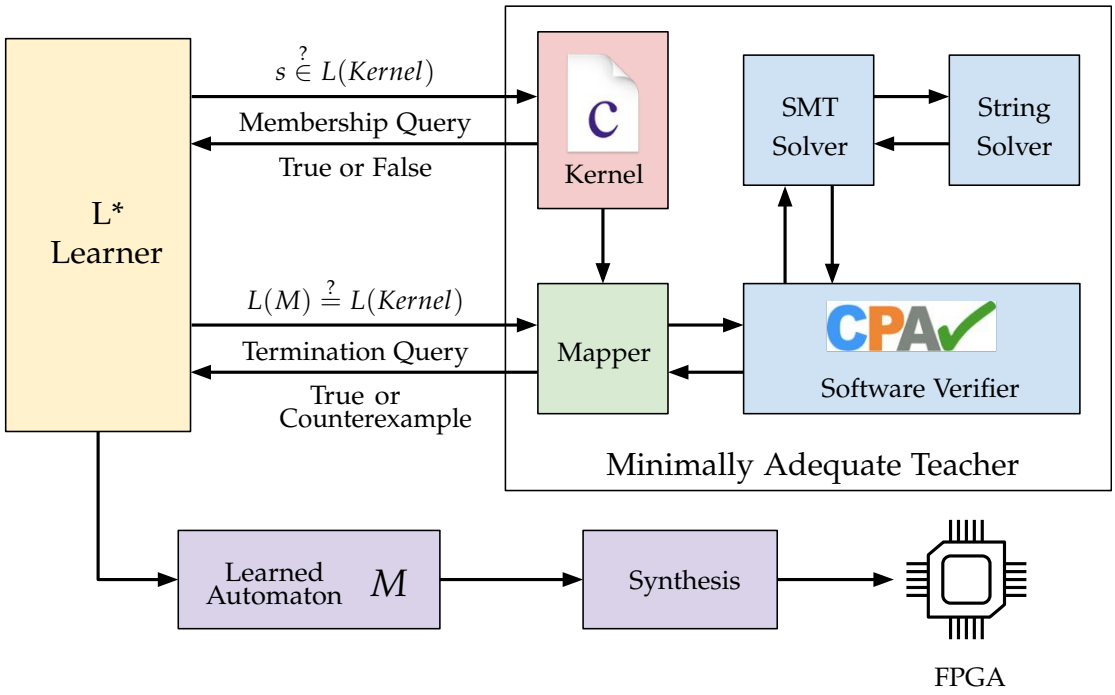


Figure 3.1: AutomataSynth system architecture. The Minimally Adequate Teacher uses the legacy kernel to answer membership queries. The kernel is combined with a candidate automaton in the mapper to produce a software verification problem. Using bounded software model checking combined with string decision procedures, we search for a counterexample that distinguishes the target language from the language of the candidate automaton. Finally, we synthesize the learned automaton for execution on an FPGA.

Learner is a DFA that encodes the same computation as the Kernel. We use this DFA to synthesize a hardware design for execution on an FPGA.

3.2 IMPLEMENTATION AND CORRECTNESS

In this section we lay out formal properties of our implementation, first demonstrating that iterative, bounded software model checking conforms to the required properties for MAT termination queries. We then prove correctness and termination for AutomataSynth. Because AutomataSynth operates on arbitrary source code and employs theorem proving techniques, correctness and termination are *relative*.

3.2.1 *Bounded Model Checking*

There are several SMT-based model checking algorithms that have been employed to verify properties of software. We use bounded model checking, an algorithm best-suited for the queries currently supported by string constraint solvers. In particular, string solvers do not currently support most interpolation queries (e.g., [66]), which are used heavily by counterexample-guided [60] algorithms such as SLAM and BLAST [23, 35]. Developing interpolation algorithms that support string constraints is beyond the scope of this work.

Bounded model checking enumerates all program paths up to a certain bound that reach a target error state (e.g., an error label in the source code) [37]. For each path, the algorithm generates a set of constraints over the program's variables, and the disjunction of these constraints is passed to an SMT solver to determine if the constraints for at least one path are satisfiable. If so, the set of variable assignments

represents a configuration of the program that would result in an error condition at runtime. In this approach, loops are unrolled a fixed number of times (the "bound"). We follow the standard practice of incremental unrolling (cf. [43]), which iteratively applies bounded model checking for increasing unrolling depths. For programs with unbounded loops this strategy results in a semi-algorithm; however, we demonstrate in Section 3.2.4 that there exists a finite unrolling that fully verifies a kernel deciding a regular language for our property.

3.2.2 *Reasoning about Strings*

As described in Section 3.1.3, AutomataSynth must verify that there are no strings in the symmetric difference of the legacy kernel and a candidate automaton. We encode the language of the candidate automaton as a regular expression constraint on the input string parameter of the kernel. We then solve the encoded problem using a bounded model checker that reasons about strings. A suitable string decision procedure must support (at minimum):

- Unbounded string length,
- Regular expression-based constraints over strings,
- Access to individual characters of strings,
- Comparison of individual characters and strings,
- Reasoning about the length of strings, and

- The ability to generate strings that satisfy a set of constraints.

Additional features supported by string decision procedures can be helpful for representing standard library string functions. Recent decision procedures, such as Z3str3 [31] or S3 [222], support these required properties.

To combine bounded model checking with string decision procedures, we extend the CPAChecker extensible program analysis framework [33] to generate and solve string constraints. We modify CPAChecker's predicate analysis algorithm to generate "String" sort constraints for string-like types in C programs (e.g., `char` and `char*` types [163]). We produce a character extraction constraint for each occurrence of indexing of (and dereferencing) string variables. Additionally, we add support for functions such as `strlen`.

The C language specification does not directly support regular expressions. To embed these constraints in a program's source code, we also add an additional function for checking if a string variable conforms to a regular expression.

3.2.3 *Verification for Termination Queries*

Listing 3.1 demonstrates our formulation of termination queries using bounded model checking with incremental loop unrolling and string decision procedures. We construct a wrapper around the source code for the kernel that adds additional assertions to the path constraints used by the software verifier. When the kernel returns `true`, we add the constraint that the input string cannot be represented by the regular expression representing the candidate automaton. We add a similar

```c
// KERNEL is the legacy kernel function
// REGEX is the language of the candidate automaton
int termination_query(char* input) {

  // call the kernel and record result
  int ret = KERNEL(input);

  if (ret) {
    // if kernel accepts, candidate DFA must reject
    __VERIFIER_assume(
      !__VERIFIER_inregex(input, REGEX)
    );
    goto ERROR;
  } else {
    // if kernel rejects, candidate DFA must accept
    __VERIFIER_assume(
      __VERIFIER_inregex(input, REGEX)
    );
    goto ERROR;
  }

  // Error state to prove unreachable
  ERROR:
  return ret;
}
```

Listing 3.1: Formulating termination queries as software verification problems. We embed regular expression constraints to force the legacy kernel and candidate automaton to disagree on the input string. If the return statement is unreachable, the two representations are equivalent. Otherwise, there is a string counterexample that can be used to continue the L* algorithm.

constraint for the `false` case as well. Finally, we ask the verifier to prove that the error label (line 23) is unreachable (note that `assume` constraints influence reachability).

3.2.4 *Correctness*

In this subsection, we conclude our formal development of termination queries based on the combination of bounded model checking with incremental loop unrolling and string decision procedures. We demonstrate that this approach satisfies the Angluin constraints for MAT termination queries (see Section 3.1.1). In particular, we prove that this algorithm always halts with a counterexample or proof of equivalence between the legacy string kernel and a candidate automaton (assuming the underlying decision procedure is correct). While incremental unrolling is a semi-algorithm in general, we demonstrate that a finite unrolling is sufficient to prove equivalence *for pure programs that decide a regular language* (i.e., programs that both recognize a regular language and halt on all inputs as described in Section 3.1.2). We assume that the program is pure to avoid non-deterministic behavior and side-effects (e.g., non-deterministic behavior resulting from I/O) and that the program decides a regular language to leverage formal results from automata theory. While these assumptions restrict the class of programs to which our formal result applies, we note that AutomataSynth can handle more complex functions, but may produce approximate results. Ultimately, our goal is to prove the following theorem:

Theorem 3.2.1. *Let K be a pure program that decides a regular language $L(K)$. There exists a finite unrolling K' of K such that if M is the candidate DFA learned by L* from K', then $K \equiv M$.*

We write $(\equiv)$ to denote equality of accepted languages, i.e., $K \equiv M$ if and only if $L(K) = L(M)$. We will prove this theorem using a sequence of lemmas as well as theoretical results from the L* algorithm. First, we demonstrate that there exists a finite unrolling of a program K that recognizes all strings in $L(K)$ shorter than a given length. The intuition is that the finite unrolling K' corresponds to the use of *bounded* model checking.

Lemma 3.2.2. *Let $p \in \mathbb{N}$ and K be a program that recognizes a subset of Σ^*. There exists an $n \in \mathbb{N}$ such that the n-finitely-unrolled program K' obtained from K (with all loops replaced with the finite unrolling of the first n iterations and all subsequent iterations removed) satisfies $\forall s \in \Sigma^*, |s| < p \implies K'(s) = K(s)$.*

Proof. Given $p \in \mathbb{N}$, we construct the set of strings $S = \{s \mid s \in \Sigma^* \wedge |s| < p\}$, on which K' must agree with K. We let n be the maximum number of iterations performed by $K(s)$ for all $s \in S$. Because S is a finite set, its maximum value is guaranteed to exist and be finite. We construct K' by unrolling n times the program K. By construction, the property $\forall s \in \Sigma^*, |s| < p \implies K'(s) = K(s)$ holds. □

We also reason using the standard Pumping Lemma for regular languages. For reference, we recall the Lemma here without proof as defined by Sipser [199, Theorem 1.70].

Lemma 3.2.3 (Pumping Lemma for Regular Languages)**.** *If A is a regular language, then there is a number p such that for all $s \in A$, if $|s| > p$ then s may be divided into three pieces, $s = xyz$, satisfying the following conditions: for each $i \geqslant 0, xy^iz \in A$, $|y| > 0$, and $|xy| \leqslant p$.*

The smallest such p is called the *pumping length*. We call out as a Lemma the association between pumping lengths and minimality [199, Proof of 1.70]:

Lemma 3.2.4. *The (smallest) pumping length of a regular language L is equal to the number of states in the* minimal *DFA that recognizes L.*

Additionally, our proof makes use of two theorems about the output of the L* algorithm. We paraphrase these results here [10]. See Section 3.1.1 for L* definitions, such as (S, E, T).

Theorem 3.2.5 (L* [10], Theorem 1)**.** *If (S, E, T) is a closed, consistent L* observation table, then the DFA M constructed from (S, E, T) is consistent with the finite function T. Any other DFA consistent with T but not equivalent to M must have* more *states.*

We will use the following corollary of this result.

Corollary 3.2.5.1. *Let p be the pumping length of the target language, L, and M be a DFA constructed from a closed, consistent L* observation table. The pumping length of $L(M)$ does not exceed p.*

Finally, we make use of the L* algorithm termination result. The property we use in our proof has been emphasized.

Theorem 3.2.6 (L* [10], Theorem 6)**.** *Given any MAT presenting a regular language* L, L* EVENTUALLY TERMINATES AND OUTPUTS A DFA ISOMORPHIC TO THE MINIMUM DFA ACCEPTING L. *Additionally, if n is the number of states in the minimum DFA recognizing L and m is an upper bound on the length of any counterexample provided by the MAT, then the total running time of L* is bounded by a polynomial in m and n.*

With these properties in hand, we are now ready to prove our original theorem.

Proof (Theorem 3.2.1). Given a pure program K, which decides a regular language, and a candidate DFA M constructed from a closed, consistent L* observation table (S, E, T), let p be the pumping length of $L(K)$. By Theorem 3.2.4, the minimal DFA that recognizes $L(K)$ has p states. By Theorem 3.2.2, there exists a finite unrolling K' of program K such that $\forall s \in \Sigma^*.|s| < p \implies K'(s) = K(s)$. We will show that verifying $K' \equiv M$ is sufficient to verify $K \equiv M$ using bounded model checking.

Verifying the property $\nexists t \in \Sigma^*$ such that $t \in L(K') \oplus L(M)$ (the symmetric difference, i.e., $t \in L(K') \cup L(M)$ and $t \notin L(K') \cap L(M)$) with incremental bounded model checking (recall K' is a finite unrolling) can result in two outcomes:

Case 1: $\exists t \in \Sigma^*$ such that $|t| < p \wedge K'(t) \neq M(t)$.

Case 2: $\forall t \in \Sigma^*$ such that $|t| < p$, $K'(t) = M(t)$ holds.

In the first case, we return t as a counterexample, concluding $K \not\equiv M$. In the second case, we conclude that $K' \equiv M$ and any counterexample must be at least as long as p; however, no such counterexample exists. The proof proceeds by contradiction.

Suppose, for the sake of contradiction, that $\exists t' \in \Sigma^*$ such that $|t'| \geqslant p$ and $K(t') \neq M(t')$. Let n be the number of states in the candidate DFA M. We now relate n to the number of states in the minimal DFA recognizing $L(K)$. By Theorem 3.2.4 and Corollary 3.2.5.1, $n \leqslant p$ because the pumping length of $L(M)$ is at most p and the number of states in M is equal to the pumping length of $L(M)$. Additionally, because the finite unrolling $K' \equiv M$, $n \geqslant p$ by Theorem 3.2.5. Therefore, the number of states in M is bounded above and below by the pumping length of our target language, implying that $n = p$. Using our assumption about t', we note that K is consistent with T but not equivalent to M, and thus by another application of Theorem 3.2.5 we conclude that the DFA recognizing $L(K)$ must have more than $n = p$ states. This contradicts the fact, from Theorem 3.2.4, that the minimal DFA recognizing $L(K)$ has exactly p states. Therefore, no such t' exists.

Because L* produces a minimal DFA (Theorem 3.2.6), and M was produced from a closed, consistent observation table, we can conclude that M must be a DFA isomorphic with the minimal DFA accepting the language $L(K)$. Thus, $K \equiv M$.

This means that, using bounded model checking on the program K' (recall that K' is a finite unrolling and thus admits *bounded* model checking), we either find a counterexample or can conclude equivalence of K and M. Therefore, $K' \equiv M \implies K \equiv M$. □

From this result, we can establish the following corollary, which allows us to conclude that our approach may be used in a MAT to answer termination queries.

Corollary 3.2.6.1. *For a given program K, there exists a finite number of iterations of incremental unrolling needed for our approach to respond to a termination query with either a counterexample or a proof of equivalence.*

3.2.5 *Implications.*

In our formulation, termination queries return an answer if the bounded software model checking with incremental loop unrolling and the string decision procedures terminates. Our result is therefore *relative* to the completeness of the model checker and underlying SMT theories (see Ball et al. for a discussion of relative completeness in software model checking [22]). For pure kernels that decide a regular language, we proved that there is a finite bound on the incremental unrolling that will determine equivalence of the kernel and a candidate automaton. In practice, we make use of a timeout on the verification process to ensure timely termination at the expense of correctness in some cases. This design decision results in an approximate solution in cases where either the finite unrolling bound has not yet been reached or the legacy kernel recognizes a non-regular program. The approximate solution is correct for strings of length up to a particular bound but may disagree on larger strings. Our empirical evaluation in Section 3.4 demonstrates that AutomataSynth successfully learns an equivalent state machine for thirteen of eighteen real-world string kernels mined from legacy source code.

3.3 EXPERIMENTAL METHODOLOGY

In this section, we describe our process for selecting real-world, legacy string kernels benchmarks as well as our experimental setup for the evaluation described in Section 3.4.

3.3.1 *Benchmark Selection*

In our evaluation, we focus on measuring the extent to which AUTOMATASYNTH learns models for real-world string functions using varied library methods. We construct our benchmark suite by mining legacy string kernels from open-source software projects on GitHub using the following protocol. First, we filter all projects for those with C source code and ordered the resulting repositories by number of stars (i.e., popular repositories first). Next, we use the Cil [163] framework to iteratively parse each source file and extract all functions with an appropriate type signature (see Section 3.1.2). We filter these functions to exclude those that referenced functions or data outside the compilation unit. We allow the use of common library function (e.g., `strlen`, `strcmp`, etc.). In total, we considered 26 repositories and mined 973 separate string kernel functions using this protocol.

After filtering for duplicates and a manual analysis to identify functions that return Boolean values (we note that while C has the `_Bool` data type, many functions still use integers of varying widths), we collected 18 meaningfully distinct real-world benchmarks. Table 3.1 provides an overview of these string

Table 3.1: Benchmark Suite of Real-World, Legacy String Kernels

FUNCTION	PROJECT	LOC	SUPPORT
`git_offset_1st_component`	*Git*: Revision control system	6	✓
`is_encoding_utf8`		38	✘*
`checkerrormsg`	*jq*: Command-line JSON processor	4	✓
`checkfail`		14	✓
`skipline`		17	✓
`end_line`	*Linux*: OS kernel	11	✓
`start_line`		11	✓
`is_mcounted_section_name`		54	✓
`is_numeric_index`	*MASSCAN*: IP port scanner	17	✓
`is_comment`		11	✓
`AMF_DecodeBoolean`	*OBS Studio*: Live streaming and recording software	2	✓
`cf_is_comment`		28	✓
`cf_is_splice`		22	✓
`is_reserved_name`		39	✓
`has_start_code`		18	✓
`number_is_valid`	*openpilot*: Open-source driving agent	72	✘†
`utf8_validate`		72	✘‡
`stbtt__isfont`		24	✓

*Requires `strcasecmp` support †Requires `strtod` support
‡Performs math on characters

kernels. We use the function name to refer to each benchmark and also indicate the source project for each. Lines of code (LOC) provides a count of the total number of non-comment lines in the post-processed version of the benchmark. Finally, we also indicate whether the kernel is supported by our prototype system. Our prototype implementation supports all but three of these legacy string kernels. The unsupported kernels use computation that is difficult to capture with present string decision procedures.

The kernels in our benchmark suite interact with strings in various manners. Some kernels, such as `is_numeric_index`, `skipline`, and `cf_is_comment`, loop over all characters in the string checking various constraints. Several also make heavy use of `strcmp` to check for the presence of specific strings (e.g., `checkerrormsg`, `is_mcounted_section_name`, and `start_line`). We also found examples of kernels (e.g., `git_offset_1st_component` and `AMF_DecodeBoolean`) that perform single character comparisons. While a developer will likely not be interested in accelerating a single character comparison, these kernels remain indicative of real-world code and allow us to demonstrate a proof-of-concept for synthesizing designs for accelerators such as FPGAs. An evaluation of benchmarks more typical of kernels accelerated by FPGAs (e.g., long-running kernels with hundreds or thousands of states) is left for future work.

3.3.2 *Experimental Setup*

Our AutomataSynth implementation produces MNRL, an open-source state machine representation language intended for large-scale automata processing applications [12]. We transform the learned DFA to be *homogeneous*, a property that admits a simplified transition rule while maintaining expressive power and that is amenable to hardware acceleration [13, 48, 75, 231]. We use Brzozowski's algorithm [46] for converting candidate DFAs to regular expressions as part of the software verification step (see Section 3.2).

Table 3.2: Experimental Results

BENCHMARK	MEMBERSHIP QUERIES	TERMINATION QUERIES	NUMBER OF STATES	TOTAL RUNTIME (S)	CORRECT
`git_offset_1st_component`	4,090	2	2	7	✓
`checkerrormsg`	32,664	2	15	86,195	✓*
`checkfail`	189,013	3	35	86,308	✓*
`skipline`	7,663	3	3	294	✓
`end_line`	510,623	4	44	29,531	✓
`start_line`	206,613	2	46	4,813	Approx.
`is_mcounted_section_name`	672,041	7	57	86,399	Approx.
`is_numeric_index`	10,727	3	4	297	✓
`is_comment`	4,090	2	2	14	✓
`AMF_DecodeBoolean`	2,557	2	2	4	✓
`cf_is_comment`	4,599	2	4	300	✓
`cf_is_splice`	1,913	2	4	3	✓
`is_reserved_name`	350,705	8	42	85,469	✓
`has_start_code`	10,213	2	7	5	✓
`stbtt__isfont`	79,598	5	19	13	✓

*AutomataSynth warned of a potential approximate solution due to timeout, but manual analysis confirmed correctness

For termination queries, we add string constraint handling to CPAChecker 1.8 [33]. We also extend the JavaSMT framework [120] to support the draft SMT-LIB strings theory interface [217]. We use Microsoft's Z3 version 4.8.6 SMT solver [71] with the *Seq* string solver [222] for all queries. All evaluations use an Ubuntu 16.04 Linux server with a 3.0 GHz Intel Xeon E5-2623-v3 with four physical cores and 16 GB of RAM and a maximum time budget of 24 hours.

3.4 EVALUATION

In this section, we evaluate AUTOMATASYNTH on fifteen real-world, legacy string kernels mined from open-source projects. We first evaluate the correctness of the state machines generated by AUTOMATASYNTH and report runtime and query counts. Second, we evaluated the suitability of the generated automata for hardware acceleration. Our evaluation focuses on metrics related to *legacy support* and *performance*. At a high level, we are guided by the following research questions:

1. How many of the real-world string kernels can AUTOMATASYNTH correctly learn? With approximation?
2. Does AUTOMATASYNTH learn automata that fit within the design constraints of modern, automata-derived, reconfigurable architectures?

3.4.1 *State Machine Learning*

Table 3.2 presents results from our empirical evaluation of AUTOMATASYNTH on a benchmark suite of fifteen legacy string kernels. We do not report results for the three benchmarks that are not supported. We report the number of membership and termination queries executed for each kernel as well as the number of states in the learned automaton and the total runtime in seconds. The final column indicates if AUTOMATASYNTH correctly learned the kernel's functionality. A check

mark means that our tool learned a fully equivalent automaton. We also indicate approximate results in which the maximum time limit was exceeded.

On average, it took seven hours to learn an automaton from the legacy string kernel, with more than half of the benchmarks terminating in fewer than five minutes. AutomataSynth correctly learned thirteen of the fifteen benchmarks. The remaining two benchmarks yield approximate solutions, with many of these approximations being extremely similar to the target kernel functionality. In our evaluation this approximation was always the result of timeouts rather than the relative completeness of the SMT solver used for termination queries. There were no instances in our benchmark set for which the SMT solver returned an unknown result due to a limitation in the string decision procedures.

We determined that there were two primary causes for AutomataSynth reaching the timeout without learning a fully equivalent state machine. First, Brzozowski's algorithm for constructing a regular expressions can produce large expressions that require simplification to remove redundant and superfluous clauses. This was most relevant to kernels that compared string suffixes with a string constant. We believe this performance limitation is an artifact of design choices in our prototype, which could be solved with more careful construction of regular expressions. Second, some SMT queries were significantly less performant than others. We discuss this challenge in more detail in Section 3.5.

The relative utility of the membership and termination queries varies between the benchmarks. For example, the function `git_offset_1st_component` checks a string to see if the first character is a forward slash (`/`). Using membership queries,

AutomataSynth learned that the first character of the string must be a slash and that any number of characters may follow. The termination query provided a single counterexample of a longer string that was initially misclassified. For this kernel, the membership queries provided much of the "learning". This is in contrast to the `stbtt__isfont` kernel, which ultimately compares an input string against four hard-coded strings. In this case, the membership queries only provided minimal information. Instead, the termination queries discovered the string constants in the kernel's source code and provided much of the learned information. In general, membership queries tended to provide more information when each character in the input string was considered separately while termination queries helped to discover string constants used for comparison by the kernels.

AutomataSynth successfully learned automata for fifteen of the eighteen legacy kernels mined from open-source projects. Of these, thirteen were exactly equivalent and two were near approximations.

3.4.2 *Hardware Acceleration*

In this work, we claim no novelty for accelerating automata using hardware accelerators, such as FPGAs. Instead, we leverage existing work in the area of high-performance automata processing. On FPGAs, Xie et al.'s REAPR framework supports high-throughput processing of data with finite automata on FPGAs [252]. For spatially reconfigurable architectures akin to FPGAs, the dominant factor

affecting performance is the number of hardware resources used by a design. For ANMLZoo benchmarks, which contain tens of thousands of states [231], REAPR successfully synthesized designs running in the range of 200–700 MHz. Because the automata learned by AutomataSynth are significantly smaller, we expect that similar throughputs could be achieved.

The finite automata learned by AutomataSynth fall withing the design constraints of FPGA-based automata accelerators, allowing for high-throughput execution.

3.5 DISCUSSION

At a high level, AutomataSynth learns the behavior of a Boolean string kernel through a combination of dynamic and static analyses and emits a functionally equivalent state machine that is amenable to acceleration with FPGAs. We believe that approaches such as AutomataSynth are very promising and could offer solutions to limitations inherent to current HLS techniques. HLS relies heavily on the structure of C-like source code to produce a hardware description, which were designed for performance on—and as an abstraction of—von Neumann architectures. As such, HLS is unlikely to produce performant FPGA designs from legacy code that was heavily optimized for CPUs [223, 268]. This places a heavy burden on developers tasked with porting code and represents a significant barrier to adoption. Our approach decouples the implementation choices of the

legacy program from the emitted hardware design. This allows us to produce a design using a model of computation—state machines—that is performant on FPGAs [231, 252].

This chapter represents an initial effort to understand the benefits and limitations of using state machine learning algorithms to compile code for FPGAs. A significant research effort remains for approaches akin to AutomataSynth to be mature enough for industry adoption. In the remainder of this section we identify four key research challenges whose solutions would lead to significant advances in learning-based synthesis for FPGAs. Additionally, we describe candidate future directions to tackle each of these.

3.5.1 *Learning More Expressive Models*

We present an approach for accelerating regular language Boolean string kernels with FPGAs. Our prototype soundly transforms such kernels to functionally equivalent hardware descriptions; Boolean functions with inputs that may be transformed into a serial data stream are also applicable. However, legacy code contains many other types of functions, and these remain an open challenge. Supporting a new function type presents a two-fold challenge: (1) identifying suitable computational models for acceleration and (2) designing or adapting an algorithm suitable for learning these models. Finite automata, as formally defined, produce a single bit of output for each string processed and are limited to recognizing Regular Languages. Additional models, such as Mealy and Moore

machines, support transforming an input value into an output value, while others, such as pushdown automata, support more expressive classes of languages.

Several efforts are underway to extend learning algorithms to more expressive computational models [41, 50, 157]. It may also be possible to leverage insights from the architecture community and recent efforts to accelerate automata processing, in which designs often support tagging output *report* signals with additional metadata [75, 233]. Further, existing DFA learning algorithms may admit learning functions that output an enumerated—or even a multi-bit—value.

An additional challenge is that determining program equivalence is, in the limit, undecidable. For example, Angluin notes that termination queries are not generally decidable for context-free languages [10]. However, existing software verifiers suffer from this same challenge and provide relative completeness [22]. Further, this challenge may be addressed in some cases through careful use of approximation.

3.5.2 *Expressive Power and Performance of String Solvers*

Our empirical evaluation of AutomataSynth demonstrated some limitations of present string decision procedures. Certain string operations (e.g., case-insensitive lexicographic comparisons and casting between characters and numbers to perform arithmetic operations) occur in real-world software but are difficult to represent as constraints in String theories. Additionally, SMT queries generated by bounded model checking algorithms can result in long-running computation.

These challenges are not new: the formal methods community has been laboring to improve string decision procedures for over a decade. Early efforts often focused on the problem domain of identifying cross-site scripting and SQL code injection vulnerabilities (e.g., [103]) and introduced new constraint types. These efforts often reasoned about fixed-sized string variables (e.g., [127]). Subsequent efforts, such as Z3str3, also focus on improving the performance of these decision procedures and have extended support to unbounded strings [31].

AutomataSynth is one of the first efforts to combine bounded software model checking with string decision procedures. This combination presents a novel and compelling use case for string solvers that requires new constraint types and optimizations. We make our tool and all of the SMT queries automatically generated by our process available[4] to the community to encourage renewed interest in—and efforts to—improve the performance of string solvers.

3.5.3 *Scaling Termination Queries*

We found, in practice, that termination queries consumed an average of 66% of the total runtime of AutomataSynth. As candidate state machines increase in size, we expect the scalability of termination queries to dominate. This challenge presents an opportunity for innovation. We presented an approach based on the novel combination of bounded software model checking and string decision

4 See the AutomataSynth repository at `https://github.com/kevinaangstadt/automata-synth`

procedures; however, alternate formulations of termination queries may provide better performance while maintaining correctness.

Many applications of model learning focus on the use of testing to provide answers to termination queries [225]. We have observed that the application of automated testing presents several challenges, such as producing a suitable quantity and diversity of inputs to identify counterexamples. Test input generators, such as Klee [47], may only support bounded length strings (rather than unbounded).

The application of other software verification techniques may also provide performance gains. Counterexample-guided abstraction refinement verifiers can abstract much of a program's state to gain performance, but require support for interpolation queries. These are not currently support by string solvers, but present an additional area for research.

3.5.4 *Characterizing and Taming Approximation*

Because scalability and decidability of termination queries are challenges, approximation may play an important role in improving the performance of learning-based approaches to synthesizing FPGA designs. Indeed, there is already significant interest in the architecture and software communities for producing approximate programs [156, 160, 170, 178].

Approximation has been a key parameter in model learning algorithms from the start [10, 225]. Results from learning theory often analyze approximation using Valiant's *probably approximately correct* (PAC) framework, which bounds the

probability of the error being less than a fixed threshold for an approximately learned model [226]. Such results can predict the number of queries necessary to bound the error but *do not characterize the locations or significance of the remaining error*. Anomalous results for frequently used inputs have a very different impact than anomalous results for seldom-used inputs. Given the design of Angluin-style algorithms, it may be possible to determine which inputs result in approximate solutions. For example, pre-populating the observation table with rows pertaining to known inputs (i.e., those taken from the test suite) ensures that the learned state machine produces the correct output for those relevant values.

3.6 CHAPTER SUMMARY

We present AUTOMATASYNTH, a framework for accelerating legacy regular language Boolean string kernel functions using FPGAs. Our approach uses a novel combination of state machine learning algorithms, software verification algorithms, string decision procedures, and high-performance automata processing architectures to learn the behavior of a program and construct a behaviorally equivalent FPGA hardware description. We demonstrate a proof-of-concept of this approach using a benchmark suite of eighteen string kernels mined from open-source projects on GitHub. AUTOMATASYNTH successfully constructs equivalent (or near equivalent) FPGA designs for more than 80% of these benchmarks. We believe this approach shows promise for overcoming some of the limitations of current HLS techniques.

By leveraging automata abstractions, we are able to successful port certain classes of legacy code to execute efficiently on hardware accelerators. AUTOMATA-SYNTH thus meets the requirements of *legacy support* and *performance* as detailed in Section 1.1. In the next chapter, we explore a custom programming language to help developers write new applications for hardware accelerators.

CHAPTER 4

RAPID: A High-Level Language for Portable Automata Processing

Having demonstrated the utility of automata-based abstractions for porting extant code, we next focus on leveraging automata for the development of new software for hardware accelerators. Because accelerator ecosystems can often be heterogeneous [5, 162, 175], we first evaluate the extent to which automata processing enables the portability of applications across CPUs, GPUs, and FPGAs, which have all been considered for the execution of automata applications [166, 231]. Then we develop a high-level programming language, RAPID, for representing pattern search problems with respect to NFAs, targeting Micron's Automata Processor (AP), CPUs, GPUs, and FPGAs. Together, these two contributions provide a programming model that is portable, reduces code size, and improves maintainability.

To evaluate the portability of the automata processing paradigm, we consider two questions: 1) do design and optimization choices for finite automata port across architectures? and 2) to what extent does automata processing support high performance across architectures? In particular, we measure the *stability* of finite automata designs across hardware platforms. We evaluate six implementation-

and optimization-techniques and demonstrate that performance gains achieved by these design choices are consistent across architectures. We contrast this result with the OpenCL programming model, which frequently demonstrates performance *inversions* across platforms. Further, we present a comparison of the performance of automata processing (demonstrated to be performant on hardware accelerators) with highly optimized, application-specific algorithms on CPUs. In total, our results indicate that the performance of automata algorithms shows great promise on the CPU platform. We argue that these stability and performance results demonstrate the viability of automata processing as a portable computation paradigm.

While automata processing provides a suitable abstraction for performance portability, finite automata programming is tedious and error-prone. This chapter presents RAPID, a high-level language that maintains the performance benefits of pattern-recognition processors while also providing concise, clear, maintainable, and efficient representations of pattern-identification algorithms. We introduce three parallel control structures 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. We also demonstrate that RAPID maintains the performance and portability benefits of automata processing across multiple architectures, including CPUs, GPUs, FPGAs, and the Micron D480 AP. We present algorithms for converting RAPID

programs into NFAs for execution via automata processing. We describe code generation and tool pipelines that are efficient across all target architectures.

To evaluate the *expressiveness* of RAPID, we re-implement a benchmark suite, in RAPID, of real-world automata-based applications that have significant speedups when executed using specialized hardware accelerators. Then, we evaluate the *performance* and *scalability* of these compiled RAPID programs against their hand-crafted equivalents, measuring program size, resource utilization, and runtime metrics. Our evaluation demonstrates that RAPID programs introduce little overhead compared with applications written at a lower level of abstraction and maintain the performance and functional portability provided by the automata paradigm.

In summary, this chapter makes the following contributions:

- An empirical evaluation of the stability and performance of automata processing optimizations and design choices across CPUs, GPUs, and FPGAs.
- RAPID, a high-level language for programming automata processing applications.
- A set of algorithms for converting RAPID programs into non-deterministic finite automata for execution with multiple automata processing engines.
- An experimental evaluation of the RAPID language against hand-crafted applications demonstrating improved density of generated NFAs as compared with hand-optimized NFAs.

The remainder of this chapter is organized as follows. section 4.1 presents our empirical evaluation of automata processing stability with respect to state-of-the-art algorithms. Section 4.2 describes the RAPID programming language. Next, section 4.3 describes the algorithms for generating Finite Automata from a RAPID program. Section 4.4 discusses the tool pipelines for compiling and executing Finite Automata applications on CPUs, GPUs, FPGAs, and the D480 AP. Finally, Section 4.5 evaluates the performance of the RAPID programming language.

4.1 AUTOMATA PROCESSING STABILITY

In this section, we evaluate the suitability of the automata processing paradigm as a performant, portable programming abstraction across disparate computer architectures. We consider both the *stability* of implementations across architectures (whether design choices impact performance on platforms differently) as well as average throughput of applications, as compared with state-of-the-art algorithms. While a thorough evaluation of performance portability is out of scope, our initial results demonstrate the potential of automata processing as a suitable abstraction.

4.1.1 *Performance Stability*

We first compare the stability of design choices in automata processing applications with the stability of those in OpenCL. OpenCL supports execution across a variety of architectures [208]. However, code written for one processor may not

Table 4.1: Performance stability of OpenCL programs

BENCHMARK	CPU	GPU	FPGA	STABLE
CFD	↓	↓	↑	✗
Hotspot	↓	↓	↑	✗
LUD	↑	↑	↓	✗
NW	↓	↓	↑	✗
Pathfinder	↓	↓	↑	✗
SRAD	↓	↓	↑	✗

↑ – Loop-based performs best ↓ – Thread-based performs best

compile for another target or may require significant re-writing to be performant on the new architecture [268]. Given two implementations of the same application and two hardware architectures, if one implementation outperforms the other on the first architecture and the opposite is true for the second architecture, we say that there is a *performance inversion*. Performance *stability* is the lack of observable performance inversions.

The OpenCL language has many observable performance inversions and is therefore not stable across architectures. We demonstrate such inversions using applications in the Rodinia HPC benchmark suite, which were optimized for multi-threaded execution [52]. Zohouri et al. have developed a second implementation based on an iterative approach [268]. For each benchmark, we time both implementations on the CPU, GPU, and FPGA. Table 4.1 presents high-level relative performance results for loop- and thread-based OpenCL Rodinia benchmarks; performance is stable if arrows within a row do not reverse direction. We find that all six benchmarks demonstrate performance inversions. That is, for all

Table 4.2: Performance stability of Automata Processing optimizations

OPTIMIZATION	CPU	GPU	FPGA	AP	STABLE
Automata Folding	↑	↑	↑	↑	✓
Counters	↓	n/a	↓	↑	✗
DRM	—	—	—	↑	✓
Prefix Collapsing	↑	↑	↑	↑	✓
Race Logic	↓	↓	↓	↓	✓
Striding	↑	↑	↑	↑	✓

↑ – improved performance ↓ – reduced performance
— – no change

benchmarks we consider, the design decisions needed for performant code vary with each architecture.

We next examine performance stability in automata processing applications, focusing on six implementation and optimization techniques from recent literature:

- AUTOMATA FOLDING [221]: reducing automata states by combining non-overlapping input comparisons.
- COUNTERS [75]: reducing states by rewriting automata to use saturating counters.
- DISJOINT REPORT MERGING (DRM) [233]: reducing data transfer overheads on spatial automata processors.
- PREFIX COLLAPSING [28]: combining common automata states to form a trie-like structure.

- RACE LOGIC [147]: providing general support for dynamic programming at the cost of performance.
- STRIDING [28]: transforming automata to support compressed input streams.

For each, we select an arbitrary application that supports the optimization[1] from the ANMLZoo automata processing benchmark suite [231]. Using one implementation with the optimization and one without, we measure relative performance (i.e., relative time-to-solution) across CPUs, GPUs, FPGAs, and the AP. These results are presented in Table 4.2. We observe only a single performance inversion (counters) in our experiments and believe this inversion is an artifact of current implementation support.[2] These results provide initial evidence that the automata processing abstraction provides stable performance across architectures for many implementations and optimizations. Design decisions for performant code in automata processing do not appear to vary as much across architectures as they do with OpenCL.

4.1.2 *Automata Processing Performance*

In addition to stability, performant code across architectures is a desirable quality of a portable programming model. Recent studies by Wadden et al. and Nourian

1 No support results in the optimization being a no-op and thus has no impact on stability.
2 Nourian et al. support counters on GPUs [166], but their software artifacts have not been made public. Performance on the CPU and FPGA is degraded due to the complexity of circuit simulation (CPU) and routing constraints (FPGA). AP counters are discussed in Section 4.5.

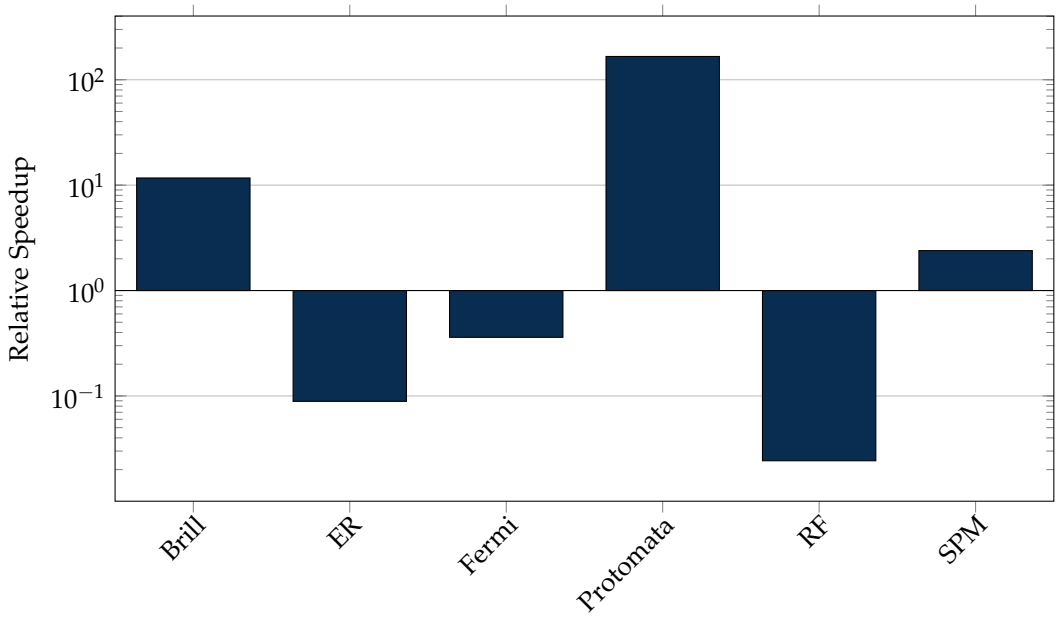


Figure 4.1: Relative performance of automata processing vs. application-specific algorithms on the CPU. Higher bars indicate better performance of the automata-based algorithm. Note that the y-axis is log-scale.

et al. investigate (and demonstrate) the performance of automata processing on several hardware accelerators, including GPUs, FPGAs, and the AP [166, 231]. Therefore, we restrict our attention in this section to CPU-based automata processing. We compare the performance of applications mapped to the automata processing paradigm with state-of-the-art CPU algorithms.

We evaluate all applications from the ANMLZoo benchmark suite [231] that were adapted from state-of-the-art, non-automata-based algorithms. These applications are:

- BRILL [267], a rule-writing processor for part of speech tagging in natural language processing.

- ENTITY RESOLUTION (ER) [40], an algorithm for detecting duplicated (or similar) names from a list.
- FERMI [240], a path recognition algorithm for particle physics experiments.
- PROTOMATA [185], a protein motif signature classification application.
- RANDOM FOREST (RF) [221], a random forest ensemble classifier for handwriting recognition.
- SPM [238], a sequential pattern mining application.

Each of these applications has been demonstrated to outperform a state-of-the-art CPU implementation when executed on the AP. Here, we study whether the algorithms designed for the AP outperform the state-of-the-art when executed on CPUs using an automata processing engine.

For each experiment, we executed the state-of-the-art implementation ten times and measured the average throughput of the core algorithm. Then, we averaged ten runs of an automata engine running the same application. We executed the benchmark automata using the Intel Hyperscan framework supplied as part of MNCaRT [12]. Experiments were performed on an Intel Core i7-5820K (3.30 GHz) processor with six physical cores and 32GB of RAM.

Figure 4.1 shows the relative speedup of automata engines over application-specific algorithms on the CPU. For three applications (Brill, Protomata, and SPM), the automata-based algorithm outperforms the state-of-the-art in terms of average throughput. By representing Brill and Protomata as automata, new

opportunities for optimization are exposed, allowing for orders of magnitude increased performance. For Fermi, the automata algorithm is within 3× of the application-specific algorithm. Entity Resolution and Random Forest are an order of magnitude slower primarily due to the increased accuracy and/or work [39] of the automata implementations. When adapting a new application to the automata paradigm, researchers should consider carefully how this might impact the work—the time or steps needed for a serial processor to complete the task—performed by the algorithm. Large increases in work may not be suitable for performant automata processing algorithms across architectures.

4.1.3 *Discussion*

We observe that automata processing is more stable across disparate architectures relative to design choices and optimizations than the OpenCL programming model. We also observe that four of our six automata benchmarks perform within 3× application-specific algorithms on the CPU, and two of these state machine-based implementation are at least an order of magnitude faster than the state-of-the-art. Additionally, automata processing is already a widely used computational model in areas such as network security [184], computational finance [3], and software engineering [7, 23]. There has been significant development of new optimizations for state machine performance on CPUs [27], and we anticipate continued improvement of automata processing performance on von Neumann architectures.

We conclude that automata processing provides *stability* (Section 4.1.1) and *performance* (Section 4.1.2) across architectures and implementations, including CPUs, GPUs, FPGAs, and the AP. That is, the performance of an automata-based algorithm is stable across architectures and often similar to the performance of an application-specific algorithm on the same hardware platform. Note that this performance portability includes applications that go beyond traditional regular expression-based algorithms, even on the CPU. We believe that portability across these architectures is beneficial to both the research and end-user communities. In particular, there is a lower overhead and risk incurred by developers who learn a programming model that is usable on multiple architectures. We believe that automata processing provides a *suitable abstraction* for representing and porting computation across multiple, dissimilar computer architectures.

4.2 THE RAPID LANGUAGE

While automata processing provides a suitable abstraction for performance portable execution of algorithms, current programming models for pattern searches have significant drawbacks (see Section 2.3). In this section, we discuss a new programming language, RAPID, which allows developers to write concise, clear, and maintainable algorithms for use with automata engines. In particular, RAPID supports searching a stream of data for many patterns in parallel. This sort of execution model is often referred to as *multiple instruction, single data* (MISD) in the architecture literature [83]. Programs are written in a combined imperative

and declarative style using a C-like syntax. In this section, we present a high-level overview of the control structures and data representations in the RAPID programming language.

4.2.1 *Program Structure*

MACROS AND NETWORKS. RAPID programs consist of one or more *macros* and a *network*. The basic unit of computation in a RAPID program is a *macro*, which defines a reusable pattern-matching algorithm. Macros in RAPID share similarities with both C-style macros and ANML[3] macros, allowing code to be written once and then used as a "rubber stamp." RAPID macros admit more customized usage than their namesakes in C and ANML; the same macro can generate all designs for a particular problem.

Statements within a macro are executed sequentially and define actions that should be taken to identify a pattern. RAPID provides several control structures, including `if` statements, `while` loops, and `foreach` loops. Unlike some languages, we guarantee in-order traversal when iterating with a `foreach` loop. The language also provides parallel control structures useful for pattern-matching, which we describe later in this section.

Additionally, macros can instantiate other macros. When a macro is called, control shifts to the called macro; all of its statements are executed, and then control returns to the calling macro. While the macro code defines how to identify

3 ANML is the automata representation language for the AP. See Section 2.3.1

```
macro hamming_distance (String s, int d) {
    Counter cnt;
    foreach (char c : s)
        if(c != input()) cnt.count();
    cnt <= d;
    report;
}
network (String[] comparisons) {
    some(String s : comparisons)
        hamming_distance(s,5);
}
```

Listing 4.1: A RAPID program for computing Hamming distances

a pattern in the input stream, the macro parameters can specify the particular characters to match, allowing for comparisons of varying lengths. Consider the macro in Listing 4.1, which performs a Hamming distance computation between a string parameter, `s`, and the input stream. Changing from comparison against a string of length five to a string of length twelve only requires passing a different string argument to the macro. As noted in Section 2.3.1, more than half of the code in the corresponding ANML implementation must be modified to make an identical change.

The *network* represents the highest level of pattern-matching within a RAPID program, and statements within a network definition are executed in parallel. The most common use of the network is to define a collection of macros for instantiation, which are executed in parallel at runtime to identify patterns in the input data stream. The network may also have parameters to specify certain values at runtime. Listing 4.1 contains a RAPID program that computes the Hamming distance for a number of given strings and reports on input within a distance

of five. The network is parameterized on an array of strings, which is used at runtime to specify the comparisons being made.

REPORTING. RAPID programs passively observe the input data stream; they cannot modify the stream. Programs can indicate interesting regions within the stream by using the `report` statement, which generates a *report event*. These events provide the offset in the input data stream where the report occurred and additional identifying meta data, such as the reporting macro. For the program in Listing 4.1, reports indicate offsets where the input stream is within a Hamming distance of five from the strings in `comparisons`.

BOOLEAN EXPRESSIONS AS STATEMENTS. Inspection of the input data stream is central to the RAPID programming model. Often, pattern identification algorithms only continue if a certain sequence of characters is detected. RAPID provides concise support for this common domain idiom by allowing Boolean expressions whenever full statements are allowed.[4] These declarative assertions terminate the thread of computation if the expression returns `false`. Line 5 in Listing 4.1 illustrates this usage.

4 This is merely syntactic sugar; the same behavior may be implemented using a less compact `if` statement.

4.2.2 *Types and Data in RAPID*

There are six primary data types in RAPID: `char`, `int`, `bool`, `String`, `Counter`, and `ref`. Both `String` and `Counter` are lightweight objects, while the remaining four are primitive types. The `ref` type stores a reference to an instantiated macro. Additionally, there is support for nested arrays of these types.

In RAPID, pattern-matching occurs in a stream of characters. Therefore, the language provides the `char` primitive type for interacting with input data. The input data stream, however, is a stream of bits and does not need to be interpreted as characters. To support this, a `char` may also store escaped hexadecimal values. RAPID also defines two character constants, which represent special symbols in the input stream: `ALL_INPUT` and `START_OF_INPUT`. The former represents any symbol within the input and the latter is a reserved symbol (character `0xFF`) for indicating the start of data. For example, if the input data stream consists of the flattening of an array, the entries would be concatenated into a stream, separated by the `START_OF_INPUT` symbol.

A `Counter` represents of a saturating up-counter. Upon instantiation, a counter is initialized to zero. Counters provide two functions: `reset()` and `count()`, which set the value to zero and increment by one, respectively. Although programs cannot access the internal value of the counter, it is possible to check against a threshold.

Listing 4.2 demonstrates the usage of counters and interacting with the input stream. The `foreach` loop iterates over each character in the string "`rapid`" se-

```
Counter cnt;
foreach(char c : "rapid") {
    if( c == input() ) cnt.count();
}
if( cnt >= 3 ) report;
```

Listing 4.2: The above code counts the number of characters matched in "rapid" and reports if the count is at least three

quentially. If that character matches the next character from the input stream, the counter is incremented. After iterating over the entire string, the program checks if the counter is at least three and reports if so. For example, if the stream contained "tepid," the count would be three, and there would be a report, but "party" results in a count of one and no report.

The input data stream in RAPID is privileged and is accessed via the `input()` function. A call to this function returns a single character from the head of the data stream. Access to the input data is destructive—no peeking or insertion is allowed during program execution. Calls to `input()` act as synchronization points across active threads in a RAPID program. Similar to how active states in an NFA process the same input symbol, all active threads execute up to an `input()` statement and then receive the same character from the input stream. For example, if the stream contains "abcd...," `input()` would return 'a' to all active threads of computation, and the stream would now contain "bcd..." There is no required number of calls to `input()` across threads and also no communication between threads. Threads with fewer calls to `input()` than another thread will simply terminate earlier. This data model supports the heterogeneity of MISD computations.

RAPID's design represents the input stream as a FIFO only accessible through a special function, `input()`, rather than as a special indexed array. This is for conceptual clarity: arrays afford a notion of random access into the stored data, while pattern-recognition processors support sequential access to an ordered sequential data stream. Global input access is intentionally similar to C's "fgetc" rather than "fread/fseek" or "mmap."

4.2.3 *Parallel Control Structures*

In pattern-matching problems, it is often useful to explore multiple possibilities in parallel. For example, a spam filter may wish to check for many black-listed subject lines simultaneously, or a gene aligner may begin matching a sequence at any point in the input stream. To facilitate such operations, RAPID provides both the network environment and also parallel control structures. Networks, as described previously, allow for parallelism at the macro level, which is useful for checking several patterns in tandem. The parallel control structures (`either/orelse`, `some`, and `whenever`) provide finer-grain control over parallel operations.

`either/orelse` STATEMENTS. This structure provides basic support for parallel exploration. An `either/orelse` statement consists of two or more blocks, which allows for an arbitrary, static number of parallel computations. Computation splits when an `either/orelse` statement is encountered during execution, and each of the blocks is executed in parallel. When the end of a block is reached,

```
either {
    hamming_distance(s,d); //hamming distance
    'y' == input();        //next input is 'y'
    report;                //report candidate
} orelse {
    while('y' != input()); //consume until 'y'
}
```

Listing 4.3: An example usage of an `either/orelse` statement

computation continues with the next statement in the program. No blocking or joining occurs, meaning that different paths in the `either/orelse` statement may begin executing the following statement at different times. This behavior is desirable because it allows for the matching of different length patterns containing the same suffix.

As an example usage of the `either/orelse` statement, consider the code fragment in Listing 4.3, adapted from the *MOTOMATA* benchmark [186] evaluated in Section 4.5. Candidates in the input stream are separated by the control character `'y'`. The computation should report the candidates within a Hamming distance of `d` from the string stored in variable `s`. We use an `either/orelse` statement to ensure that computation continues to the next candidate when the current candidate does not fall within the threshold. The first block of the `either/orelse` statement performs the Hamming distance comparison, while the second block consumes input until the control character is reached, always preparing the program to check the next candidate.

`some` STATEMENTS. In certain cases, for example instantiating macros based on the content of an array, the ability to generate a dynamic number of parallel paths is desirable. The `some` statement provides this functionality.

This statement is the parallel dual of a `foreach` loop. During execution, the program iterates over a provided array or string and instantiates a parallel thread of execution for each item. Similar to an `either/orelse` statement, the execution of each parallel thread continues with the subsequent statement in the program; different threads in the some statement may reach this next statement at disjoint times. The `some` statement in Listing 4.1 instantiates a Hamming distance macro for each string in the `comparisons` array. The number of parallel threads executed depends on the number of entries in `comparisons`.

`whenever` STATEMENTS. A common operation in pattern-matching algorithms is a *sliding window* search, in which a pattern could begin on any character within the input stream. The `whenever` statement consists of a Boolean guard and an internal statement. The guard specifies a condition on the input stream that must be true or a counter threshold that must be met before the internal statement is executed. At any point in the data stream (potentially multiple times) where this guard is satisfied, the internal statement will be executed in parallel with the rest of the program. A `whenever` statement is the parallel dual of a while statement. Whereas a while statement checks the guard condition before each iteration of the internal statement, a `whenever` statement checks the guard in parallel with all other computations, if any.

```
1 whenever( ALL_INPUT == input() ) {
2     foreach(char c : "rapid")
3         c == input();
4     report;
5 }
```

Listing 4.4: Execution of a sliding window search over the entire input stream for the string "rapid"

The code fragment in Listing 4.4 will perform a sliding window search for the string "rapid." The predicate within the guard will return true on any input, and therefore the block of code will begin execution at every character in the input stream. The whenever statement can also perform restricted sliding window searches depending on the predicate in the guard. For example, an application searching through HTTP transactions might use the predicate matching "GET" before matching specific URLs.

Sliding window searches are fundamental to stream pattern recognition. All RAPID programs perform a sliding window search on the START_OF_INPUT symbol. In the common case, this sliding window search occurs at the topmost level of a RAPID program, i.e. right within the network. To reduce verbosity, RAPID infers this whenever statement, only requiring developers to specify a whenever statement with non-default sliding window searches.

4.3 CODE GENERATION

In this section, we present techniques for converting RAPID programs into automata for execution with automata processing. Our technique takes two files as input: the RAPID program and a file annotating properties of the network parameters (e.g., lengths of arrays and strings). Our tool converts the RAPID program into two files: an ANML or MNRL[5] specification and host driver code. The ANML or MNRL file specifies the configuration of automata processing engine needed to perform the given pattern-matching algorithm given by the RAPID program. The driver code is executed on the CPU at runtime and handles execution of the automata processing core and collecting report events. This section focuses on the transformation of RAPID into the ANML or MNRL specification.

We employ a staged computation model to convert RAPID programs: comparisons with the input stream and counters occur at runtime, while all other values are resolved at compile time. To aid in partitioning, we annotate expressions with their return type during type checking. Allowable annotations include the five types listed in Section 4.2.2 as well as an internal `Automata` type, which denote expressions interacting with the input stream. Expressions annotated with `Automata` or `Counter` are converted into ANML or MNRL (allowing for runtime execution), while the remaining expressions are evaluated during compilation.

Our conversion algorithm recursively transforms RAPID programs into finite automata in much the same way that regular expressions can be transformed

5 MNRL is an open-source state machine representation language. See Appendix A.

into NFAs. Comparisons with the input stream are transformed into STEs. The statement in which the comparison occurs determines how the STEs attach to the rest of the automaton. Rules for transforming automata expressions determine the structure of the STEs within a given statement. We describe the conversion of expressions, statements and counters in turn.

4.3.1 *Converting Expressions*

Expression transformation results in the formation of a chain of STEs. No cycles are generated by expressions, but chains may include bifurcations. Figure 4.2 provides examples of transformations from RAPID expressions to automata structures. The most basic transformation is a comparison between a character and the input stream, generating a single STE. AND expressions behave as concatenation because reading from the input stream is destructive. The conversion of an OR expression generates a bifurcation in the generated automaton. A special case occurs when both sides of the OR expression contain input comparisons of length one. In these instances, we take advantage of STE character classes to specify multiple accepting symbols for a single STE.

Negations of expressions generate the most complex structures of all the expression types. Traditionally, an automaton is negated by swapping accepting and non-accepting states. This construction, however, does not work for our use case because RAPID programs consume the same number of symbols for an expression

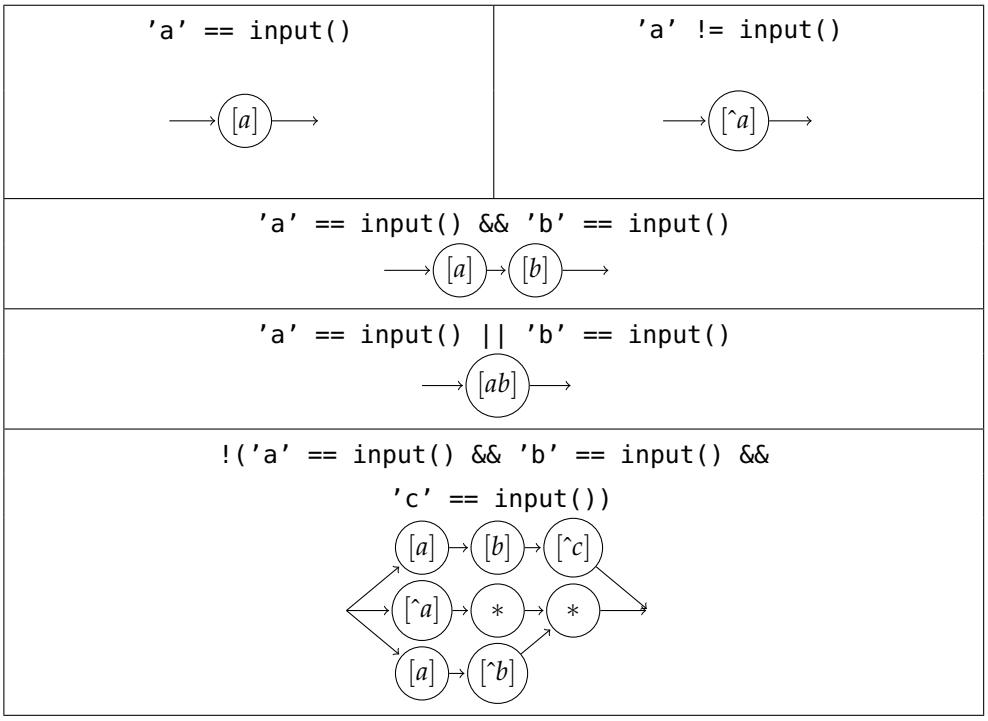


Figure 4.2: Transformations of RAPID expressions into automata

and its negation. The traditional transformation does not maintain this property. Instead, we transform the expression via De Morgan's laws and generate STEs for the resulting statement. After any mismatch in this negation, the remaining symbols do not matter, but still must be consumed. We therefore use *star* states, which match on any character.

4.3.2 *Converting Statements*

Statements in RAPID are transformed into the high-level automaton structures, allowing for additional pipelining, feedback loops, and parallel exploration of patterns. We present the overall structures in Figure 4.3.

During compilation, A `foreach` loop is unrolled into straight-line pattern-matching. Parallel `either/orelse` and `some` statements are transformed by generating the code for each statement and connecting these structures in parallel into the overall design. This mirrors the language semantics that the `some` statement is the parallel dual of `foreach`. Note that `some` statements typically depend on compile-time parameters (via input annotations on the network) while `either/orelse` statements do not (see Section 4.2.3).

There is also a similarity between `while` loops and `whenever` statements. `While` loops alternately perform predicate checks and execute the body code. This generates a feedback loop structure in the automaton. In a `whenever` statement, predicate checking begins on every character consumed. To support this, we generate a self-activating STE that accepts all symbols (see $*$ node in Section 4.3.2). This

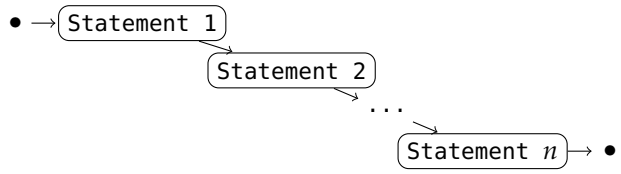
added STE maintains an active transition into the predicate, allowing matching to begin on every symbol consumed. Once the predicate accepts, the body of the `whenever` statement will begin to execute (although the predicate is still checked again in parallel on subsequent input characters).

4.3.3 *Converting Counters*

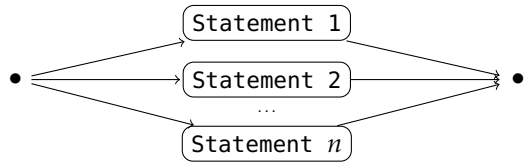
Counters in RAPID are challenging to implement because the state of a hardware counter on the AP cannot be directly accessed. Therefore, counter comparisons in RAPID programs are transformed into a pattern-matching operation using a combination of one or more saturating counters and Boolean gates. The basic structure consists of a saturating counter set to latch (once the threshold is reached, the output signal remains active) and an inverter, which allows for detection of the counter target not being reached.

Physical counters on the AP have three connection ports: count enable, reset, and output. `Counter` object function calls to `count()` and `reset()` in RAPID are connected to their respective ports on the counter. Output signals then connect to the next statement in the program.

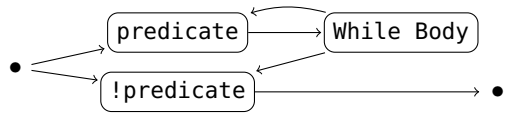
We follow the set of rules for determining the threshold and outputs of a `Counter` shown in Table 4.3. Equality checking with a `Counter` requires the use of two physical counter elements. While traversing the program, we note which



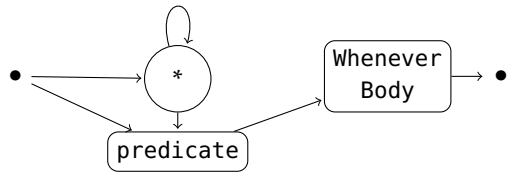
(a) `Foreach` Loops



(b) `Either/Orelse` and `Some` statements



(c) `While` Loops



(d) `Whenever` statement

Figure 4.3: Automaton designs for RAPID statements

Table 4.3: Rules for thresholds and outputs on counters

COMPARISON	THRESHOLD	TRUE OUTPUT
`< x`	x	inverted
`<= x`	x+1	inverted
`> x`	x+1	non-inverted
`>= x`	x	non-inverted
`== x`	convert to `<= x && >= x`	
`!= x`	convert to `< x || > x`	

`Counter` objects are used for equality checking and during code generation emit two counter elements for each.

This technique only allows for one threshold to be checked per counter in the RAPID program. An alternate solution would be to use positional encodings, which duplicate an automaton for each value of a counter, encoding the count in the position of states within an automaton. While this design allows for easy checking of multiple thresholds, it also significantly increases the number of states in the final automaton and does not support counter resetting. We chose not to implement this technique in our initial compiler because it does not support full, generic functionality.

We must also support the use of `Counter` variables as predicates in a `whenever` statement. For the body of a `whenever` statement to execute, the `Counter` must have reached its threshold, and the statement itself must have been reached within the control flow of the RAPID program. We use a self-activating STE matching all symbols to track when the statement is reached. An AND gate checks both of

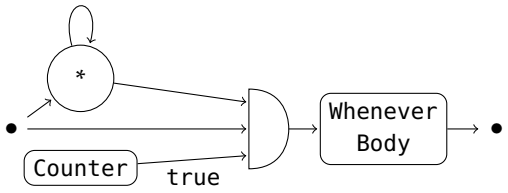


Figure 4.4: Structure of `whenever` statement with counters

these conditions before executing the body of the `whenever` statement. This design is demonstrated in Figure 4.4.

`Counter` threshold checks are also used as assertions or as predicates in `if` statements and `while` loops. Because NFAs do not have dynamic memory (beyond the states themselves), we handle this case by both generating automata and also pre-transforming the input stream. For each such `Counter`, we create a unique reserved input symbol. This new symbol indicates that the threshold for that particular `Counter` has been met. We add an STE matching the symbol to the subsequent statement; whenever the symbol is encountered in the input data stream, the appropriate subsequent statement begins execution. This symbol must be injected into the input data stream before the RAPID program begins execution. Actual injection is handled by the runtime code and can occur while data is being streamed to the AP (but before execution of the RAPID program begins).

We attempt to automatically determine the pattern for inserting the count threshold symbol into the input stream. An example pattern is "insert the symbol after every 25 characters in the input stream." Often, the compiler can infer the pattern by counting the number of symbols consumed before the counter check occurs. When certain `while` loops are included in the program, however, it may

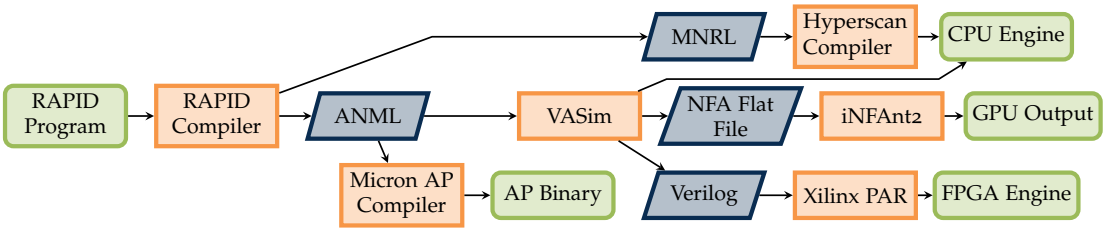


Figure 4.5: Supported pipelines for executing RAPID programs. RAPID programs can be executed on CPUs (using VASim or Hyperscan), GPUs (using iNFAnt2), FPGAs, and Micron's D480 AP. Rounded green boxes indicate the input and output of the pipeline. Orange rectangles are software tools used to generate intermediate and output files. Blue parallelograms are intermediate files generated by our pipeline.

not be possible to determine where in the input stream to inject the symbols. In these cases, we currently output a warning at compile time and rely on the developer to provide the pattern for inserting the control character into the data stream.

4.4 EXECUTING RAPID PROGRAMS

A primary goal of the RAPID programming language is to support cross-platform portability of pattern searching applications. This allows an application to be tested on a developer's machine, which might not contain high-performance hardware, and be easily deployed into a heterogeneous hardware environment. Finite automata provide a portable, intermediate computation form that can be ported to many hardware backends, including CPUs, GPUs, FPGAs, and Micron's

D480 AP. We achieve this by developing and adapting automata engines for each platform.

As discussed in Section 4.1, automata processing provides a suitable abstraction for efficient execution of applications across architectures. Such an approach effectively decouples high-level application development from low-level optimizations. Any advances in automata processing performance (e.g. new optimizations and new computational approaches) can be beneficial for all high-level applications.

In the previous section, we described the process for compiling a high-level RAPID program to finite automata. Now, we discuss workflows for executing automata across common computer architectures. Figure 4.5 outlines our workflow for targeting CPUs, GPUs, FPGAs, and the AP.

4.4.1 *Targeting the Automata Processor*

Micron provides a proprietary tool chain for converting ANML specifications into a loadable binary image for the AP. This tool places and routes the NFAs onto the hardware states and reconfigurable routing mesh of the processor. We use this tool directly to synthesize ANML for the AP.

4.4.2 *Targeting CPUs*

We have developed and collected a set of algorithms for optimizing and transforming finite automata. These algorithms are implemented in VASim, a tool we

created to facilitate automata research and experimentation [235]. This framework supports easy prototyping, debugging, simulation, and analysis of automata-based applications and architectures. We use VASim to optimize the automaton from the RAPID compiler using *common prefix collapsing* [28]. This process merges states that match the same input symbols, beginning with the starting states, producing a functionally equivalent NFA with fewer states. In our Brill tagging benchmark, for example, prefix collapsing results in a 57% reduction in the number of states. Additionally, VASim contains a multi-threaded simulation core, which is capable of executing automata on an input stream. The simulator was designed specifically to execute ANML files, making VASim an excellent candidate for a RAPID CPU backend.

While VASim is currently 4×–694× faster than existing simulation tools for Micron's AP, regular expression processors, such as Hyperscan [111] outperform VASim for pure NFA applications. When a compiled RAPID program contain no counters, we choose to execute with Hyperscan, using the compilation and runtime tools supplied as part of the MNCaRT ecosystem [12]. We instruct the RAPID compiler to emit MNRL and then use the Hyperscan compiler to generate a serialized *pattern dictionary* and perform Hyperscan-specific optimizations to the automata. We then execute the pattern dictionary against a supplied input stream using the *hsrun* tool provided with MNCaRT.

4.4.3 *Targeting GPUs*

We support the execution of pure NFAs with a GPU backend. RAPID programs that do not use counters can therefore be executed on GPUs. We use iNFAnt2, the optimized GPU-based NFA engine used by Wadden et al. with the ANMLZoo benchmark suite [231]. The iNFAnt2 engine reads in a transition table and uses individual SIMD threads to compute possible transitions on a given input symbol.

We use VASim to convert the ANML produced by the RAPID compiler to the transition tables needed by iNFAnt2. Similar to the CPU target, we optimize the ANML using VASim's optimization framework. Next, we output the NFA transition table using the Becchi-style format [27]. To execute on the GPU, we provide both this transition table and an input stream to iNFAnt2, which produces reporting output.

4.4.4 *Targeting FPGAs*

When targeting an FPGA, we first optimize the compiled automata and then convert to a hardware description using VASim. VASim transforms the optimized NFA into a Verilog hardware description. Our tool generates a module with inputs for clock, reset, and an 8-bit input symbol and outputs for report events. Within the module, activations of states in the automaton are stored in registers, which are updated on every clock cycle. A state becomes *active* if it is enabled (a state with an incident edge to the current state is active) and the current

input symbol matches. Using this update rule, it is possible to execute the NFA directly in hardware. Finally, we target Xilinx FPGAs by synthesizing the hardware description produced by VASim. Additional optimization of automata kernel generation for FPGAs using this same technique has been explored by Xie et al. [252].

4.5 EVALUATION

We evaluate RAPID against hand-crafted designs for five real-world benchmark applications, which were selected based upon previous research demonstrating significant acceleration using Micron's AP [40, 186, 239, 267]. We predominantly consider metrics related to *expressive power*, *scalability*, and *performance*. We consider the following research questions:

1. Do RAPID constructs allow for the representation of regular languages?
2. Do RAPID constructs generalize to pattern search problems across multiple problem domains?
3. Do RAPID programs require fewer lines of code than a functionally equivalent ANML program to represent a given pattern search problem?
4. Are RAPID programs no less efficient at runtime and during synthesis than hand-optimized ANML programs?

4.5.1 *Expressive Power*

We begin our evaluation of the RAPID language by demonstrating that regular expressions can be represented in our language. To do this, we will briefly sketch the RAPID constructs necessary to implement each of the rules detailed in Section 2.3.1. Because we consider a streaming model of computation, we do not consider empty strings (i.e., we assume there is input).

RAPID programs for singleton matches and empty set matches are trivial: a RAPID program with a single character comparison with `input()` followed by a `report` and a program with no `reports`, respectively. A regular expression union matches either one regular expression or another. In RAPID, this same behavior is achieved using an `either/orelse` statement where the blocks of the statement encode each expression in the union. Concatenation is similarly direct: statements and expressions for matching both expressions are written in sequence within the RAPID program.

Kleene closures are the most challenging to represent because they do not map naturally to the looping constructs in our language. However, we can leverage a macro reference to the body of the closure to provide the same semantics as a Kleene closure. After calling this macro, we use an `either/orelse` statement to both call the same instance of the macro or continue on to match the next portion of the pattern. Finally, we use a surrounding `either/orelse` statement to allow for zero matches of the body of the closure. We provide an example RAPID program for matching b^*c in Listing 4.5. While this construction is neither intuitive nor

```
macro b() {
  b == input();
}

macro bstar_c() {
  // create a reference to the body of the Kleene closure
  ref b_inst = b();

  either{
    b_inst;

    // this either/orelse creates the backwards loop to the
    // body of the Kleene closure
    either {
      b_inst;
    } orelse {
      // this empty block allows us to transfer
      // control past the Kleene closure
    }
  } orelse {
    // this empty block lets us match 0 instances
  }

  c == input();
  report;
}

network {
  bstar_c();
}
```

Listing 4.5: Example implementation of the regular expression b^*c in RAPID.

concise, we note that such a formulation never arose in our implementation of real-world applications for our evaluation in Section 4.5.2. The control structures in the RAPID language were designed to address pattern-matching paradigms found in real-world applications.

We note that embedding of regular expression operators into the RAPID language would provide a better solution in many cases; however, we have demonstrated that this is not necessary with respect to the expressive power of the language.

RAPID has sufficient expressive power to represent all regular expression operations.

4.5.2 *Empirical Evaluation*

Next, we conduct an empirical evaluation of the RAPID language. Table 4.4 provides descriptions of the benchmarks used. For each benchmark, we chose an instance size representative of a real-world problem. These sizes come either directly from previous work or from conversations with the authors of the previous work. The generation method column indicates the technique used to create the handcrafted code, which ranged from custom Java or Python programs for generating an ANML design to the use of a GUI design tool (Workbench) for crafting automata by hand. The authors of the *ARM* [239] and *Brill* [267] benchmarks provided us with their original code, including a collection of regular

Table 4.4: Description of benchmarks

BENCHMARK	DESCRIPTION	GENERATION METHOD	SAMPLE INSTANCE SIZE
ARM/FIS [239]	Association rule mining / Frequent itemset	Python + ANML	24 Item-Set
Brill [267]	Rule re-writing for Brill part of speech tagging	Java	219 Rules
Exact [40]	Exact match DNA sequence search	Workbench	25 Base Pairs
Gappy [40]	DNA string search with gaps between characters	Workbench	25-bp, Gaps $\leqslant 3$
MOTOMATA [186]	Fuzzy matching for bioinformatics planted motif search	Workbench	(17,6) Motifs

expressions for performing the *Brill* benchmark. We recreated the remaining designs, using algorithms and specifications published in previous work.

RAPID constructs generalize to a range of application domains for pattern-searching problems.

Table 4.5 lists design statistics for the benchmarks. We compare the lines of code needed to generate ANML 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. The regular expression representation for *Brill* is more compact than RAPID.

We created the *Gappy*, *Exact*, and *MOTOMATA* benchmarks using a GUI design tool. For these, we present the lines of code in ANML, which is roughly equivalent to the number of actions taken within the design tool. ANML file sizes are

dependent on the specific instance of a problem, and the numbers we present are for a single instance of the problem listed in Table 4.4. In all cases, the RAPID program is significantly more compact than the ANML it generates.

RAPID programs are significantly more concise to write than hand-crafted automata. Further, RAPID programs can also be more concise than automata-generator scripts.

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. The placement and routing tools modify the original automaton to better match the architectural design of the AP. These optimizations are similar to those applied by VASim for our CPU, GPU, and FPGA targets. For most benchmarks, RAPID-generated automata contain fewer device STEs, taking up less space on the device. Only the *Gappy* benchmark requires more device STEs. Although we could optimize the RAPID code to reduce the size of the generated automaton, we found that this more natural design, although larger, has comparable placement and routing efficiency. For *MOTOMATA*, the RAPID version requires approximately half the STEs of the hand-crafted version. The compiled RAPID version makes use of a saturating counter, while the handcrafted version uses positional encoding.

Due to the lock-step execution of automata on the AP, runtime performance of loaded designs is linear in the length of a given input stream. Therefore, we focus on evaluating the space efficiency of RAPID programs. In Table 4.6, we present

Table 4.5: Comparison between RAPID and hand-crafted code with respect to lines of code (LOC) and STE usage

BENCHMARK		LOC	ANML LOC	STES	DEVICE STES
ARM	H	118	301	79	58
	R	18	214	58	56
Brill	H	1,292	9,698	3,073	1,514
	R	688	10,594	3,322	1,429
	Re	218	–‡	4,075	1,501
Exact	H	–†	193	28	27
	R	14	85	29	27
Gappy	H	–†	2,155	675	123
	R	30	2,337	748	399
MOTOMATA	H	–†	587	150	149
	R	34	207	53	72

R – RAPID H – Hand-coded Re – Regular Expression

† The GUI tool does not have a LOC equivalent metric.

‡ No ANML statistics are provided by the regular expression compiler.

the performance of RAPID programs compared to hand-crafted ANML based on placement and routing statistics for the AP, using version 1.4-11 of the AP SDK to generate the placement and routing information. The total blocks column measures the number of routing matrix blocks[6] needed to accommodate the design; lower numbers represent a more compact design. STE utilization indicates the percent of used STEs within the routed blocks; high numbers indicate a design with fewer unused 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. 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 ANML, the final device binaries are more compact, using fewer resources on the AP.

We also evaluate the space efficiency of the FPGA engines our tools produce. We synthesize our designs for a Xilinx Kintex UltraScale XCKU060. Table 4.6 also lists the number of LUTs and registers needed to implement the hardware

6 blocks are a subunit of the hierarchical routing matrix found on the AP [75].

Table 4.6: 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		AP STES	AP MBRA	AP CLK	FPGA LUTS	FPGA REG
ARM	H	58	20.8%	1	73	76
	R	56	20.8%	1	83	65
Brill	H	1,514	65.4%	1	201	1483
	R	1,429	60.6%	1	358	1360
Exact	H	27	4.2%	1	6	25
	R	27	4.2%	1	28	27
Gappy	H	123	77.1%	1	73	123
	R	399	70.8%	1	52	399
MOTOMATA	H	149	75.0%	0.5	114	148
	R	72	75.0%	1	85	60

H – Handcrafted R – RAPID

description of the benchmark. Lower numbers indicate smaller footprints for the circuits, which allows for more widgets to be run in parallel on the FPGA. As with the AP results, RAPID programs do not incur significant space overheads on the FPGA. A complete timing analysis and comparison with other FPGA engines falls outside the scope of this work but is examined by Xie et al. [252].

Despite representing problems at significantly higher levels of abstraction than hand-optimized automata, RAPID programs do not incur significant hardware overheads once compiled.

4.6 CHAPTER SUMMARY

As data sets continue to grow in size, new hardware and software approaches are needed to quickly process and analyze available data. This chapter explores the viability of automata processing as an intermediate computational representation to support high-throughput processing across computer architectures. We present RAPID, a new language for defining pattern-matching algorithms. RAPID is motivated by pattern-recognition processors, such as the Automata Processor, which greatly accelerate pattern detection in streams of data, but lack easy-to-use programming models.

Automata processing allows for a developer to write a single application and execute on all common architectures. Further, our empirical evaluation demonstrates that automata optimizations maintain performance stability across CPUs, GPUs, FPGAs, and the AP.

RAPID raises the level of abstraction for programming pattern-recognition applications, resulting in clear, concise, maintainable, and efficient programs. We develop a notion of macros and networks, which we argue improve program maintainability. Additionally, RAPID provides parallel control structures to support common tasks in pattern-matching algorithms, such as sliding window searches. We present techniques for converting RAPID programs to finite automata that can be executed on CPUs, GPUs, FPGAs, and Micron's D480 AP. Although RAPID programs are written at a higher level of abstraction than current hand-crafted code, our evaluation indicates that RAPID programs have similar, if not better,

device utilization. RAPID therefore meets the requirements of *scalability* and *performance*.

Thus, in addition to supporting extant code (see Chapter 3), our programming model also allows developers to write new applications for hardware accelerators. Next, we will develop software maintenance tools, built atop our RAPID language, to help developers identify and fix bugs in their code.

CHAPTER 5

Interactive Debugging for High-Level Languages and Accelerators

The introduction of new domain-specific languages (DSLs), such as the RAPID language presented in Chapter 4, and the adoption of new accelerators both create challenges from a software maintenance standpoint. Developers may wish to port existing code to these new languages or rewrite algorithms to be better-suited for these new accelerators, tasks which can introduce new faults [264, 266]. For automata processing applications, these faults can be particularly difficult to localize. Developers may not observe abnormal behavior until processing large quantities of data (i.e., testing samples may not exhibit high coverage of corner cases). Extracting a smaller input for analysis from the large data set can be challenging or costly, since many pattern-matching algorithms perform a sliding-window comparison where the relevant piece of data is not known a priori. It is therefore desirable to support high-throughput data processing with the ability to interrupt accelerated program execution and transfer control to a debugging environment. As described in Section 1.2.3, CPUs are too slow for effective debugging of many automata-based applications and debugging on accelerators is currently conducted at extremely low levels of abstraction.

Therefore, we present an approach for building an interactive, source-level debugger using low-level signal inspection on hardware accelerators. Our debugging system includes support for breakpoints and data inspection. We demonstrate prototype implementations for both the AP and Xilinx FPGAs; no modifications to the underlying accelerators are needed. While we focus our presentation on one indicative DSL, the techniques we present 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 leverages four key insights:

- A co-designed hardware accelerator and CPU-software simulation system design allows for both high-speed data processing as well as interactive debugging.

- Micron's AP contains context-switching hardware resources, which are often left unused, for processing multiple input streams in parallel. Additionally, FPGA manufacturers provide logic analyzer APIs to inspect the values of signals during data processing. We repurpose these hardware features to transfer control from the execution context on the accelerator to an interactive debugger on the host system.

- Runtime state for automata processing applications is compact, consisting only of the set of active states. We lift this state to the semantics of the source-level program through a series of mappings generated at compile time. The mapping from source-level expressions to architecture-level automata

states is traceable within the RAPID compiler; our approach is applicable to any high-level programming language for which such a mapping from expressions to hardware resources may be inferred.

- Setting breakpoints on expressions in a program is not directly supported by the automata processing paradigm. Instead, we set and trigger breakpoints on input data, pausing execution after processing N bytes. We can leverage these pauses to provide the abstraction of more traditional breakpoints set on lines of code.

We also extend our basic design to support low-latency time-travel debugging near breakpoints by stopping accelerated computation early and recording execution traces with a software-based automata simulator. The addition of software simulation allows our system to support logical backward steps in the subject program near breakpoints without incurring significant delays while data is re-processed.

Capturing the state information from each automaton state on FPGAs incurs a hardware, performance, and power overhead, in contrast to the AP (where support is built into the architecture). We evaluate the *scalability* of our debugging approach on the ANMLZoo benchmarks [231] using the REAPR automata-to-FPGA tool [252] and a server-class FPGA. We were able to achieve an average of 81.70% of the baseline clock frequencies. We also discuss the trade-off between resource overheads and support for debugging.

We evaluate the *ease of use* of our debugging approach using an IRB-approved human study to understand how our technique affects developers' abilities to

localize faults in pattern-matching applications. During the study, we collected data using a set of ten programs indicative of real-world applications with a total of twenty seeded defects. Our human study included 61 participants with a wide range of programming experience, including a mix of undergraduate and graduate students at our home institution, as well as a professional developer. We found a statistically significant 22% increase ($p = 0.013$) in localization accuracy when participants were provided with debugging information generated by our system.

This chapter, therefore, makes the following contributions:

- A technique for interactive debugging of automata processing applications written in a high-level DSL. We leverage an accelerator to quickly process input data and repurpose existing hardware mechanisms to transfer control and initiate a debugging session.

- A characterization of breakpoint types for the automata processing domain. We differentiate between breakpoints set on input data and on expressions.

- An empirical evaluation of our debugging system on a Xilinx FPGA. We achieve an average of 81.70% of the baseline clock frequencies for the ANMLZoo benchmarks.

- A human study of 61 participants using our debugging tool on real-world applications. We observe a statistically significant ($p = 0.013$) increase in fault localization accuracy when using our tool.

In the remainder of the chapter, we first introduce our debugging system in Section 5.1. Then, we evaluate scalability on FPGAs in Section 5.2, and present the statistical analysis of our human subjects study in Section 5.3.

5.1 HARDWARE-SUPPORTED DEBUGGING

In this section, we present a novel technique for accelerating debugging tasks for sequential pattern-matching applications using a hardware-based automata processor. Our technique bridges the semantic gap between the underlying computation and the source-level RAPID program and can be extended to other languages whose compilers map program expressions and state to hardware resources. We consider two varieties of breakpoints (line and input) and describe how input-based breakpoints can be used in our system to implement traditional line-based breakpoints. We also extend our debugging system to support low-latency time-travel debugging by using a software-based automata simulator. While the technique generalizes to various automata processing architectures (including CPUs), we present the approach with respect to Xilinx FPGAs and Micron's D480 AP.

5.1.1 *Example Program*

Listing 5.1 provides an example RAPID program, which we will consider at various points in this chapter. The program matches the string "hello world".

```
macro helloWorld() {
  // match "Hello world" anywhere in the input stream
  whenever(  ALL_INPUT == input() ) {
    // match the word "Hello" in the input data stream
    foreach(char c : "Hello") {
      // match each character in turn
      // computation stops if a character doesn't match
      c == input();
    }

    // match with a space (' ') between the two words
    input() == ' ';

    // match with the word "world" in the input data stream
    foreach(char c : "world") {
      c == input();
    }

    // if we successfully matched everything, report
    report;
  }
}

network() {
  // instantiate a single search using the helloWorld macro
  helloWorld();
}
```

Listing 5.1: An example RAPID program that matches "hello world" anywhere in an input string

To do this, we instantiate a single instance of the `helloWorld` macro. This macro continually attempts to match our target string (line 3). To match the "hello world" string, we iterate over the characters in "hello", matching each in turn (lines 5–9). Then, we match a space (line 12), iterate over the characters in "world" (lines 15–17). If these characters are successfully matched, a report event is generated (line 20). For a detailed description of the keywords and operators in the RAPID language, please refer to Section 4.2.

5.1.2 *Breakpoints*

Breakpoints allow a developer to begin interacting with a debugger [124]. The subject program executes until a breakpoint is reached, and then control is transferred into an interactive session, allowing the user to inspect program state [149]. *Watchpoints*, or conditional breakpoints, are another common tool developers use to debug programs. Unlike breakpoints, a watchpoint only transfers control when the value of a variable changes or an assertion becomes true. Because watchpoints may be implemented as breakpoints [190], we focus solely on breakpoints in this work.

LINE BREAKPOINTS. Traditionally, breakpoints are set on lines of code, statements, or expressions in a program. Execution stops every time control reaches the corresponding program point. We refer to this type of breakpoint as a *line breakpoint*. In the example RAPID program in Listing 5.1, a line breakpoint could

be set on line 16 to halt execution for each match of a character in the sequence "world".

INPUT BREAKPOINTS. Automata-based pattern-recognition programs often process large quantities of data, and spurious or incorrect reports[1] may only appear after a significant portion of the input stream has been consumed. To debug these defects, a developer may wish to pause program execution after a given number of input symbols have been processed by all parallel searches. In other words, the developer might wish to set a breakpoint on the input stream given to an application. We refer to this type of breakpoint as an *input breakpoint*. This abstraction provides functionality similar to several automata simulators that support "jumping" to a given offset in input data.

5.1.3 *Hardware Abstractions for Debugging*

Unlike traditional (non-parallel) CPU debugging, we explicitly target a setting with a particular kind of parallelism, one where multiple pattern-matching searches and multiple automata states can be active simultaneously. Central to our technique is the ability to inspect the *active set*, or currently active states, in the executing automata. On both the AP and FPGA, this information is tracked using the activation bit stored within each STE (see Section 2.2), and we refer to this collection of data as the *state vector*. The state vector provides a complete and

1 False negatives (missing reports) remain an open challenge.

compact snapshot of machine execution after processing a given number of input symbols (in NFAs, there is no other notion of "memory" such as a stack or tape).

5.1.4 *Accessing the State Vector*

To support our debugging system, a target hardware platform must provide access to the state vector of the executing automata. We describe accessing this vector on both the AP and Xilinx FPGAs; no modifications or additions to the hardware platform are needed to support these techniques.

MICRON'S AP. Off-chip access to the state vector is provided through the context switching cache on the AP [75]. This cache was developed to allow automata executing on the AP to switch between—and process in parallel—several input streams. Additionally, the AP runtime allows the host system to inspect the contents of the context switching cache. We repurpose this hardware to transfer control to the interactive debugging session: when a breakpoint is reached, our debugger captures the state vector from the executing automata and copies the values back to the host system.

XILINX FPGA. We consider two approaches to accessing the state vector on Xilinx FPGAs: integrated logic analyzers (ILAs) [109] and virtual IOs (VIOs) [229]. Both of these Xilinx IP (Intellectual Property) blocks are used for runtime debugging the FPGA and come with different design trade-offs [230].

The ILA is a signal-probing core that can be used to monitor a hardware design's internal signals by attaching logical *probes* to these signals. It supports advanced, dynamically configurable triggering conditions that specify when the ILA captures data. This functionality allows the developer to trigger data capture on complex hardware events represented by a combination of signals. ILAs use block RAM to probe the internal design signals at the clock speed of the design under test but have a fairly high hardware utilization cost. For our application, ILAs allow us to dynamically specify breakpoint triggering conditions while having a negligible impact on the data throughput of automata being debugged.

VIOs are similar to the ILAs, allowing logical probes to sample data within a target design but without the advanced triggering functionality. Consequently, VIOs are more compact than ILAs while still providing the needed access to data in automata state vectors. Because they are instantiated within the design and are synchronous with the design, VIOs can result in reduced design clock speeds.

While ILAs provide a richer set of features with little impact on clock frequency, we found that the space requirements needed to interface with automata processing designs frequently exceeded the capacity of FPGAs for indicative applications. In particular, ILAs for our debugging system require more BRAM resources than our server-class FPGA made available. Therefore, we choose to implement our debugging system using VIOs, which require fewer hardware resources, but may reduce clock frequencies. Our empirical evaluation (see Section 5.2) demonstrates that these reductions are less than 20% for most automata applications.

We extend Xie et al.'s REAPR (see Section 2.2.3) to automatically generate VIOs or ILAs attached to the activation bits of STEs for a given automaton. Applications built with automata often consist of tens of thousands of states (see Table 5.1), but the current VIO implementation provided by Xilinx only supports 256 individual probes, and ILAs are limited to 1024. To address this dichotomy in scale, we increase the *width* of each VIO probe, treating a set of N STEs as a single, multi-bit value. Once the state vector data is transferred to the host system, we disambiguate the individual STEs. For a probe width of 256 (the maximum supported width), our technique is able to monitor a total of $256 \times 256 = 65,536$ STEs with a single VIO; multiple VIOs may be used for larger designs. We greedily assign STEs to VIO probes in the order STEs are encountered in an input automaton. A more sophisticated graph analysis (e.g., calculating connectivity of states) could result in probe assignments that reduce final placement and routing overheads. We leave exploration of such optimizations to future work.

OTHER PROCESSORS. Other processors may be used in place of the AP in our debugging system as long as the state vector abstraction is exposed. For example, inspection of the state vector for some CPU-based automata processors (e.g., VASim [235]) requires iterating through all states in the automaton to capture the active set. Other custom accelerators for automata processing, such as the Cache Automaton [209], also provide direct support for accessing the state vector.

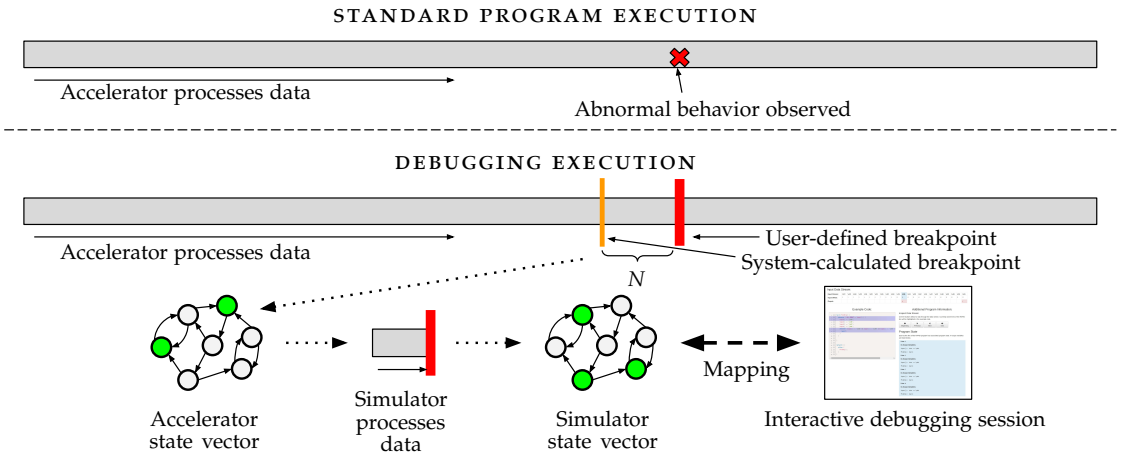


Figure 5.1: An example debugging scenario. While executing the RAPID program, abnormal behavior is observed deep into processing data. The user sets an input breakpoint, and the debugging system sets an input breakpoint N symbols prior for low-latency time-travel support. Data is processed on the hardware accelerator until the input breakpoint is reached, the state vector is exported, and the final N symbols are processed using a software automata simulator. The resulting state vector is then lifted to the semantics of the user-level RAPID program and control is transferred to the interactive debugging session.

5.1.5 *Hardware Support for Breakpoints*

A typical use case for our debugging system begins with developers observing abnormal behavior during the execution of a RAPID program. They then set a breakpoint that triggers near the abnormal behavior and re-execute the program. When the breakpoint is reached, runtime state is transferred to the host system, lifted to the semantics of the source-level RAPID program, and control is transferred to an interactive debugger. An overview of this process is given in Figure 5.1. In this subsection, we describe the steps needed to trigger a breakpoint

on an automata processing engine. We first consider input breakpoints, and then we describe how line breakpoints may be transformed into input breakpoints.

Input breakpoints are implemented through partitioning of the input data stream. We split the data such that the input stops at the offset of the desired breakpoint and process this using the AP. When processing completes, we export the state vector of the executing automata to the host system.

Line breakpoints in source-level RAPID programs cannot be directly implemented in the underlying AP or VIO-based FPGA hardware platforms. The automata processing paradigm only generates reports; there is no notion of a program counter or *printf*-like behavior that we can leverage.

We thus use reports to map line breakpoints to input breakpoints by recording the offsets at which the NFA states associated with a RAPID statement or expression (determined during compilation) are active while processing the input data. This is achieved by compiling two distinct sets of automata from an input RAPID program. One set of automata (machine A) perform computation as normal. The second set (machine B) report whenever selected lines of code execute. We modify the RAPID compiler to emit machine B. Given a set of line numbers, the modified compiler removes all previous reporting states and instead configures STEs associated with the given lines to report. By processing data with machine B, we identify the input stream offsets at which breakpoints are triggered. Processing the input data a second time with machine A allows our system to capture relevant hardware state and trigger input breakpoints at offsets discovered with machine

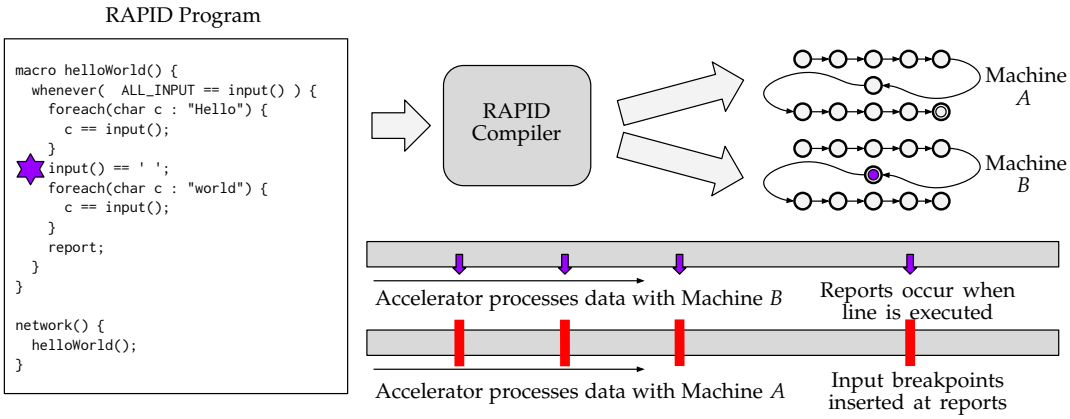


Figure 5.2: Transformation of a line breakpoint to an input breakpoint. Reports generated by STEs mapped to RAPID expressions determine input breakpoints.

B. Updating or selecting new line breakpoints requires regenerating machine *B*. This transformation is illustrated in Figure 5.2.

While the double compilation and execution steps do incur a minimum of a 2× overhead[2] for line breakpoints over execution containing no line breakpoints, we note that *current hardware supports this approach*. A more efficient approach would be to support hardware-based debugging signals. On a straightforward modification of the AP, these could be implemented similar to reporting events, serving a similar role as a hardware break- or watch-point in a general-purpose CPU [190]. Breakpoint signals are supported on FPGA-based automata processing engines using ILAs to capture the state vector; however, space overheads are currently too significant for use with most real-world applications.

2 Naively, processing of the input stream twice approximately doubles the execution time. However, this does not consider the additional time needed to compile a second automaton, reconfigure the AP or FPGA, or process reporting events.

5.1.6 *Debugging of RAPID Programs*

After capturing of the state vector, our system lifts the underlying state to the semantics of the input RAPID program. Our approach is similar to traditional CPU debugging, in which processor state is mapped to expressions in the input program using lookup tables generated at compile time [190].

We augment the RAPID compiler to produce a *debugging automaton*, $(Q, \Sigma, \delta, S, F, id, d)$. The additional term, d, is a mapping from NFA states to RAPID source locations and known program variable state. RAPID employs a staged computation model (Section 4.3); the values of some variables are resolved at compile time and are known at the time of NFA state generation. These are stored in the mapping.

Compilation for the AP transforms an input automaton to a configuration for the processor's memory array and routing matrix (see Section 2.2), and compilation for the FPGA maps an automaton to LUTs and FFs. These compilation processes may result in multiple states being mapped to a single hardware location (state merging) or a single state being mapped to multiple hardware locations (state duplication) as a result of optimizations to better utilize available hardware resources (cf. debugging with optimizations [101]). The compiler also produces a mapping, *loc*, from hardware locations to automaton state IDs. This debugging technique can be directly extended to any underlying automata processing engine that can provide this location mapping.

When an STE-level breakpoint is triggered, we determine the corresponding location(s) in the original RAPID program by calculating

$$\bigcup_{q \in Q_{active}} d(id(loc(q)))$$

where Q_{active} is the set of active states extracted from the state vector. Due to the inherent parallelism in RAPID programs, the locus of control may be on several statements in the program simultaneously. Our technique for lifting the underlying program state of the automata processing core to the semantics of the RAPID program therefore returns a minimal set of the currently executing RAPID statements.

5.1.7 *Time-Travel Debugging*

Many debuggers provide the ability to step backward in a program, a functionality often referred to as time-travel debugging [126]. This feature is beneficial for automata-based applications to find the start of a spuriously matched sequence. To step backward in the source-level RAPID program or data stream, our debugger would have to reprocess the input data, leading to high latency when breakpoints are set deep in the data stream. We now describe a modification to our system that significantly reduces this overhead.

When triggering input breakpoints, our debugging system splits the input stream N bytes (symbols) before the user-specified location (rather than splitting

the data at the specified input offset). Once the input has been processed, we export the current state vector like before and have access to the state vector N bytes before the user's breakpoint.

We then load the automata into a modified version of VASim [235], a CPU-based automata execution engine. We have modified VASim to record and output state vectors similar to those produced by the AP and FPGA.[3] We then execute the final N bytes before the breakpoint using VASim and save the state vector. For the N bytes before the breakpoint, our system has low-latency access to the execution state that is lifted to the semantics of the source-level RAPID program. This allows a developer to step forward and backward near a breakpoint with minimal processing delay.

In our initial implementation, we choose to stop processing on the accelerator 50 bytes (symbols) before the actual breakpoint. We find that this provides suitable time travel without incurring significant slow-downs; however, a complete sensitivity analysis is beyond the scope of this work.

5.2 FPGA EVALUATION

In this section, we present the results of an empirical evaluation of our FPGA-based debugging system. Our evaluation focuses on the overheads of debugging support. We repurpose existing hardware on the AP for debugging, and therefore do not introduce additional overhead. Thus, we focus our evaluation on the space

3 Modified version available at `https://github.com/kevinaangstadt/VASim/tree/statevec`.

and time overheads incurred for the additional FPGA hardware needed in our system. Our goal is to characterize the *performance* and *scalability* of our framework. We consider the following research questions:

1. What percent of baseline (standard execution) clock frequency can applications achieve when synthesized with our debugging hardware?

2. Are applications synthesized with debugging hardware able to fit within the resource constraints of server-class FPGAs? How many passes over the data are needed when an application cannot fit?

5.2.1 *Experimental Methodology*

We evaluate our prototype automata debugging system on a server-grade Xilinx FPGA using the ANMLZoo automata benchmark suite, which consists of fourteen real-world-scale finite automata applications and associated input data [231]. The benchmarks are varied, including both regular expression-based and hand-crafted automata. We present a summary of the applications in Table 5.1, including the number of states in each benchmark as well as the average degree (number of incoming and outgoing transitions) for each state. The higher the degree, the more challenging the benchmark is to map efficiently to the FPGA's underlying routing network.

For each benchmark, we generate an FPGA configuration using our modified version of REAPR [252], producing Verilog including both VIOs (for capturing

Table 5.1: ANMLZoo benchmark overview

BENCHMARK	FAMILY	STATES	AVG. NODE DEGREE
Brill	Regex	42,658	1.03287
ClamAV	Regex	49,538	1.00396
Dotstar	Regex	96,438	0.97396
PowerEN	Regex	40,513	0.97601
Protamata	Regex	42,009	0.99110
Snort	Regex	69,029	1.08831
Hamming	Mesh	11,346	1.69672
Levenshtein	Mesh	2,784	3.26724
Entity Resolution (ER)	Widget	95,136	2.28372
Fermi	Widget	40,783	1.41176
Random Forest (RF)	Widget	33,220	1.00000
SPM	Widget	100,500	1.70000
BlockRings	Synthetic	44,352	1.00000
CoreRings	Synthetic	48,002	1.00000

state) and also Wadden et al.'s reporting architecture [233] for efficient transfer of reports to the host system. We also use REAPR to generate a baseline configuration that does not include the VIOs.

We synthesize and place-and-route each application for an Alphadata board rev 1.0 with a Xilinx Kintex-Ultrascale xcku060-ffva1156-2-e FPGA using Vivado 2017.2 on an Ubuntu 14.04.5 LTS Linux server with a 3.70GHz 4-core Intel Core i7-4820K CPU and 32GB of RAM. As of 2019, this configuration represents a high-end FPGA on a mid-range server. For both the baseline and our version supporting

debugging, we measure the hardware resources required, the maximum clock frequency and the total power utilized. We present these results next.

5.2.2 *FPGA Results*

Performance results for FPGA-based debugging are presented in Table 5.2. We were able to successfully place and route thirteen of the fourteen benchmarks—the Xilinx toolchain fails with a segmentation fault for one of the synthetic benchmarks. We limit our discussion to these thirteen benchmarks.

Entity Resolution, Snort, and SPM require two VIOs due to the number of states in the automata. Nonetheless, all but Entity Resolution and SPM—our two largest benchmarks—fit within the hardware constraints when synthesized with debugging hardware. We support these two benchmarks by partitioning the automata. Most applications in ANMLZoo, including these two, are collections of many small automata or rules. By splitting the applications into two pieces, we still support debugging on an FPGA, but throughput is halved if run serially on a single FPGA. The numbers presented in Table 5.2 include this overhead.

Our additional debugging hardware has average LUT and FF overheads of 2.82× and 6.09×, respectively. The overheads vary significantly between applications, and we suspect that this is due to aggressive optimization during synthesis. The area overhead of state capture is unknown in the AP (area details for structures are not published), but since it is provided for context switching, using it for debugging incurs no extra hardware cost. For FPGAs, the area overhead of

Table 5.2: FPGA-Based debugging system performance results

BENCHMARK	WITHOUT DEBUGGING				WITH DEBUGGING				NUM. VIOS	LUT OVERHEAD	FF OVERHEAD	PERCENT ORIG. FREQ.	POWER OVERHEAD
	LUTs	FFs	Clock (MHz)	Power (W)	LUTs	FFs	Clock (MHz)	Power (W)					
Brill	27,621	27,782	166.67	0.817	89,605	169,323	166.67	1.973	1	3.24	6.09	100.00%	2.41
ClamAV	42,178	42,067	204.08	0.923	95,891	199,336	121.95	1.257	1	2.27	4.74	59.75%	1.36
Dotstar	49,774	46,965	169.49	0.938	172,350	372,074	142.86	2.622	1	3.46	7.92	84.29%	2.80
PowerEN	35,359	31,530	163.93	0.832	77,900	161,156	149.25	1.302	1	2.20	5.11	91.05%	1.56
Protamata	49,791	36,285	126.58	0.838	85,604	167,646	108.70	1.206	1	2.10	4.62	85.87%	1.44
Snort	43,061	28,047	98.04	0.783	128,684	266,600	91.74	1.478	2	2.99	9.51	93.58%	1.89
Hamming	5,602	6,637	312.50	0.701	25,170	46,080	312.50	1.065	1	4.49	6.94	100.00%	1.52
Levenshtein	2,538	2,242	434.78	0.666	4,218	11,263	400.00	0.737	1	1.66	5.02	92.00%	1.11
ER*	50,349	47,102	212.77	1.066	21,3461	38,1258	56.82	1.447	2	4.24	8.09	26.70%	1.36
Fermi	36,314	32,261	116.28	0.991	86,879	167,089	99.01	1.537	1	2.39	5.18	85.15%	1.55
RF	31,060	25,769	200.00	0.990	66,686	130,007	192.31	1.611	1	2.15	5.05	96.15%	1.63
SPM*	64,615	59,106	126.58	1.017	225,315	406,241	60.24	2.605	2	3.49	6.87	47.59%	2.56
BlockRings	44,446	44,185	256.41	1.215	90,333	178,905	256.41	2.119	1	2.03	4.05	100.00%	1.74
CoreRings†	–	–	–	–	–	–	–	–	–	–	–	–	–
									AVERAGE	2.82	6.09	81.70%	1.76

* Benchmark must be partitioned to fit within FPGA resource limits with added debugging. The clock frequency reflects this partitioning.
† The current commercial Xilinx toolchain terminates with a segmentation fault during synthesis.

our approach is 2–3× for LUTs (except for Hamming) and 5–10× for FFs. This area overhead is high. For complex programs, the compiled automata may need to be partitioned, which is straightforward and supported by our infrastructure. However, partitioning requires either running multiple passes over the input (end-to-end latency increases as passes are added) or using multiple FPGAs (increasing hardware costs, but as of August 2018 cloud computing providers offer instances with up to eight FPGAs[4] for $13.20 an hour[5]). We believe this is a small price to pay for debugging support: any extra costs (e.g., FPGA overheads) are small compared to the value of a programmer's time, and the presence and quality of debugging support can increase accuracy (see Section 5.3) and reduce maintenance time (e.g., [171, Sec. 5.1]). Lowering the area cost, either via more selective state monitoring or more optimized synthesis, remains future work.

Adding VIOs to a design can reduce operating clock frequencies (see Section 5.1.4) and increase power usage. For our benchmarks, the average power overhead is 1.76×, and we are able to achieve an average of 81.70% of the baseline clock frequencies. Even with the partitioned automata, the throughput of our prototype remains at least an order of magnitude greater than the throughput reported by Wadden et al. for a CPU-based automata processing engine [231]. Therefore, we expect our FPGA-accelerated system to provide better performance than a CPU-only approach.

4 https://aws.amazon.com/ec2/instance-types/f1/
5 https://aws.amazon.com/ec2/pricing/on-demand/

Despite high resource overheads, our debugging system achieves an average of 81.70% of the baseline clock frequencies for all benchmarks. Our system remains an order of magnitude faster than a CPU-based automata processing engine.

5.3 HUMAN STUDY EVALUATION

In this section we evaluate our debugging system using a human study by presenting participants with code snippets and asking them to localize seeded defects. We measure their accuracy and the time taken to answer questions. This section characterizes our study protocol and participant selection and presents a statistical analysis of our results.

5.3.1 *Experimental Methodology*

Our IRB-approved human study[6] was formulated as an online survey that presented participants with a sequence of fault localization tasks. Participants were provided with a written tutorial on the RAPID programming language and sample programs. These resources were made available to the participants for the duration of the survey. We presented each participant with ten randomly selected and ordered fault localization tasks from a pool of twenty. For each task, participants were asked to identify faulty lines in the code and justify their answer. We

6 University of Virginia IRB for Social and Behavioral Sciences #2016-0358-00.

recorded the participants' responses and the total time taken for each question. Participants were given the opportunity to receive extra credit (for students) and enter a raffle for a $50 gift certificate.

Each fault localization task consisted of a description of the program and fault, the code for the program with a seeded defect, and an input data stream. On half of the tasks, our debugging information was also displayed. The description of the program detailed the purpose of the presented code and also provided the expected output. Code for each task ranged from 15–30 lines and was based on real-world use cases [231]. Similar to GPGPU programs, RAPID programs accelerate a kernel computation within a larger program. While our selected programs are relatively small in terms of line count, they are both complete and also indicative: we adapted automata processing kernels to RAPID programs from various published applications, such as Brill tagging [267], frequent subset mining [239], and string alignment for DNA/Protein sequencing [40, 186]. We seeded a variety of defects into the code for our fault localization tasks, based on RAPID developer mistakes discovered by our initial study of RAPID in Chapter 4. When provided, the debugging information included buttons to step forward and backward in the data stream. For a given offset in the input stream, our tool highlights lines of code corresponding to the current locus of control. We also provided variable state information for each of the loci. Figure 5.3 provides an example fault localization task presented to survey participants.

Participants were all voluntary and predominantly from the University of Virginia. We advertised in Data Structures, Theory of Computation, and Program-

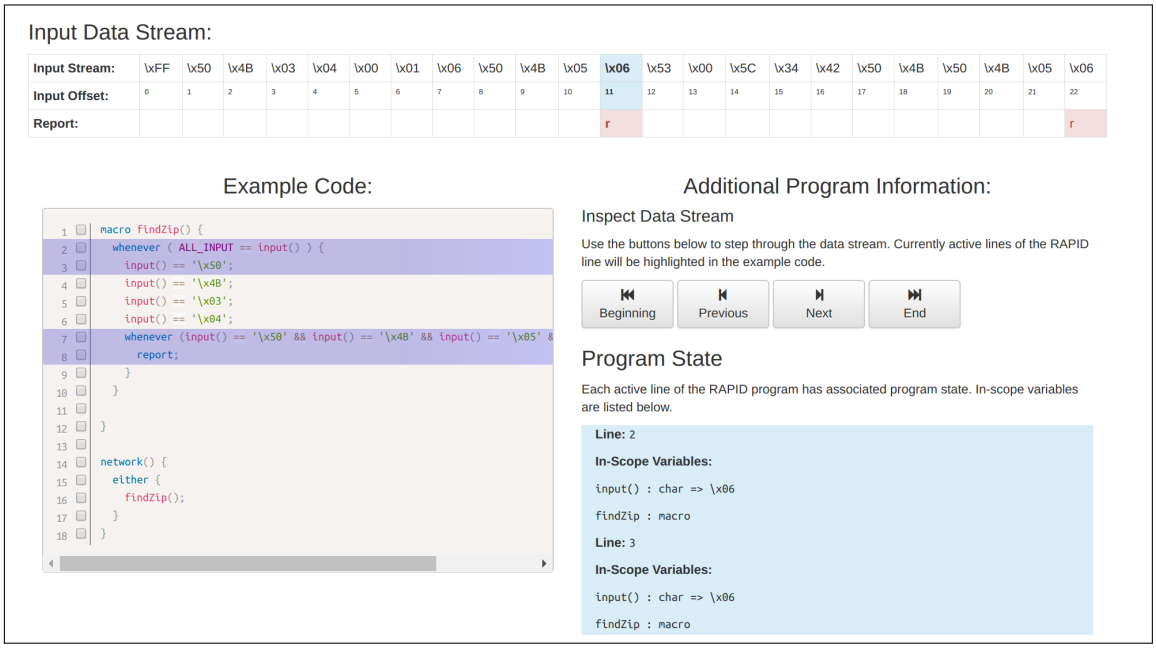


Figure 5.3: A question from the human study including generated debugging information. Task text and program state information are elided for space.

ming Language undergraduate CS courses, in a graduate software engineering seminar, and to members of the D480 AP professional development team. Participants are enumerated in Table 5.3.

5.3.2 *Statistical Analysis*

Next, we present statistical analyses of the responses to our human study with an eye toward understanding the *ease of use* of our debugger. We address the following research questions:

1. Does our technique improve fault localization accuracy?

Table 5.3: Participant subsets and average accuracies. The study involved $n = 61$ participants. Average completion times are for individual fault localization tasks.

SUBSET	AVERAGE TIME (MIN.)	AVERAGE ACCURACY	PARTICIPANTS
All	8.17	50.3%	61
Intermediate Undergraduate Students	7.3	49.2%	37
Advanced Undergraduate Students	10.14	50.0%	21
Grad Students and Prof. Developers	5.07	66.7%	3

2. Is there an interaction between programming experience and ability to interpret RAPID debugging information?

In total, 61 users participated in our survey each completing ten fault localization tasks, resulting in over 600 individual data points. Table 5.3 provides average accuracy rates and task completion times for subpopulations in our study.

Does our debugging information improve fault localization in RAPID programs?

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 debugging information ($p = 0.013$).

Mean accuracy increased from 45.1% to 55.1%, meaning participants were 22% more accurate when using our tool.

Fault localization improvements can be difficult to evaluate: researchers must be careful to avoid simply reporting the fraction of lines implicated [171, Sec. 6.2.1] rather than the actual impact on developers. Independent of time, accuracy is important because even in mature, commercial projects, 15–25% of bug fixes are incorrect and impact end users [260]. The improvement in accuracy provided by our information is modest but significant and is orthogonal to other approaches.

Our debugging tool improves a user's fault localization accuracy for RAPID programs in a statistically significant manner ($p = 0.013$).

Is there an interaction between programmer experience and our tool?

Previous studies (cf. Parnin and Orso [171]) have found that the effectiveness of debugging tools can vary with programmer experience. We examined our data for similar trends. Following an established practice from previous software engineering human studies (e.g., Fry and Weimer [86]), we partitioned our data between experienced (students in final-year undergraduate electives or above) and inexperienced (students not yet in final-year undergraduate classes) programmers. Such a partitioning likens final-year undergraduates to entry-level developers. To measure the interaction between programmer experience and our debugging tool, we used Aligned Rank Transform (ART) analyses. This technique allows us to perform factorial nonparametric analyses with repeated measures (such

as the interaction between experience and debugging information in our study) using only ANOVA procedures after transformation [249]. We found that there was no statistically significant interaction between experience and our debugging tool with respect to either accuracy ($p = 0.92$) or time ($p = 0.38$). This suggests that novices and experts alike benefit from our tool. Due to the limited number of professional developers in our initial study, we leave investigation of further partitions for future work.

There is no statistically significant interaction between experience and the ability to interpret our debugging information: both novices and relative experts benefit.

5.3.3 *Threats to Validity*

Our results may not generalize to industrial practices. In particular, our selection of benchmarks may not be indicative of applications written by developers in industry. We attempt to mitigate this threat by selecting a diverse set of applications from common automata processing tasks [231].

One threat to construct validity relates to our analysis of expertise. A different partitioning of participants into inexperienced and experienced programmers (i.e., a different definition of expertise) could yield different results; however, testing multiple partitions requires adjustment for multiple analyses. Additionally, our study recruited predominantly undergraduate students. A more balanced

participant pool may also provide additional insight into the interaction between expertise and debugging information in automata processing applications. We leave a larger-scale study including more professional developers for future work.

5.4 CHAPTER SUMMARY

Debuggers aid developers in quickly localizing and analyzing defects in source code. We present a technique for extending interactive debugging, including breakpoints and variable inspection, to the domain of automata processing. We describe the mappings needed to bridge the gap between the state of the executing finite automata and the semantics of a high-level programming language. We focus on the RAPID DSL, but our approach to exposing state from low-level accelerators lays the groundwork for more general support. Our system provides high-throughput data processing before transferring control to a debugger at breakpoints by executing automata on either Micron's D480 AP or a server-class FPGA. Only one bit of information per automata state at a given breakpoint must be copied to the host to support an interactive debugger. For FPGAs, we automatically generate custom logic, leveraging virtual IO ports, and capture state information from the executing automata. On the AP, we leverage built-in context switching hardware.

We achieve an average of 81.70% of the original clock frequency across 13 benchmarks while supporting interactive debugging. Despite high resource overheads, our system provides a valuable tool for debugging at a level of abstraction higher

than hardware signals. Reducing these overheads with, for example, static or dynamic analyses and innovative hardware, remain open challenges for future work.

To analyze the utility of our debugging system, we conducted a human study of 61 programmers tasked with localizing faults in RAPID programs. We observed a statistically significant 22% increase ($p = 0.013$) in accuracy from our tool's debugging information and found that our tool helps both novices and experts alike.

In summary, our debugging framework provides the *performance* and *scalability* afforded by hardware accelerators while improving *ease of use* by aiding developers in locating the source of bugs programs written for these accelerators. This concludes our development of front-end programming tools. In the next chapter, we consider architectural back-ends to support additional high-level applications.

CHAPTER 6

Architectural Support for Automata-Based Computation

We now shift away from the development of software tools and instead focus on building out additional architectural support for automata-based computation. Current architectures have been demonstrated to be suitable for a plethora of application domains [183, 184, 186, 220, 221, 232, 238, 240, 267]; however, these architectures do not support all applications. In this chapter, we consider two case studies—detection of security attacks and parsing of data—to develop both new system integrations of automata architectures as well as expanding the expressive power of this hardware.

As described in Section 2.5, two trends point to the need for robust, low-overhead detection of novel attacks: (1) the advent of attacks that exploit architectural vulnerabilities, such as Spectre [130] or Meltdown [142], and (2) the widespread use of embedded systems intended to run a set of authorized programs but vulnerable to the injection of unauthorized code [63, 65]. These problems, especially architectural vulnerabilities, are not easily and efficiently mitigated with software patches. Thus, there is a need for solutions that can be deployed with minimal modification to existing hardware, that impose min-

imal overhead on running software (cf. disabling hardware features to defeat attacks [150, 237]), and that generalize to detect novel attacks.

In addition to considering security applications that are integral to most computing platforms, we also address processing of tree-structured or recursively nested data, which 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. Further, the grammatical structure of English text is tree-like in nature [56]. Studies on data processing and analytics in industry demonstrate both increased rates of data collection and also increased demand for real-time analyses [62, 67, 211]. Therefore, scalable and high-performance techniques for parsing and processing data are needed to keep up with industrial demand. Unfortunately, parsing is an extremely challenging task to accelerate and falls within the "thirteenth dwarf" in the Berkeley parallel computation taxonomy, which characterizes important classes of computation [20]. Software parsing solutions often exhibit irregular data access patterns and branch mispredictions, resulting in poor performance (see Section 2.5.1). Custom accelerators exist for particular parsing applications (e.g., for parsing XML [68]), but do not generalize to multiple important problems.

To tackle these two challenges, we develop two new automata-based architectures. We choose to implement both of these architectures in the Last Level Cache (LLC) of a CPU, which has two primary advantages. First, we are able to reuse and repurpose existing hardware elements in the CPU, and second, embedding

these architectures in the CPU enable low-latency, tightly coupled execution with other CPU-based processing. We briefly describe each architecture in turn.

First, we present MARTINI,[1] a low-overhead, hardware-assisted anomaly-based intrusion detection framework that detects anomalous and malicious program execution at the memory access level, including cache side-channel attacks (Section 6.1). MARTINI extends earlier behavior-based IDSs that typically operate at the software level by focusing on memory access patterns (cf. system calls), representing them in a way that is implementable in hardware with negligible run-time overhead. In MARTINI, authorized behavior is modeled with *dictionaries* that represent an n-gram, or sliding window, of short sequences of memory accesses, where each memory access is compressed into eight bits of information. Because MARTINI uses n-grams rather than complex pattern matching, once the dictionary is trained on indicative, authorized behavior, subsequent queries can be formulated in terms of finite automata inputs. Thus, MARTINI can be deployed in hardware with low overhead and latency by leveraging near-memory processing and in-cache computation (Section 6.3). We develop a new functional unit with a custom data path that can be deployed in the processor core or Last Level Cache of modern CPUs, which admits real-time monitoring of memory accesses.

Next, we present ASPEN,[2] for efficient parsing of data (Section 6.4). Our key insight is that many parsing applications can be modeled using deterministic pushdown automata (DPDA) as defined in Section 2.1.2. ASPEN implements a DPDA processing engine in LLC, and our design is based on the insight that

1 MARTINI = Memory Address Representation To INfer Intrusions.
2 ASPEN = Accelerated in-SRAM Pushdown ENgine

much of the 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 is 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 support direct adaptation of a large class of legacy parsing applications, we implement a compiler for converting existing grammars for common parser generators to DPDAs executable by ASPEN (Section 6.2). We develop two key optimizations for improving the runtime of parsers on ASPEN, which work together to reduce stalls in input symbol processing (Section 6.2.2.3).

Our evaluations of MARTINI and ASPEN measure the *expressive power*, *scalability*, and *performance* of each design. We evaluate MARTINI with respect to two benchmark suites and four recent exploits, finding an overall false positive rate of 4.4% with a true positive rate of 100% (area-under-curve = 0.9954). In total, we consider more than 2,400 program traces and more than 13 billion individual memory accesses. We evaluate the expressive power of ASPEN by compiling parsers for four different languages to demonstrate that our architecture supports common data formats and that the resulting pushdown automata fit within the hardware resources of the architecture. We evaluate the performance of ASPEN on a benchmark suite of 23 XML files, observing that our approach is 14× faster than a state-of-the-art software-based XML parser.

In summary, this chapter presents the following scientific contributions:

- MARTINI, an approach for detecting unauthorized program behavior, including architectural side-channel attacks, using dictionaries of n-grams of memory accesses. We develop a system integration of an automata processing architecture to provide per-cycle monitoring of memory accesses.

- ASPEN, a scalable execution engine which re-purposes LLC slices for DPDA acceleration. We design a custom data path for DPDA processing using SRAM array lookups. ASPEN implements state matches, state transition, stack updates, includes efficient multipop support, and can parse one token per cycle.

- An optimizing compiler for transforming existing language grammars into DPDAs. Our compiler optimizations reduce the number of stalled cycles during execution. We demonstrate this compilation on four different languages: Cool (object-oriented programming), DOT (graph visualization), JSON, and XML.

- An empirical evaluation of ASPEN on a tightly coupled XML tokenizer and parser pipeline. Our results demonstrate an average of 704.5 ns per KB parsing XML compared to 9983 ns per KB in a state-of-the-art XML parser across 23 XML benchmarks.

- An empirical evaluation of MARTINI's classification accuracy on over 2,400 program traces from two large benchmark suites and four exploits, including

Spectre and Meltdown proofs-of-concept. We find that Martini is able to classify intrusive activity with high accuracy and precision (AUC 0.9954) while requiring a very small chip area. Moreover, deploying Martini in hardware would enable classification without runtime overhead.

We organize the remainder of this chapter as follows. In Section 6.1, we describe our approach for detecting malicious program behavior by monitoring memory accesses. Then, we describe the process of transforming parsing applications into pushdown automata computation in Section 6.2. Next, we describe the architectural designs of Martini and ASPEN in Section 6.3 and Section 6.4, respectively. We then describe our unified experimental methodology in Section 6.5 and perform architectural evaluations in Section 6.6. Following the architectural evaluation, we present application-specific evaluations of Martini in Section 6.7 and of ASPEN in Section 6.8.

6.1 detecting attacks with memory accesses

In this section, we present an application-level design and implementation of the Martini framework. We present the micro-architectural design in Section 6.3.

6.1.1 *The Memory Access Pattern Abstraction*

Many abstractions have been proposed to compactly characterize program behavior, including the cadence of cache misses [254], program counters [192], taint tracking in I/O inputs [210], and hardware performance counters [69, 197]. Despite demonstrating their ability to accurately identify anomalous behavior, these models often require intrusive instrumentation that degrades system performance. We aim for an abstraction of memory access patterns that is suitable for low-overhead, high-accuracy real-time monitoring.

Programs are represented as data stored in memory, and program execution proceeds by reading, modifying, and storing data in memory. Program behavior therefore partially manifests as a sequence of memory accesses produced during execution. These sequences are inherent to the underlying execution path and structure of program code. That is, alterations to the way in which a program processes information are revealed by its memory access patterns. For example, a calculator program that parses and interprets expression strings will generate distinct memory traces when multiplying vs. adding operands due to variations in the execution's control flow. By contrast, changing the operands (i.e., the numerical data) will typically not result in a change in the trace of memory addresses. The Martini design leverages this insight to characterize program execution as either benign or malicious (i.e., either authorized or unauthorized by the system operator) in a way that, ideally, generalizes to subsequent inputs.

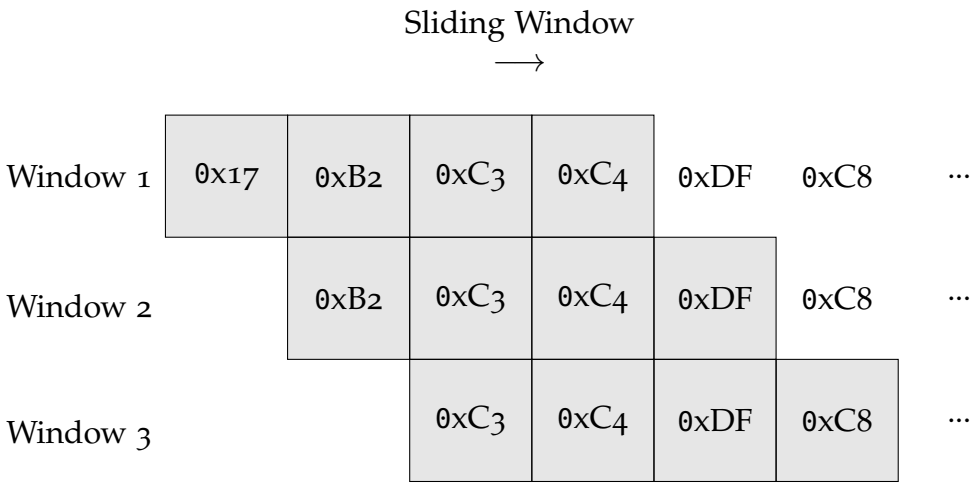


Figure 6.1: Example of a four-address, fixed-width window. Here, an executing program accesses addresses `0x17`, `0xB2`, `0xC3`, etc. Each window (n-gram) thus represents a local snapshot of accessed memory locations as the window slides across all memory accesses.

Instead of considering a program's unique sequence of memory accesses as a whole, we present a stream-based approach that can scale to arbitrary-size programs, observing a fixed-width window of the n most recently accessed memory addresses. This window acts as a shift register, allowing MARTINI to observe a *sliding window* of memory addresses as program execution proceeds. This provides a localized, contextual view of a program's recent memory behavior that can be monitored during the execution of the program. Figure 6.1 provides an example of the construction of windows of width four from a stream of memory addresses.

While individual n-grams are likely shared across the execution of different programs, we hypothesize that the *set* of all windows for a given program pro-

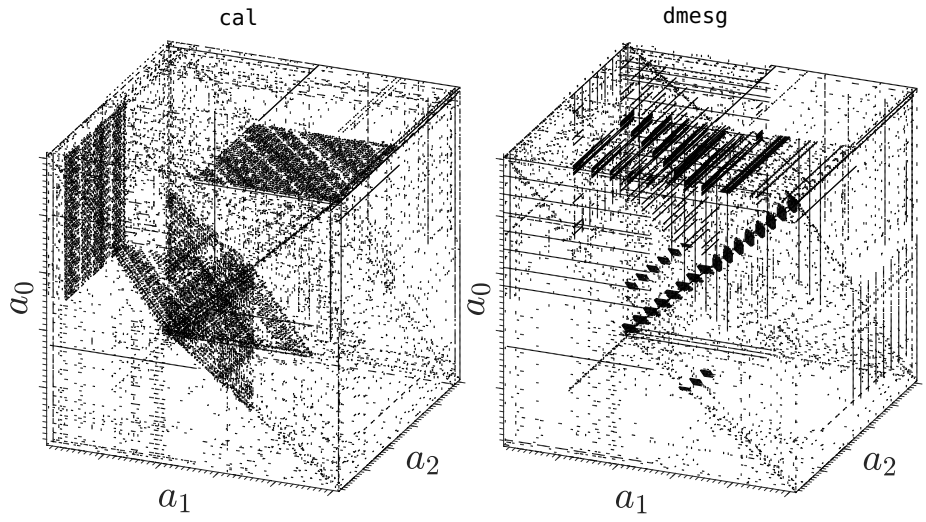


Figure 6.2: Visualization of n-gram representation for two programs. Sets of windows of size three are shown for memory accesses of `cal` and `dmesg`. Each point (a_0, a_1, a_2) in three-dimensional space represents a unique window recorded during the execution of the program, where a_n represents the n^{th} address in the window. The plots are structurally different between the two programs indicating significant differences in their behavior. We consider such windows in 8 dimensions.

vides a unique signature that is difficult to spoof. This is the intuition behind our approach—programs are characterized by the pattern of memory addresses accessed during execution. As an example, consider three-address windows: Figure 6.2 plots windows for the Linux utilities `cal` and `dmesg` in three-dimensional space, where each dimension represents one bit of the address. Each point represents a unique sequence of three memory addresses recorded during the execution of the program. The two plots are structurally dissimilar (e.g., the dense behavior on the "center-left," a_0–a_2 region, for `cal`), which suggests that simply comparing fixed-width memory access window sets will differentiate the execution of individual programs. We rigorously evaluate this hypothesis in Section 6.7.

6.1.2 *Dictionaries of Program Behavior*

Next, we extend the notion of memory access windows to (1) allow a system operator to define a collection of authorized programs and (2) compactly represent sets of valid windows.

Statically determining the exact execution path of a program is undecidable. Instead, we sample many indicative memory traces from each authorized program. We next construct a *dictionary* with all of the windows generated by this *training* set. The dictionary is an abstract model of authorized program behavior. New programs and traces can be added to a dictionary without retraining from scratch. Similar to other IDS approaches, the quality of the model is determined by the extent to which the training set generalizes to all normal behaviors. However, previous experience has shown that most programs have highly conserved execution patterns under benign inputs.

Two related challenges in offline learning of labeled training data include overfitting and model size [54, 100]. We require a solution that avoids overfitting (so that it will generalize to untrained benign program input data for high-assurance whitelisting) and that admits a compact representation (so that it can be efficiently deployed in hardware, such as in a compact, automata-derived functional unit). To address these challenges, we introduce three additional refinements:

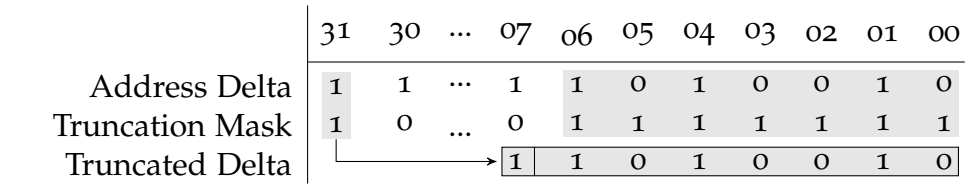


Figure 6.3: Example of address truncation. Memory address deltas are bitwise AND'd with a truncation mask and packed into 8-bit values. In practice, a sign bit and the seven least significant bits produce accurate results.

6.1.2.1 Δ-*Windows*

Defensive techniques such as address space layout randomization (ASLR) randomize important memory locations of processes to harden systems against classes of exploits [193]. For Martini, the execution of identical processes could produce significantly different absolute memory traces. We generalize otherwise identical memory traces by storing the *distance between* consecutively accessed memory addresses rather than absolute locations. While absolute addresses can vary across executions, we hypothesize that these distances, or *deltas*, likely remain constant and generalize. We refer to windows of address deltas as Δ-*windows*.

6.1.2.2 *Truncation*

Δ-windows mitigate some of the risk of overfitting due to address randomization, but differences between physical and virtual addresses remain. We mitigate this by *truncating* the address delta values to b bits, excluding bits in the delta that may be specific to the physical page selected at runtime. This also significantly reduces the address space represented by our model, helping generalize the model and reduce

overfitting. MARTINI supports general masking of the address deltas. Although a full parameter sweep falls outside the scope of this work, our experimentation showed that storing a sign bit and the seven least significant bits of the address deltas produces good results. An example of address delta truncation is given in Figure 6.3.

6.1.2.3 *Compression*

Simply truncating deltas (e.g., to 8 bits) improves the model, but is still insufficient for our needs. For example, for Δ-windows of length 8 containing 8-bit truncated deltas, there are $2^{8 \cdot 8} = 2^{64}$ unique values that could be stored in a dictionary. Even when storing fewer than half of these values, we found that a dictionary trained on a subset of Linux Coreutils contained approximately 40 million windows, which is several orders of magnitude larger than what MARTINI can efficiently support (Section 6.3).

To address this scalability challenge, we compress dictionaries using a method similar to earlier work on system calls [84]. In this scheme, the first element of a window is stored exactly, but each subsequent position is represented by the *unordered set* of all observed values at that offset from that starting element. For example, if the windows $\langle c, a, t \rangle$, $\langle c, o, w \rangle$, and $\langle d, o, g \rangle$ were observed, the compressed dictionary would store $\langle c, \{a, o\}, \{t, w\} \rangle$ and $\langle d, \{o\}, \{g\} \rangle$. Note that this compressed dictionary accepts the original three windows as well as "caw" and "cot"; the compression is not lossless. Additionally, "dog" is stored separately in the compressed dictionary because its first element is distinct. Thus, while the

compression generalizes a dictionary, it also reduces the size, admitting efficient hardware implementation.

For windows of length k consisting of b-bit deltas, the number of possible values stored in the compressed dictionary reduces from $2^{k \cdot b}$ to $k \cdot 2^b$. Our empirical evaluation in Section 6.7 demonstrates that compressed dictionaries retain sufficient fidelity to detect unauthorized program execution, including difficult-to-observe hardware side-channel attacks.

6.1.3 *Detecting Anomalous Program Execution*

During the training phase, MARTINI records all of the memory traces associated with runs of authorized programs on indicative workloads. We consider fixed-width sliding windows of addresses from those traces, convert adjacent addresses to Δ-windows, truncate each delta to a smaller number of bits, and finally generate a compressed dictionary to store (an over-approximation of) the set of truncated Δ-windows associated with those program and runs. Our experimental results show that such sets of abstracted memory addresses characterize program behavior in a way that is sensitive to the classes of anomalies in which we are interested.

After training, we determine whether a new sequence of memory accesses matches the model by converting incoming accesses to a truncated Δ-window and querying the dictionary for membership. If observations fall outside the dictionary, MARTINI flags the sequence as anomalous (and possibly malicious). We refer to

these anomalous sequences as *mismatches*. A *mismatch counter* c, initialized to zero, increments by one whenever MARTINI detects a mismatch. The mismatch counter is multiplied by a *decay* coefficient d ($0 \leqslant d < 1$) every N windows to retain local context and eventually forgive past mismatches. The *mismatch rate* r ($0 \leqslant r < 1$) is defined as $r = (1 - d)c/N$. An alarm triggers when the mismatch rate exceeds a predefined threshold t. Briefly, the mismatch rate reflects the concentration of mismatches at any point in time. This allows the system to tolerate some false positives, while still responding to legitimate deviations. t controls the sensitivity of the system and allows MARTINI to be configured to optimize the trade-off between false- and true-positives in different settings. Proper tuning of thresholds has been demonstrated to mitigate many false positives [202]. We evaluate mismatch rate thresholds in Section 6.7.

This work focuses on detecting anomalies. Responses to anomalies could be incorporated in various ways: (1) alarm signals could be used by the OS to terminate the process or (2) the memory system could delay completion of the memory transaction. Termination would require careful implementation to avoid denial of service and livelock. Delay-based approaches are effective in some OS settings because users can often tolerate an occasional, slight delay [202]. For some versions of Spectre and Meltdown, delays would explicitly defeat relevant timing-based calculations. We leave the development of robust response mechanisms for future work.

6.2 COMPILING GRAMMARS TO PUSHDOWN AUTOMATA

In this section, we describe context-free grammars, our algorithms to compile such grammars to pushdown automata, and our prototype implementation.

6.2.1 *Context-Free Grammars*

While DPDAs provide a functional definition of computation, it can often be helpful to use a higher-level representation that generates the underlying machine. Just as regular expressions can be used to generate finite automata, *context-free grammars* (CFGs) can be used to generate pushdown automata. We briefly review relevant properties of these grammars (the interested reader is referred to references such as [87, 93, 99, 199] for additional details).

CFGs allow for the definition of recursive, tree-like structures using a collection of *substitution rules* or *productions*. A production defines how a symbol in the input may be legally rewritten as another sequence of symbols (i.e., the right-hand side of a production may be *substituted* for the symbol given in the left-hand side). Symbols that appear on the left-hand side of productions are referred to as *non-terminals* while symbols that do not are referred to as *terminals*. The *language* of a CFG is the set of all strings produced by recursively applying the productions to a starting symbol until only terminal symbols remain. The sequential application

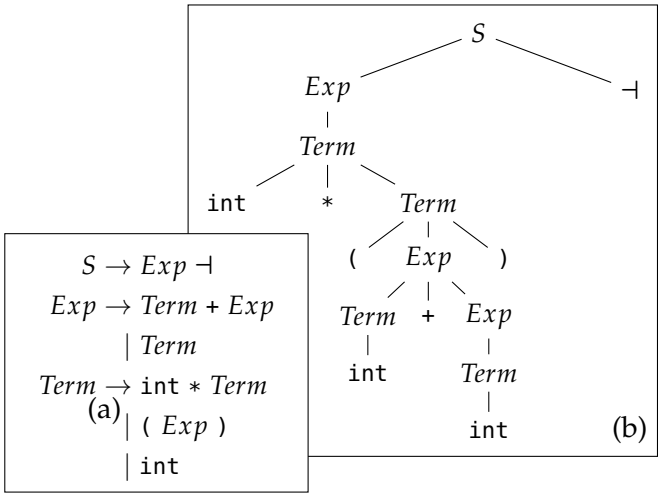


Figure 6.4: An example CFG (a) and parse tree (b). The grammar represents a subset of arithmetic expressions. We use ⊣ to signify the endmarker for a given token stream, which is needed for transformation to a DPDA. The parse tree given in (b) is for the expression $3 * (4 + 5)$. Note that integer numbers are transformed to `int` tokens prior to deriving the parse tree.

of these productions to an input produces a *derivation* or *parse tree*, where all internal nodes are non-terminals and all leaf nodes are terminals.

An example CFG for a subset of arithmetic operations is given in Figure 6.4 (a). This particular grammar demonstrates recursive nesting (balanced parentheses), operator precedence (multiplication is more tightly bound than addition), and associativity (multiplication and addition are left-associative in this grammar). Figure 6.4 (b) depicts the parse tree given by the grammar for the equation $3 * (4 + 5)$.

6.2.2 *Compiling Grammars to DPDAs*

Next, we consider the process of compiling an input CFG to a DPDA. As noted in Section 2.1.2, PDAs and DPDAs do not have equal representative power. Therefore, there are CFGs that cannot be recognized by a DPDA. We focus on support for a strict subset of CFGs known as $LR(1)$ grammars, which are of practical importance and supported by DPDAs. Most programming language grammars have a deterministic representation [199], and many common parser generator tools focus on supporting $LR(1)$ grammars [26, 108, 138]. By targeting this class of grammars, we can therefore support parsing common languages such as XML, JSON, and ANSI C.

Existing parser generators (e.g., `YACC` or `PLY`) are unsuitable for compiling to ASPEN because these tools do not produce hDPDAs (or even DPDAs!). Instead, they generate source code that makes use of the richer set of operations supported by CPUs. We do, however, demonstrate how existing tools may be leveraged for a portion of our compilation process.

This transformation from grammar to hDPDA is broken down into three stages: (1) parsing automaton generation, (2) hDPDA generation, and (3) optimization.

6.2.2.1 *Parsing Automaton Generation*

Parsing of input according to an $LR(1)$ grammar makes use of a DFA known as a *parsing automaton*,[3] a state machine that processes input symbols and determines

3 Also referred to as *DK* in the literature after its creator, Donald Knuth [199].

the next production to apply. This machine encodes *shift* and *reduce* operations. Shifts occur when another input token is needed to determine the next production and are encoded as transitions between states in the parsing automaton. Reduce operations (the reverse applications of productions) occur when the machine has seen enough input to determine which substitution rule in the grammar to apply and are encoded as accepting states in the DFA. Each accepting state represents a different production. Determining the correct shift or reduce operation may require inspecting the current input symbol and also a subsequent *lookahead* symbol.

We leverage off-the-shelf tools to generate parsing automata. Concretely, we support parsing automata generated by the `GNU Bison`[4] and `PLY`[5] parser generator tools. These two tools produce CPU-based parsers and generate parsing automata as an intermediate output.

Conceptually, parsing proceeds by processing input symbols using the parsing automaton and pushing symbols to the stack until an accepting state is reached. The input string is rewritten by popping symbols from the stack. The most recently pushed symbols are replaced by the left-hand-side of the discovered substitution rule. Processing is then restarted from the beginning of the rewritten input, repeating until only the starting non-terminal symbol remains. With this classical approach, parsing requires multiple iterations over (and transformations to) the input symbols.

4 `https://www.gnu.org/software/bison/`
5 `http://www.dabeaz.com/ply/`

6.2.2.2 *hDPDA Generation*

To improve the efficiency of parsing, we simulate the execution of the parsing automaton using a DPDA [199, Lemmas 2.58, 2.67] to process input tokens in a single pass with no transformations to the input. With this approach, input symbols are *not* pushed to the stack. Instead, the stack of the hDPDA is used to track the sequence of states visited in the parsing automaton. Shift operations push the destination parsing automaton state to the stack (shifts are transitions to other states in the parsing automaton). When a reduce operation rewriting n symbols to a single non-terminal symbol is performed by the parsing automaton, the hDPDA pops n symbols off the stack. The symbol at the top of the hDPDA stack is the state of the parsing automaton that immediately preceded the shift of the first token from the reduced rule. In other words, popping the stack for a reduction "runs the parsing automaton in reverse" to undo shifting the symbols from the matched rule. The hDPDA then continues simulation of the parsing automaton from this restored state.

Our prototype compiler generates an hDPDA by first reading in the textual description of the parsing automaton generated by `Bison` or `PLY`. Next, for each state in the parsing automaton, we generate hDPDA states for each terminal and non-terminal in the grammar. A separate state is needed for each terminal and non-terminal symbol because the homogeneity property only supports a single pushdown automata operation per state, as defined in Equation (2.1):

- For each terminal symbol, we generate two states: one state matches the lookahead symbol (i.e., lookahead symbols are stored in "positional" memory) and one state encodes the relevant shift or reduce operation. A shift operation pushes parsing automaton states on the stack, while a reduce operation pops a symbol from the stack and generates an output signal.

- For each non-terminal symbol, only one state is generated: the state performing the shift/reduce operation. In addition, this state must also match the top of the stack to validate undoing shift operations.

Then, we add additional states to perform stack pop operations for the reduce operations, one pop for each symbol reduced from the right-hand side of a production. Finally, we connect the states with transitions according to transition rules from the parsing automaton.

The final hDPDA is emitted in the MNRL file format. MNRL is an open-source JSON-based state machine serialization format that is used within the MNCaRT automata processing and research ecosystem [16]. We extend the MNRL schema to support hDPDA states, encoding the stack operations with each state. Using MNRL admits the reuse of many analyses from MNCaRT with minimal modification.

6.2.2.3 *Optimization*

While our algorithm to transform the parsing automaton to a DPDA is direct, the resulting DPDA contains a large number of ε-transitions and extraneous states.

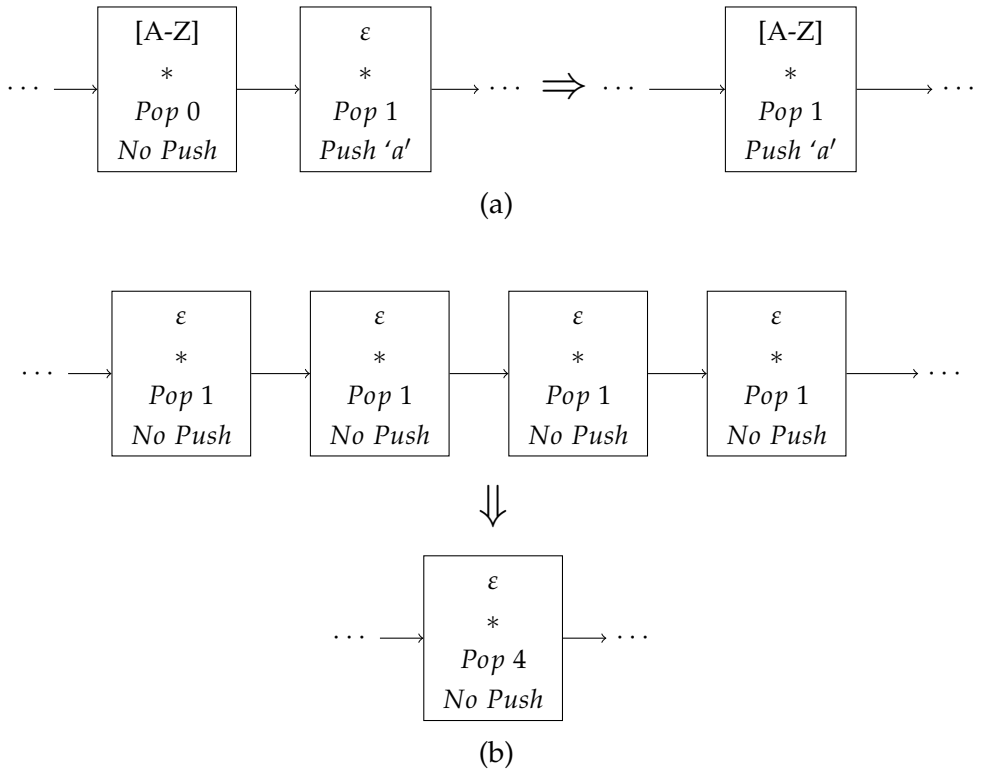


Figure 6.5: Two compiler optimizations for reducing the number stalls incurred by ε-transitions. Epsilon merging (a) attempts to combine states to perform non-overlapping operations. Multipop (b) allows for the stack pointer to be moved a configurable distance in one operation.

First, we remove all unreachable states (states with no incoming transitions). Then, we perform optimizations to reduce the total number of ε-transitions within the hDPDA. Recall that ε-transitions occur when stack operations take place without reading additional input (e.g., when popping the stack during a reduce operation and transitioning to another state). We make two observations about the hDPDA produced by our compilation algorithm.

First, the algorithm produces separate states to "read in" input symbols and to perform stack operations. In many cases, these states may be combined, or

merged, to match the input and perform stack operations simultaneously. After producing the initial hDPDA, we perform a post-order depth-first traversal of the machine and merge such connected states when possible. We call this optimization *epsilon merging* and apply it conservatively: only states that occur on a linear chain are merged. Figure 6.5 (a) shows an example in which a state performing input matching on capital letters and a state (with no input comparison) performing a pop and a push are merged.

Second, our basic algorithm assumes a computational model that only supports popping one symbol at a time. On reduction operations for productions containing several symbols on the right-hand side, this results in long-duration stalls. Note, however, that no comparisons are made with these intermediate stack symbols. If our architecture can support moving the stack pointer by a variable amount, then a reduction may be performed in one step. We refer to this as *multipop*. Figure 6.5 (b) demonstrates a reduction of four states to one state with multipop.

6.2.3 *Compilation Summary*

We presented an overview of CFGs, a high-level language representation that may be used to generate pushdown automata. Then, we described an algorithm for compiling an important subset of CFGs ($LR(1)$ grammars) to hDPDAs. We leverage existing tools to produce an intermediate parser representation (the parsing automaton), which we then encode in an hDPDA for execution with ASPEN. We also introduce two optimizations, epsilon merging and multipop,

to reduce stalls while processing input. Our approach supports and accelerates existing parser specifications without modification. This means that parsers do not have to be redesigned to take advantage of ASPEN's increased parsing performance.

6.3 MARTINI ARCHITECTURAL DESIGN

Having detailed both MARTINI's and ASPEN's application-level design, we now describe the microarchitectural design and efficient implementation of MARTINI. We describe the design of ASPEN in the following section. First, we present the homogeneous finite automaton compressed dictionary representation. Then, we describe our architecture for monitoring memory accesses, which is embedded in the Last Level Cache (LLC) of the CPU.

6.3.1 *From Dictionaries to Automata*

We represent compressed dictionaries as homogeneous NFAs (as defined in Section 2.1.1) to facilitate hardware implementation and execution. A separate automaton, or *connected component* [209], is created for each Δ-window in the trained dictionary. Because there is a single entry in the compressed dictionary for

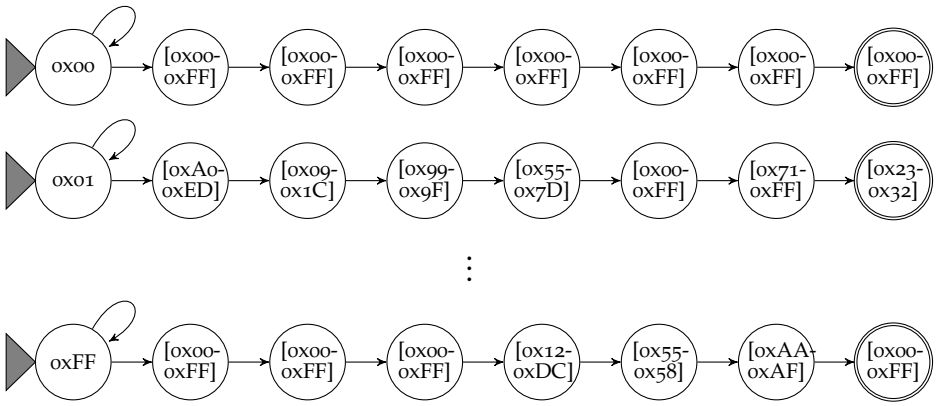


Figure 6.6: Homogeneous NFA representation of a dictionary. The NFA consists of 256 connected components, each containing a chain of eight STEs. The initial STE for each component matches a unique value; all subsequent STEs match a set of possible values. Training determines the values within.

each unique address delta that begins a window, b-bit deltas result in 2^b connected components.

Within a given automaton, we allocate one STE for each window offset, which forms a linear chain. The symbol match conditions are taken directly from the compressed dictionary. Additionally, each initial STE contains a self-loop to account for the sliding window comparison. The last state in each chain (equivalent to the final position in the window) generates a report signal if activated. In Martini Figure 6.6 illustrates the automata layout. When a new dictionary is trained, the overall automata topology is unchanged; only the symbols within individual STEs change. This insight allows us to simplify hardware-level routing to save space in silicon.

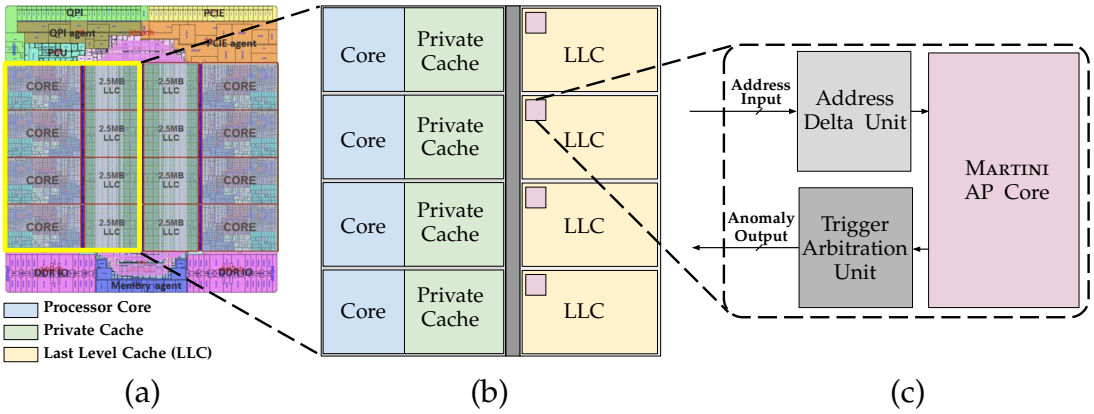


Figure 6.7: High-level architectural design of Martini. An 8-core Xeon processor, each with private L1 and L2 caches and shared 2.5MB Last Level Cache (LLC) slice with embedded Martini processor, and a block diagram of the Martini processor (shown in pink). Note that regions are not to scale.

The automata input is the sequence of truncated memory address deltas generated by the execution of a program. The automata generate a report for every input Δ-window that matches the encoded dictionary.

6.3.2 Martini *Address Monitor*

To support real-time monitoring of memory accesses, we embed Martini in the Last Level Cache (LLC) region of the CPU. Figure 6.7 shows an enterprise 8-core Intel Xeon-E5 processor. The Xeon family of processors typically includes 8–16 slices of LLC (one slice per core) [42, 53, 105]. In our prototype, each processor core is allocated a dedicated Martini unit (the pink rectangles within each private cache in the Figure). Our Martini unit consists of three components: the Address Delta Unit, Automata Processing (AP) Core, and Trigger Arbitration Unit.

The Address Delta Unit snoops the memory address lines of the core and calculates the truncated delta between two consecutive addresses (as described in Section 6.1.2). In its simplest form, this unit performs two's complement arithmetic on 8-bit values; however, a more sophisticated unit could support dynamically masking and truncating address deltas.

The generated address deltas are then fed into the AP core, described in the next subsection. This core executes the automata computation, producing triggers when a window of address deltas is not found in the loaded dictionary.

The Trigger Arbitration Unit tracks triggers and generates an alarm signal when a pre-defined threshold is exceeded. Our prototype implementation consists of two counters. The first counter tracks the number of windows processed, while the second counter tracks the number of triggers produced by the AP core. Whenever the window counter reaches its threshold, the trigger counter is shifted to decay the value and favor local context. If the trigger counter reaches its threshold (i.e., the mismatch rate from Section 6.1.3), the Trigger Arbitration Unit produces a hardware signal indicating an anomaly, which is handled by the OS or memory system.

6.3.3 *Automata Processing Core*

The AP Core is responsible for taking an input Δ-window and determining whether it is present in the dictionary. We next describe our prototype design for

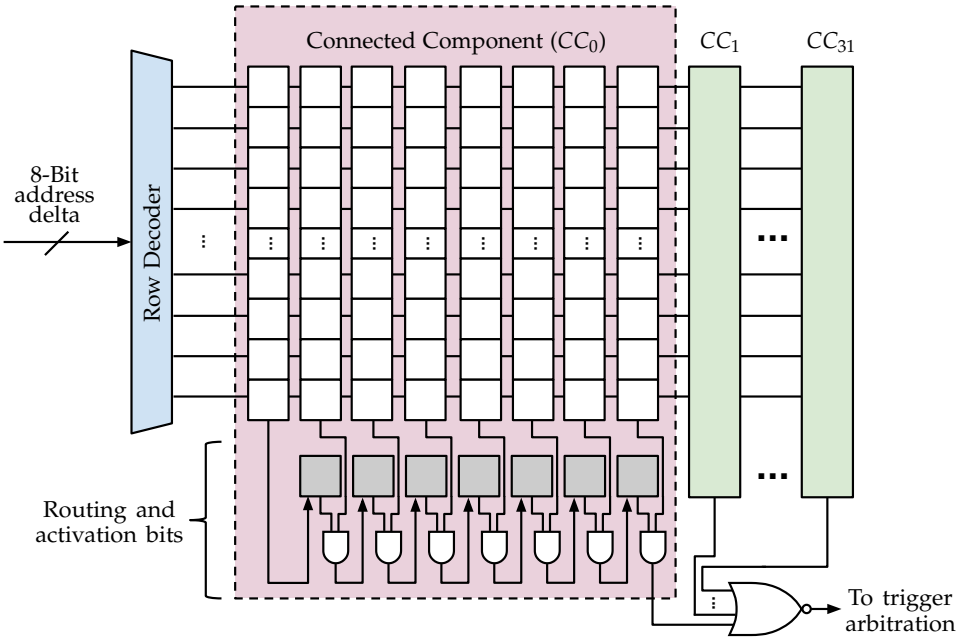


Figure 6.8: Specialized Martini automata processing architecture. Routing of activation signals is simplified because connected components in the automata consist of chains of eight STEs (shown in the dashed region). A single 256×256 SRAM Array contains 32 connected components. We require eight arrays (256 connected components) in total. Masked address deltas are fed as input to the row decoders, and outputs from each connected component are fed to trigger arbitration.

implementing Martini in a current-generation Intel Xeon CPU, which represents a novel, system-level integration of automata processing.

Our AP Core follows much of the implementation of the Cache Automaton [209]; however, we make several application-specific modifications to reduce space overhead and improve performance. Each 2.5 MB slice of LLC in the Xeon processor is organized into 20 ways, each of which is subdivided into five 32 kB banks. Four of these banks constitute data arrays, while the fifth is used for storing cache state [42, 53, 105]. Internally, the banks used for data arrays are made

up of four 8 kB (256 × 256) SRAM arrays. We repurpose these SRAM arrays to perform automata computation. A single SRAM array can accommodate 256 STEs, meaning that to accommodate all 2048 STEs of the compressed dictionary, eight arrays—two banks—are repurposed for the AP Core.

Figure 6.8 depicts the repurposed SRAM array. As described in Section 2.2, each column encodes the input matching rule for an STE following the state-match design of previous memory-centric AP models [75, 209]. The row decoder converts the current address delta to a 256-bit one-hot encoding. The homogeneity property of the automata ensures that STEs can be represented by a single column of SRAM. Each STE also has a corresponding activation bit. An STE must both match the input symbol and also be active to generate a transition signal. One exception is the initial STE in each connected component: this STE is always active (every cycle is also the start of a new sliding window).

In general-purpose automata processing, a second SRAM array is used to support a reconfigurable routing matrix for transition signals. For Martini, this is not needed; the topology of the automata is fixed, consisting of chains of eight STEs. This allows for static routing in which the transition signal from the previous STE feeds into the activation bit register of the next STE, resulting in a more compact design. The transition signal out of the last STE in each connected component chain feeds into a NOR gate, which aggregates signals from all of the connected components and produces a trigger for the Trigger Arbitration Unit.

6.3.4 *System Integration*

Compressed automata dictionaries are (1) placed and routed for hardware resources and (2) stored as a bitmap containing STE input match symbols and the thresholds for the Trigger Arbitration Unit. At runtime, the OS loads the bitmap into the monitoring unit using standard load instructions and Intel Cache Allocation Technology [110]. Anomaly alarms trigger a hardware interrupt, allowing the OS to implement custom mitigation strategies. The configuration overheads are small (roughly equivalent to loading 2 kB of data into the LLC) and typically only occur once. The unit only needs to be reconfigured when loading a new dictionary.

6.4 ASPEN ARCHITECTURAL DESIGN

Having described the architectural design of MARTINI, we now focus on describing the ASPEN architecture that augments LLC slices with support for DPDA processing. We also discuss the design of a DPDA processing pipeline based on ASPEN and the trade-offs involved.

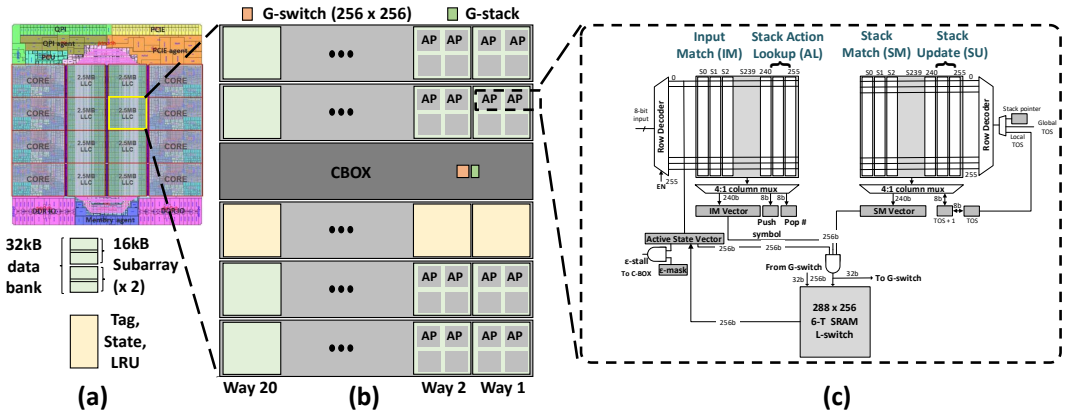


Figure 6.9: Xeon processor with SRAM arrays repurposed for DPDA processing. The figure shows (a) 8-core Xeon processor, (b) one 2.5MB Last Level Cache (LLC) slice and (c) Internal organization of one 32kB bank with two 8kB SRAM arrays repurposed for DPDA processing.

6.4.1 *Cache Slice Design*

The ASPEN architecture augments the Last Level Cache slices of a general purpose processor to support in-situ DPDA processing. Figure 6.9 (a) shows an 8-core enterprise Xeon-E5 processor with LLC slices connected using a ring interconnect (not shown in figure). Typically, the Intel Xeon family includes 8-16 such slices [42, 53, 105]. Each Last Level Cache slice macro is 2.5 MB and consists of a centralized cache control box (C-BOX). A slice is organized into 20 ways, with each way further organized as five 32 kB *banks*, four of which constitute data arrays, while the fifth one is used to store the tag, valid and LRU state (Figure 6.9 (b)). All the ways of the cache are interconnected using a hierarchical bus supporting a bandwidth of 32 bytes per cycle. Internally, each bank consists of four 8 kB SRAM arrays (256×256).

A bank can accommodate up to 256 states and a DPDA can span several banks. We repurpose two of the four arrays in each bank to perform the different stages of DPDA processing. The remaining two arrays (addressed by the `PA[16]` bit) can be used to store regular cache data. State-transitions are encoded in a hierarchical memory-based interconnect, consisting of local and global crossbar switches (L-switch, G-switch). A 256-bit register is used to track the active states in each cycle (Active State Vector in Figure 6.9 (c)). We provision input buffers in the C-BOX to broadcast input symbols or tokens to different banks. Output buffers are also provided to track the report events generated every processing cycle.

6.4.2 *Operation*

This subsection provides the details of DPDA processing. Recall that, in a DPDA, only a single state is active in every processing cycle, and initially, only the start state is active. Each input symbol from the DPDA input buffer is processed in five phases. In the *input match* and *stack match* phases, we identify the active DPDA state which has the same label as that of the input symbol and the top of stack (TOS) symbol respectively. In the *stack action lookup* phase, the stack action defined for that state is determined (i.e., push symbol or number of symbols to pop from the stack). The stack is updated in the following phase (*stack update*). Finally, in the *state-transition phase*, a hierarchical transition interconnect matrix determines the next active state.

Cycles in which states with an ε-transition are active require special handling. These states do not consume an input symbol but perform a stack action in that cycle (i.e., push or pop). A 256-bit ε-mask register tracks the ε-states in each bank. A logical AND of the ε-mask register and Active State Vector is used to determine if an ε-state is active in the next processing cycle. If an ε-state is active, a 1-bit ε-stall signal is sent to the C-BOX to stall the input for the next processing cycle.

While a single stack action per cycle is sufficient to support DPDA functionality, reducing stalls to the input stream can significantly improve performance. The *multipop* optimization, discussed in Section 6.2.2.3, reduces stalls due to ε-transitions and is supported in hardware by manipulating the stack pointer and encoding the number of popped symbols in the stack action lookup phase. We now proceed to discuss the different stages involved in DPDA processing.

(1) INPUT-MATCH (IM): We adapt the state-match design of memory-centric automata processing models [75, 209] for the input-match phase. Each state is mapped to a column of an SRAM array as shown in Figure 6.9 (c). A state is given a 256-bit input symbol label which is the one-hot encoding of the ASCII symbol that it matches against. The homogeneous representation of DPDA states ensures that each state matches a single input symbol and each state can be represented using a single SRAM column. The input symbol is broadcast as the row address to the SRAM arrays using 8-bits of global wires. By reading out the contents of the row into the *Input Match Vector*, the set of states with the same label as the input symbol can be determined in parallel.

(2) STACK-MATCH (SM): In contrast to NFAs, where all active states that match the input symbol are candidates for state-transition, DPDA states have valid transitions defined only for those states that match *both* the input symbol and the symbol on the top of the stack (8-bit TOS in Figure 6.9). We re-purpose an SRAM array in each bank to determine the set of DPDA states that match the top of stack (TOS) symbol. Similar to Input-Match, we provision 8 bits of global wires to broadcast the TOS symbol as the row address to SRAM arrays. By reading out the contents of the row into the *TOS Match Vector* and performing a logical AND with the *Input Match Vector* and the *Active State Vector*, the candidate states for state-transition are determined. We refer to these candidate states simply as *active states*.

We leverage sense-amplifier cycling techniques [209] to accelerate the IM and SM stages.

(3) STACK ACTION LOOKUP (AL): Each DPDA state is also associated with a corresponding stack action. The supported stack actions are `push`, `pop` and `multipop`. The stack action is encoded with 16 bits. Each `push` action uses 8 bits to indicate the symbol to be pushed onto the stack. The remaining 8 bits are used by the `pop` action to indicate the number of symbols to be popped from the stack ($>$ 1 for `multipop`).

The stack action corresponding to each state is packed along with the IM SRAM array in each bank. However, in the AL stage, we lookup this SRAM array using the 256-bit result vector obtained after logical AND in the previous step (see Figure 6.9). This removes the decoding overhead from the array access time. We

reserve 16 bits of global wires to communicate the stack action results from each bank to the stack control logic in the C-BOX.

(4) STATE TRANSITION (ST): The state-transition phase determines the set of states to be activated in the next cycle. We observe that the state transition function can be compactly encoded using a hierarchy of local and global memory-based crossbar switches. The state transition interconnect is designed to be flexible and scales to several thousand states. The L-switches provide dense connectivity between states mapped to the same bank while the G-switch provides sparse connectivity between states mapped to multiple banks. A graph partitioning based algorithm [121] is used to satisfy the local and global connectivity constraints while maximizing space utilization.

The crossbar switches consisting of N input and output ports and $N \times N$ cross-points are implemented using regular 6-T SRAM arrays (e.g., L-switch in Figure 6.9 (c)). The 6-T bitcell holds the state of each cross-point. A flip-flop or register can also be used for this purpose, but these are typically implemented using 24 transistors making them area inefficient. A '1' is stored in bitcell (i, j) if there is a valid transition defined from state i to state j. All the cross-points are programmed once during initialization and used for processing several MBs to GBs of input symbols. The set of *active* states from the previous phase serve as inputs to the crossbar switch. For DPDAs, only a single state can be active every cycle and we can use 6-T SRAM arrays for state transition, since only a single row is activated.

(5) STACK UPDATE (SU): To allow for parallel processing of small DPDAs, (e.g., in non-parsing applications, such as subtree mining), we provide a local

stack in each bank. We repurpose 8 columns of the SM array to accommodate the local stack. Larger DPDAs (e.g., in XML parsing) make use of a global stack to keep track of parsing state. The global stack is implemented in the C-BOX using a 256×8 register file and is shared by all the DPDAs mapped to two adjacent ways. Providing a stack depth of 256 is sufficient for our parsing applications (see Section 6.8). Note that only one sort of stack (local or global) is enabled at configuration time based on the DPDA size. The stack pointer is stored in an 8-bit register and is used to address the stack. We also store the symbols at stack positions TOS and TOS+1 in separate 8-bit registers. This optimization saves a write and read access to the larger stack register file and ensures early availability of the top-of-stack symbol for the next processing cycle. The `push` operation writes the stack symbol to TOS+1. A lazy mechanism is used to update the stack with the contents of TOS. Similarly, the `pop` operation copies TOS to TOS+1, while lazily reading the stack register file to update TOS.

6.4.3 *Critical Path*

ASPEN's performance depends on two critical factors: (1) the time taken to process each symbol in the input stream (i.e., clock period) and (2) the time spent stalling due to ε-transitions. The multipop optimization reduces stalls due to ε-transitions. We now consider the clock period.

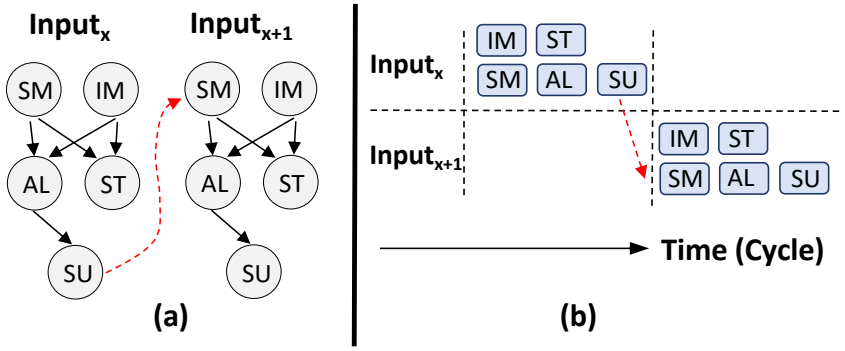


Figure 6.10: DPDA processing on ASPEN. (a) Dependency graph between stages. (b) Serial processing of input symbols.

In a naïve approach, each input symbol would be processed sequentially in five phases, leading to a significant increase in the clock period. However, not all phases are dependent on each other and need to be performed sequentially. Figure 6.10 (a) shows the dependency graph for the DPDA processing stages. The intra-symbol dependencies are shown in black, while the inter-symbol dependencies are marked in red. Using the dependency graph, each of the five stages can be scheduled as shown in Figure 6.10 (b), where the propagation through the interconnect (wire and switches) for state-transition is overlapped with stack action lookup and stack update. Since the top of stack cannot be determined until the stack has been updated based on the previous input symbol, DPDA processing is serial. We contrast this with NFA processing, which has two independent stages (input-match and state-transition) that can be overlapped to design a two-stage pipeline [209]. We find that the critical path delay (clock period) of ASPEN is the time spent for input/stack-match and the time taken for stack action lookup and update. The time spent in state-transition is fully overlapped with stack

related operations. Section 6.6.3 discusses the pipeline stage delays and operating frequency.

6.4.4 *Support for Lexical Analysis*

Two critical steps in parsing are *lexical analysis*, which partitions the input character stream to generate a token stream, and *parsing*, where different grammar rules are applied to verify the well-formedness of the input tokens (see Section 2.5.1). ASPEN can accelerate both these phases. We leverage the NFA-computing capabilities of the Cache Automaton architecture [209] for lexical analysis. To identify the longest matching token, we run each NFA until there are no active states. When the Active State Vector is zero, a *state exhaustion* signal is sent to the lexer control logic in the C-BOX. The symbol cycle and reporting state ID of the most recent report are tracked in a 64-bit report register in the C-BOX. A 256-bit reporting mask register is used to mask out certain reports based on lexer state. On receiving the *state exhaustion* signal from all banks, the lexer control logic resets the reporting mask, reloads the NFA input buffer for the next token and generates a token stream to be written into the DPDA input buffer (using a lookup table to convert report codes to tokens).

6.4.5 *System Integration*

ASPEN shares the Last Level Cache with other CPU processes. By restricting DPDA computation to only 8 ways of an LLC slice, we allow for regular operation in other ways. Furthermore, the cache ways dedicated to ASPEN may be used as regular cache ways for non-parsing workloads. Cache access latency is unaffected since DPDA-related routing logic uses additional wires in the global metal layers.

DPDAs are (1) placed and routed for ASPEN's hardware resources, and (2) stored as a bitmap containing states and stack actions. At runtime, the driver loads these binaries into cache arrays and memory mapped switches using standard load instructions and Intel Cache Allocation Technology [110]. The input/output buffers for ASPEN are also memory-mapped to facilitate input streaming and output reporting, and ISA extensions are used to start/stop DPDA functions. We disable LLC slice hashing at configuration time. The configuration overheads are small, especially when processing MBs or GBs of input, but are included in our reported results. To support automata-based applications that require counting, we provision four 16-bit counters per way of the LLC.

Post-processing of output reports takes place on the CPU. For XML parsing pipelines, a DOM tree representation (see Section 2.5.1) can be constructed by performing a linear pass over the DPDA reports. Richer analyses (such as verifying opening and closing tags match for XML parsing supporting arbitrary tags) may be implemented as part of tree construction. Although the CPU-ASPEN pipeline can support this, we leave evaluation of DOM tree construction for future work.

6.5 EXPERIMENTAL METHODOLOGY

In this section, we describe our methodology for evaluating both MARTINI and ASPEN. Although we have not fabricated the custom data paths described in Section 6.3 or in Section 6.3, we evaluate both using cycle-accurate simulation. We first describe our methodology for collecting data for our evaluation of MARTINI. Then, we describe the benchmark suites used in each of our evaluations.

6.5.1 *Recording Memory Traces to Evaluate* MARTINI

We built two helper tools to collect memory access traces of target programs. First, we leverage an extension to QEMU [30] called PANDA (the Platform for Architecture-Neutral Dynamic Analysis) [76]. Since QEMU is a full-system emulator, this approach has the advantage that we can instrument every instruction executed by the guest system without perturbing its behavior. Second, we used Intel's Pin tool [144] to collect memory traces of userspace programs. In contrast to the QEMU-based approach, Pin can collect memory traces much more quickly, where faithful modeling of the cache hierarchy is necessary. However, the primary disadvantage to Pin is the need to statically modify a target binary, potentially changing memory addresses that are accessed at runtime.

These PANDA and Pin instrumentations are used only in our simulation evaluation to establish a ground truth; they are not part of our proposed deployment. We use both approaches to collect memory traces of a suite of benchmark pro-

grams. While Pin instrumentation modifies the software under test, we observed a difference of less than 1%.

6.5.2 *Building and Testing Dictionaries*

We next construct a dictionary from the recorded memory traces by applying the refinements described in Section 6.1.2. First, we calculate the differences between consecutive memory accesses in the traces. Next, we slide a fixed-width window across this data to form 8-delta-long Δ-windows while simultaneously truncating each value to 8 bits. Finally, we construct a dictionary by creating sets of address deltas for each window offset.

We use MNRL to generate automata from a compressed dictionary. MNRL is an open-source state machine representation language and language API intended for large-scale automata processing applications [12, 16]. To simulate the execution of our accelerator architecture, we use VASim, a cycle-accurate simulator for automata processing architectures [235]. We extend the simulation to support the operations of the Address Delta and Trigger Arbitration units. In our evaluation, we load memory traces from the testing set into VASim and process the data using the compressed dictionary NFA, producing a list of generated alarms.

6.5.3 *Benchmarks*

We next describe the benchmarks we use to evaluate the performance of both MARTINI and ASPEN. Because these two architectures are designed to perform disjoint tasks, we consider different workloads for each.

MARTINI BENCHMARKS. We use multiple software benchmarks as indicative examples of both benign and malicious behavior. Table 6.1 shows these programs aggregated into one of three benchmark suites based on general behavior. In total, we collect and test on over 2,400 program traces and over 13 billion memory accesses.

The *Coreutils Subset* features a subset of 16 of the Linux *coreutils* programs, commonly used as benchmarks (e.g., [158, 250]). Programs in this suite represent a wide range of benign applications; we executed each with a variety of command-line arguments to gather memory access traces. The *PARSEC* benchmark [36] is composed of larger multithreaded programs that are designed to simulate a diverse set of highly parallelizable programs (e.g., ray tracing, fluid simulation, video compression, etc.). We use these programs as a test set for evaluating whether our technique can successfully detect execution that is not part of a trained dictionary. The *Security* benchmarks include representative malicious behavior: Spectre, Meltdown and two recent CVEs associated with Linux programs: `dnstracer` and `objdump`. The additional "Detect" column indicates whether

Table 6.1: Summary of benchmarks used to evaluate MARTINI

Coreutils Subset Suite

Program	Version	Traces	Avg. Trace Length
`cal`	N/A	269	307,994
`cat`	coreutils 8.25	227	133,667
`cp`	coreutils 8.25	175	274,652
`date`	coreutils 8.25	29	102,393
`diff`	coreutils 3.3	50	266,856
`dmesg`	util-linux 2.27.1	50	7,077,967
`dnstracer`	1.8.1	100	107,111
`du`	coreutils 8.25	257	6,498,869
`grep`	2.25	266	429,125
`ls`	coreutils 8.25	260	884,068
`objdump`	Binutils 2.26.1	150	1,066,007
`ps`	procps-ng 3.3.10	30	2,112,550
`readelf`	Binutils 2.26.1	50	8,911,505
`sed`	4.2.2	50	370,897
`tar`	1.28	232	458,187
`uname`	coreutils 8.25	57	93,281

PARSEC Suite

Program	Version	Traces	Avg. Trace Length
`blackscholes`	3.0	1	233,020,782
`bodytrack`	3.0	1	614,815,076
`canneal`	3.0	1	946,425,872
`dedup`	3.0	1	1,005,640,971
`facesim`	3.0	1	232,921,684
`ferret`	3.0	1	766,278,169
`fluidanimate`	3.0	1	640,500,257
`freqmine`	3.0	1	1,287,177,742
`raytrace`	3.0	1	1,005,358,609
`streamcluster`	3.0	1	473,597,488
`swaptions`	3.0	1	786,879,131
`vips`	3.0	1	1,596,912,331
`x264`	3.0	1	233,132,151

Security Suite

`Program`	Version	CVE	Traces	Avg. Trace Length	Detect
`Meltdown`	N/A	N/A	64	13,361,880	✓
`Spectre`	N/A	N/A	52	3,614,351	✓
`dnstracer`	1.8.1	2017-9430	52	227,616	✓
`objdump`	Binutils 2.26	2018-6323	50	814,034	✓

MARTINI can successfully detect the execution of these CVEs using a dictionary of indicative benign programs (see Section 6.7).

We trained dictionaries in our evaluation using a random 60% of the corresponding program's traces. For example, a dictionary containing `diff` would be trained using 30 of its traces, randomly selected.

ASPEN BENCHMARKS. We evaluate ASPEN against the widely used open-source XML tools Expat (v.2.0.1) [58], a non-validating parser, and Xerces-C (v.3.1.1) [18], a validating parser and part of the Apache project. The validation application used is *SAXCount*, which verifies the syntactic correctness of the input XML document and returns a count of the number of elements, attributes and content bytes. We restrict our analysis to the SAX interface and WFXML scanner of Xerces-C and filter out all non-ASCII characters in the input document. We do not include DOM tree generation in our evaluation. This is consistent with prior work and evaluations (e.g., Parabix, Xerces SAX, and Expat). We assume that input data is already loaded into main memory. Our XML benchmark dataset is derived from Parabix [141], Ximpleware [253] and the UW XML repository [251]. We only evaluate XML files larger than 512 kB in size, as we were unable to obtain reliable energy estimations when baseline benchmark execution time was under 1 ms. To evaluate the lexing-parsing pipeline, we extend the open-source, cycle-accurate virtual automata simulator, VASim [235], to support DPDA computation and derive per-cycle statistics. The tight integration of the lexer and parser in the LLC enables ASPEN to largely overlap the parsing time. Each lexing report can be processed and used to generate the token stream for the DPDA in 2 cycles.

All CPU-based evaluations use a 2.6 GHz dual-socket Intel Xeon E5-2697-v3 with 28 cores in total. We used PAPI [214] and Intel's RAPL tool [70] to obtain performance and power measurements. We utilize the METIS graph partitioning framework [121] to map DPDA states to cache arrays.

Table 6.3: Runtime overhead of reducing LLC capacity

	FULL CACHE		REDUCED CACHE		
PROGRAM	Runtime (ms)	Std. Dev.	Runtime (ms)	Std. Dev.	CHANGE
blackscholes	152.7	4.7	154.8	3.7	1.42%
bodytrack	457.2	25.5	455.2	30.1	-0.43%
canneal	2774.9	24.3	2809.4	26.1	1.24%
dedup	3236.1	25.3	3242.9	33.8	0.21%
facesim	11.4	0.6	11.5	0.5	0.92%
ferret	467.9	18.4	463.2	12.2	-1.02%
fluidanimate	3.1	0.3	3.1	0.3	-2.00%
freqmine	1053.8	78.5	1042.6	71.5	-1.07%
raytrace	2798.6	10.8	2794.5	11.2	-0.15%
streamcluster	35375.0	8428.7	32491.0	8015.4	-8.15%
swaptions	3.7	0.5	3.8	0.4	0.45%
vips	259.0	10.1	259.7	10.1	0.27%
x264	718.9	8.9	717.2	9.8	-0.23%

6.6 ARCHITECTURAL EVALUATION

In this section, we evaluate the runtime performance, chip area, and energy consumption of our hardware units.

6.6.1 *System Performance Impact*

Because computation in MARTINI is decoupled from cache operations, performance impacts are predominantly the result of reduced LLC capacity. ASPEN does couple tightly with CPU operations but does also reduce LLC capacity in a similar fashion. We therefore evaluate the performance impact incurred by repurposing

part of the LLC for the hardware address monitor using the *PARSEC* benchmark suite. We collected runtime performance metrics using a server running Ubuntu 16.04 with 192 GB of RAM and two Intel Xeon Platinum 8275CL CPUs, each with 36 cores running at 3 GHz. Each processor has 36 MB of LLC, subdivided into eleven cache ways. We execute each benchmark twenty times, recording wall clock execution time. Then, using Intel Cache Allocation Technology [110], we reduce the number of cache ways from eleven to ten and execute each benchmark an additional twenty times. We note that one cache way exceeds the resources required by our design of MARTINI for each processor core; the results here present an upper bound for the runtime overhead. ASPEN will generally consume more cache ways and may thus incur a larger overhead. Aggregate results are presented in Table 6.3.

In general, we find that the runtime overhead of our hardware is negligible. In the worst case (blackscholes), we observed a 1.42% increase. The largest change in performance (streamcluster) actually ran 8.15% *faster* with the reduced cache size. We hypothesize that this performance gain is caused by improved cache data alignment. However, any observed performance gain or loss is negligible: using a Wilcoxon signed-rank test, we are unable to find a significant difference in the average execution times for the full and reduced cache configurations ($p = 0.1536$).

6.6.2 MARTINI *Parameters*

Next, we study the feasibility of deploying MARTINI in real silicon by considering a current-generation Intel Xeon CPU. The 256 × 256 SRAM arrays in the Xeon LLC can operate at 4GHz [53, 105]. We estimate the area overhead for in-memory automata processing accelerators [17, 209]. We model the area overhead of the AP core with IBM's 45 nm `soi12s0` cell library and Synopsys Design Compiler. The total area is 0.016mm^2. For comparison, an 8-core Intel Xeon E5 processor has a die size of 354mm^2 [42] in a 22nm manufacturing process. Thus, our architectural changes would increase the overall die size by less than 0.04%. The synthesis results also shows that all the additional circuits achieve a 4GHz frequency after technology scaling to 22nm, thus maintaining existing frequencies.

Our AP core is built by adapting an SRAM array, which we use to estimate energy consumption. In the absence of publicly available data on array area and energy, we use the standard foundry memory compiler at 0.9 V in the 28nm technology node to estimate the power and area of a 256 × 256 6-T SRAM array. The energy to read out all 256 bits was calculated as 22 pJ. Since MARTINI is based on a Xeon-E5 processor modeled at 22nm, we scale down the energy per access to 13.6 pJ. As there are eight arrays used in an AP Core, the peak power of our AP Core is estimated at 0.435W. The peripherals of AP Core, the Address Delta Unit, and the Trigger Arbitration Unit together consumes 0.035W of power based on the synthesis results with IBM's 45nm cell library. In total, these sum to 0.470W,

Table 6.4: Stage delays and operating frequencies in ASPEN

DESIGN	IM/SM	ST	AL	SU	MAX FREQ.	FREQ OPER.
ASPEN	438 ps	573 ps	349 ps	349 ps	880 MHz	850 MHz
CA	250 ps	250 ps	-	-	4 GHz	3.4 GHz

which is far below the TDP of the Xeon E5 processor core (160 W). Therefore, the architecture does not incur any significant power overhead to the system.

6.6.3 *ASPEN Parameters*

We use the same 13.6 pJ calculation to estimate the energy to read out all 256 bits from the SRAM arrays in ASPEN. The area of each array and 6-T crossbar switch were estimated to be 0.015 mm^2 and 0.017 mm^2 respectively. Each LLC slice contains 32 L-switches and 4 G-switches to support DPDA computation in up to 8 ways. These switches can leverage standard 6-T SRAM push-rules to achieve a compact layout and have low area overhead ($\sim$6.4% of LLC slice area). Being 6-T SRAM based, these switches can also be used to store regular data when not performing DPDA computation. Similar to the Cache Automaton [209], we use global wires to broadcast input/stack symbols and propagate state transition signals. These global wires with repeaters have a 66ps/mm delay and an energy consumption of 0.07pJ/mm/bit.

Table 6.4 shows the stage delays for DPDA processing on ASPEN. The IM/TM phases leverage sense-amplifier cycling [209] and take 438 ps. The ST stage requires 573 ps, composed of 198 ps wire delay and 375 ps due to local and global switch

traversal. AL and SU each take 349 ps, composed of 99 ps wire delay and 250 ps for array access. Thus, when evaluating ASPEN in Section 6.8, we consider a CA running at 3.4GHz feeding tokens to our DPDA engine executing at 850MHz.

6.7 ATTACK DETECTION EVALUATION

In this section, we present an empirical evaluation of the MARTINI framework for detecting anomalous and malicious program execution at the memory access level. We focus primarily on *expressive power* and *performance*. In particular, we consider the following four research questions to validate design assumptions and investigate system-level hypotheses:

1. Can MARTINI identify program executions and differentiate between them?
2. What are the effects of compressing dictionaries?
3. Can MARTINI tell malicious from benign inputs?
4. Can MARTINI detect anomalous/malicious programs?

6.7.1 *Differentiating Programs*

We collected traces of memory accesses for each program in the Coreutils Subset (Table 6.1) and constructed a dictionary for each utility using 8-access windows from the traces (see Section 6.1.2). Then, we used execution traces from the other

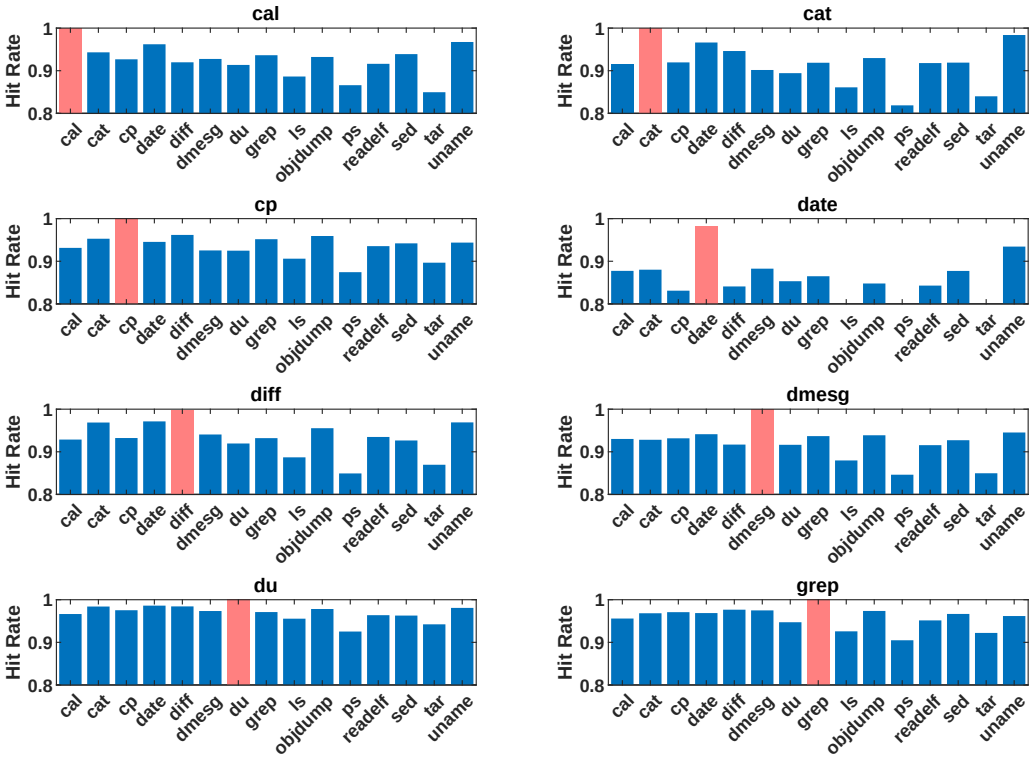


Figure 6.11: Comparison of memory traces between pairs of Linux utilities. We train a dictionary using each utility, then measure the ratio of similarities between additional traces of that utility and traces of the other utilities. Each red bar shows the result of comparing the trained utility to a subsequent execution of that same utility. Note that the red bar is approximately one for all utilities while all blue bars are less than one, demonstrating that memory traces can effectively identify programs.

utilities and measured the fraction of tested 8-access windows that are found in the trained dictionary. This experiment measures sequences of memory accesses that are the same between a trained dictionary and some subsequent test program execution. For example, we expect that a dictionary constructed from `ls` will match a high number of memory accesses in a subsequent run of `ls` on different

arguments, but will match a low number of accesses from a trace of `cat`, an entirely different program.

Figure 6.11 summarizes our findings, showing a different bar graph for each utility (the remainder show similar results and are elided for space). For each bar graph, the x-axis shows the testing program and the y-axis shows the "hit rate," or fraction of 8-access sequences in the testing program trace that matches the dictionary. We gain confidence in our assumptions if (1) testing and training on the same program shows a high hit rate (i.e., the red bar is near 1.0), and (2) testing and training on separate programs shows a low hit rate. The figure shows clear separation between these two measurements, which establishes that we can set a threshold to distinguish between different programs, based only on 8-access sequences of memory accesses.

MARTINI is able to differentiate programs from each other by observing sequences of abstracted memory accesses.

6.7.2 *Effects of Dictionary Compression*

Next, we evaluate the effectiveness of dictionary compression (Section 6.1.2.3). Recall that (1) the primary goal for the compression is minimizing the chip area required for implementation, and (2) we hypothesize that we can compress the model without significantly increasing collisions of Δ-windows, which would cause false negatives. To study this question, we compare MARTINI's accuracy at

classifying authorized vs. unauthorized program behavior, both for uncompressed and compressed dictionaries.

We built compressed and uncompressed dictionaries from traces of 12 of the Coreutils Subset programs. We then used traces from the four CVE proofs-of-concept and the four held-out Coreutils Subset to determine whether those programs would trigger alarms. In this setup, traces from the in-dictionary programs *should not* trigger alarms, while traces from the out-of-dictionary programs *should*. We measured true- and false-positive and -negative data (these results are detailed in Section 6.7.4). We found that the uncompressed dictionary yielded an AUC of 0.9995, while the compressed dictionary yielded an AUC of 0.9928, a trivial decrease in performance. Thus, we conclude that compressing dictionaries does not significantly influence classification performance.

Dictionary compression introduces minimal error in program classification, while allowing a dictionary to fit within the resource constraints of the MARTINI data path.

6.7.3 *Distinguishing Malicious from Benign Inputs*

Having established that MARTINI separates benign from anomalous behavior (e.g., `cal` from `objdump`) and validated that compression is effective, we consider malicious program inputs (e.g., `objdump` normal operation vs. an `objdump` exploit). CVE-2018-6323 describes an unsigned integer overflow in the `elf_object_p` func-

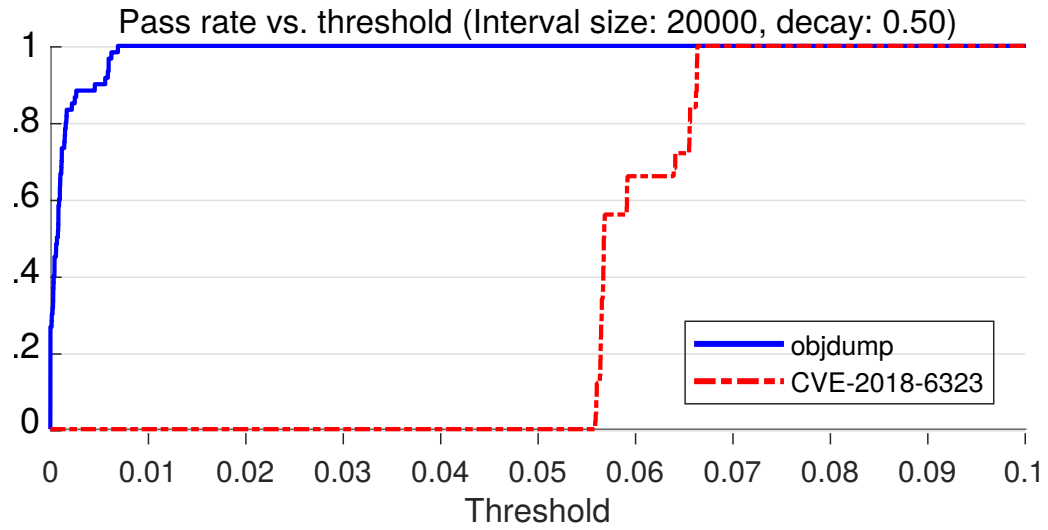


Figure 6.12: MARTINI performance on `objdump` CVE-2018-6263, as a function of threshold for benign and malicious inputs. The dictionary was trained on traces from `objdump` running on benign inputs; the vulnerability was exploited using the malicious input, and MARTINI classified the runs as benign or malicious. True negatives are shown in solid blue and false negatives in dashed red. Note that thresholds of 0.01–0.05 allow all benign runs to pass, while raising alarms on 100% of malicious runs.

tion of `objdump` that can be triggered when it reads a specially crafted ELF file. We generated a training dictionary by running `objdump` over 34 different ELF files. We evaluated with respect to three traces each of 16 benign ELF inputs and traces of the malicious ELF.

We used these to evaluate the detector's performance as a function of the threshold t chosen (see Section 6.1.3). We say that a trace *passes* if it does not trigger any alerts. Figure 6.12 plots the pass rate of the held-out benign `objdump` dictionary (i.e., true negatives, shown in solid blue) and the held-out malicious CVE (i.e., false negatives, shown in dashed red), both as a function of threshold. Note that

interval size and decay are configurable parameters discussed in Section 6.1.3. Thresholds from 0.01 to 0.05 catch *all* malicious behavior with *no* false positives.

> MARTINI distinguishes malicious from benign program behavior resulting from input given to the program with accuracy and precision.

6.7.4 *Detecting Anomalous and Malicious Programs*

We also investigate MARTINI's ability to detect anomalous and malicious programs, such as novel (unauthorized) programs or multiple exploits and attacks.

In this evaluation, we train a dictionary with 60% of all traces of 12 of our Coreutils Subset programs. The resulting model is then simultaneously subjected to four types of testing traces: (1) trained programs, (2) untrained Linux utilities (i.e., the remaining held-out Coreutils), (3) exploits of trained programs, and (4) Spectre and Meltdown proofs-of-concept. In our use case, only trained programs are considered normal; all the others are anomalous and should cause MARTINI to raise an alarm. We find that we can detect anomalous behaviors with a high true positive rate and a low false positive rate.

Figure 6.13 shows the receiver operating characteristic (ROC) curve of the results. The curve plots the true positive rate and false positive rate parametrically as a function of the threshold: each point on the curve represents a different threshold that can be chosen and thus a different trade-off in the space. A common metric to evaluate such figures is the Area Under the ROC Curve (AUC); an effective

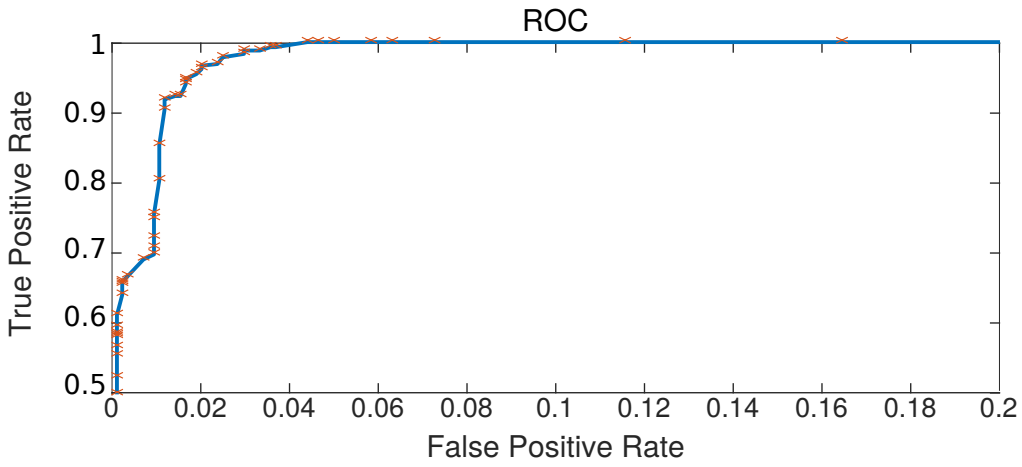


Figure 6.13: Experimental MARTINI results for tested anomalies. The ROC curve reports data for *all* our benchmarks (they are combined as they share identical shape). MARTINI's dictionary was trained using 60% of traces collected from 12 Coreutils programs, then evaluated against the held-out traces of those programs, all traces of held-out Coreutils programs, and traces from Spectre, Meltdown, and the two CVE traces. The blue line reports the average detection rate across all data points using the 12 Coreutils dictionary with 2400 program traces. AUC=0.9954.

classifier that admits many true positives and few false positives has a high AUC. Our results demonstrate that when trained on Linux utilities, MARTINI distinguishes them from other utilities, and *all* of our other benchmark security exploits and side-channel attacks powerfully, with an area-under-curve (AUC) of 0.9954. At 100% true positive rate, our smallest false positive rate was 4.4%.

We also collected PARSEC traces to act as indicative long-running processes that could be considered anomalous with respect to our Coreutils dictionary. We test if MARTINI detects anomalous behavior *early*, regardless of the trace size (i.e., detection soon after an attack, rather than minutes later). For this

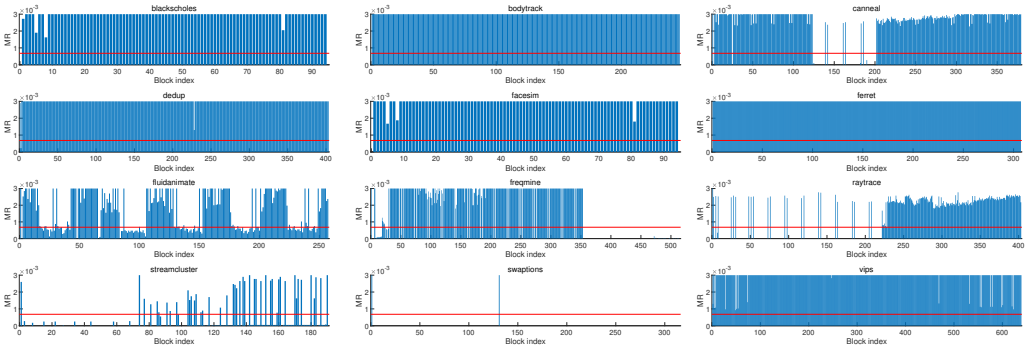


Figure 6.14: Detection of anomalous, out-of-dictionary program execution. We trained a dictionary using our Coreutils subset of 16 Linux utilities. Then, we ran each of the PARSEC benchmarks to determine if our approach would correctly identify such execution as anomalous. Because the PARSEC benchmarks are long-running, we broke their traces into *blocks* of 250 million memory accesses each. On each graph, the x-axis represents execution time in terms of these blocks, and the y-axis shows the mismatch rate, or the fraction of blocks that were not contained in the trained dictionary. The horizontal line represents a configurable threshold (set to 0.00068 in the figure). These plots suggest that we can use our approach along with a simple counter for mismatches between memory access sequences and the trained dictionary to determine when behavior is anomalous. In particular, across all benchmarks, we detect the anomalous execution within an average of 85,000 memory references. Using this simple thresholding approach, we can correctly identify all benchmarks as anomalous very early in their execution.

experiment, PARSEC traces are split into blocks with 2.5 million memory accesses each and assigned a reasonable threshold. The result is shown in Figure 6.14. The x-axis of each graph shows execution time (in block index units) and the y-axis shows the mismatch rate. The programs generally fall into two categories. Some, like `bodytrack` or `vips`, consistently trigger alarms. Others, like `fluidanimate` or `streamcluster`, trigger alarms sporadically, likely due to coincidental overlap with address sequences in the dictionary. For example, `blackscholes` and `facesim` show identical behavior because these benchmarks shared a common helper binary that

produced very similar memory access traces. Across all benchmarks, we correctly identified anomalous execution within an average of 85,000 memory references (minimum 20,000, maximum 360,000, stdev 48,400).

For this benchmark suite, we first note that in each case, anomalous behavior is detected almost immediately (i.e., the blue bar crosses the red line very far to the left on each subgraph). Second, the overall performance is quite high: 91.87% true positive rate and 2.39% false positive rate.

> MARTINI is able to distinguish anomalous and malicious program execution—including Spectre and Meltdown—with high accuracy. Further, MARTINI is capable of detecting abnormal behavior early in program execution.

6.7.5 MARTINI *Evaluation Summary*

In this section, we evaluated the MARTINI framework with respect to *expressive power* and *performance*. Our experiments provide evidence that our dictionary-based approach of modeling abstracted memory accesses provides sufficient expressiveness to differentiate programs. Further, MARTINI is able to quickly and accurately detect both malicious inputs to trained programs as well as unseen, malicious programs. Next, we evaluate our architecture for in-memory processing of DPDA.

Table 6.5: Description of grammars

LANGUAGE	DESCRIPTION	TOKEN TYPES	GRAMMAR PRODUCTIONS	PARSING AUT. STATES
Cool	Programming language	42	61	147
DOT	Graph visualization	22	53	81
JSON	Data interchange	13	19	29
XML	Data interchange	13	31	64

6.8 DPDA PROCESSING ENGINE EVALUATION

In this section, we evaluate ASPEN on real-world applications with indicative workloads. We focus on a study of parsing (one of our motivating applications from Section 2.5.1) to evaluate ASPEN with respect to *expressive power*, *scalability*, and *performance*. In particular, we are guided by the following research questions:

1. Does the underlying hDPDA computational model of ASPEN generalize to real-world parsers?
2. Do the multipop and epsilon merging optimizations improve performance?
3. What is the end-to-end performance improvement of ASPEN of state-of-the-art parsers?

Table 6.6: Grammar compilation results. Our optimizations reduce the number of epsilon states by an average of 65%.

LANGUAGE	OPTIMIZATIONS	HDPDA STATES	EPSILON STATES	AVERAGE COMPILATION TIME (SEC)
Cool	None	3505	2733	0.88
	Mutlipop + Eps	1666	894	2.75
DOT	None	1690	1494	0.34
	Mutlipop + Eps	1062	866	0.98
JSON	None	764	619	0.16
	Mutlipop + Eps	461	316	0.5
XML	None	2068	1653	0.36
	Mutlipop + Eps	865	450	0.88

6.8.1 *Parsing Generality*

We first demonstrate compilation of four different languages: *Cool*, an object-oriented programming language[6]; *DOT*, the language used by the GraphViz graph visualization tool [78]; *JSON*; and *XML*. We selected these benchmarks because grammar specifications (for either `PLY` or `Bison`) 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. Details for each of these languages, including number of token types, number of grammar rules, and the size of the parsing automaton, are provided in Table 6.5. Higher numbers of parsing automata states (see Section 6.2)

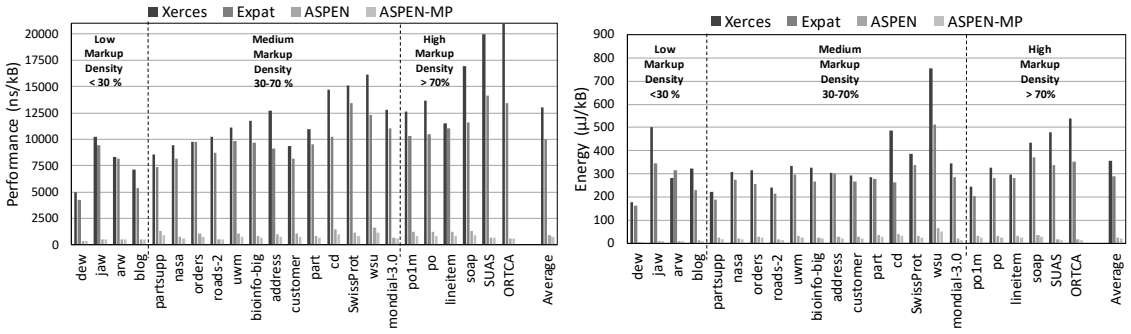
6 `https://en.wikipedia.org/wiki/Cool_(programming_language)`

indicate more complex computation for determining which grammar production rule to apply. This complexity is related both to the number of token types as well as the total number of productions in the grammar.

In Table 6.6, we present compilation statistics using our prototype compiler. We report 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. As noted in Section 6.2, reducing epsilon states reduces input stalls. Note that the numbers reported here are prior to placement and routing of the design for ASPEN. The final hDPDA may contain more states to reduce fan-in or fan-out complexity; however, the length of epsilon chains will neither increase nor decrease.

Next, we evaluate the performance of XML parsing using our compiled XML grammar. While we expect performance results to generalize, for space considerations, we do not evaluate the other parsers in detail.

The hDPDA computational model is sufficiently rich to support parsing of common serialization formats, such as JSON and XML. Further, our multipop and epsilon merging optimizations reduce the number of epsilon states in compiled hDPDA by an average of 65%.



(a) ASPEN performance (in ns/kB)

(b) ASPEN energy on SAXCount compared to Expat and Xerces

Figure 6.15: Performance and energy evaluation of ASPEN.

6.8.2 *XML Parsing Performance*

Using the graph partitioning framework METIS, we find that the XML parser hDPDA (with optimizations) maps to 8 cache arrays and results in an LLC cache occupancy of 128KB.

Figure 6.15 compares ASPEN's performance and energy against Expat and Xerces on the SAXCount application (lower is better). 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. Performance of Expat and Xerces drops as the markup density of the input XML document increases, because complex documents tend to produce a large number of tokens for verification. ASPEN also sees a slight increase in runtime with increase in markup density, but the dependence is less

pronounced. There is a noticeable trend in performance and energy 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. 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 and consumes 20.9 μJ/kB energy. When compared with Expat, a 14.1$\times$ speedup and 13.7$\times$ energy saving is achieved. ASPEN-MP also achieves 18.5$\times$ speedup and consumes 16.9$\times$ lower energy than Xerces for SAXCount. Even after considering the idle power of the CPU core, XML parsing on ASPEN takes 20.15 W, which is well within the TDP of the Xeon-E5 processor core (160 W). The low power consumed can be attributed to: (1) removal of data movement and instruction processing overheads present in a conventional core, and (2) only a single bank of the cache being active in any processing cycle, due to the deterministic nature of the automaton, resulting in energy savings.

ASPEN achieves a 14.1$\times$ speedup and 13.7$\times$ energy savings over Expat, a state-of-the-art XML parser. Further, our multipop optimization results in a 30% improvement in performance and energy over a baseline design of ASPEN.

6.8.3 *ASPEN Evaluation Summary*

In this section, we evaluated ASPEN, an architecture that repurposes a portion of LLC to support high-performance processing of DPDA. Our evaluation demonstrates that ASPEN provides sufficient expressive power to support parsing of popular data formats such as JSON and XML. Further, we demonstrate that our DPDA optimizations reduce the required hardware resources and improve runtime performance by at least 30%. Finally, ASPEN outperforms state-of-the-art XML parsers on representative workloads by a factor of at least 14×.

6.9 CHAPTER SUMMARY

In this chapter, we provide additional architectural support for automata-based computation. As part of our hardware/software co-design approach to addressing the challenges of programming hardware accelerators, we strive to ensure that the underlying architectures support common applications. In particular, we focus on case studies of detecting security attacks and parsing data, which are two common and wide-reaching application domains that benefit from hardware acceleration.

Architectural side-channel attacks such as Spectre and Meltdown potentially impact billions of devices. There is a need for techniques that efficiently and precisely identify when such attacks occur. In this chapter, we present MARTINI, an algorithmic approach for leveraging memory accesses of programs to classify behavior as authorized or anomalous. We also describe an implementation strategy

using NFAs appropriate for in-cache computation and demonstrate that Martini accurately and precisely classifies benign and malicious program execution using a suite of Coreutils programs, PARSEC benchmarks, and four CVEs.

We also present ASPEN, a general-purpose, scalable, and reconfigurable memory-centric architecture that supports rich push-down automata processing for tree-like data. We design a custom data path that performs state matching, stack update, and transition routing using memory arrays. We also develop a compiler for transforming large classes of existing grammars to pushdown automata executable on ASPEN. Our evaluation against state-of-the-art CPU tools shows that our approach is general (supporting multiple languages and serialization formats), highly performant (up to 18.5× faster for parsing), and energy efficient (up to 16.9× lower for parsing). By providing hardware support for DPDA, ASPEN brings the efficiency of recent automata acceleration approaches to a new class of applications.

Martini and ASPEN provide architectural support for the key application areas of security and data parsing, achieving improvements over the current state of the art. These architectures provide sufficient *expressive power*, *performance*, and *scalability* to support new applications. We successfully leverage automata abstractions in both architectures to bridge the gap between these high-level applications and the underlying hardware resources available in LLCs of modern CPUs.

CHAPTER 7

Conclusions

Data is being collected and analyzed at ever increasing rates, and hardware component scaling trends have resulted in a shift toward adoption of hardware accelerators in data processing pipelines. The state of support for these devices, however, is such that programming and debugging are difficult. In this dissertation, we leverage hardware/software co-design principles to develop a programming model that leverages automata abstractions to ease the adoption of hardware accelerators. We consider the thesis that:

> 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.

We provide evidence in support of this thesis by evaluating prototype tools leveraging the following insights. First, finite automata computation applies to a broad class of applications, including: natural language processing [267], network security [184], graph analytics [183], high-energy physics [240], bioinformatics [185, 186, 220], pseudo-random number generation and simulation [232], data-mining [238,

239], and machine learning [221]. Second, the compact state of finite automata enables efficient inspection and capture of relevant program state on hardware accelerators for debugging tasks. Third, finite automata map naturally, and are performant on, reconfigurable architectures [75, 209, 252]. Finally, techniques from machine learning [10], software engineering [33, 37], and formal methods [31, 222] can be combined to aid in the design of automata that bridge the gap between high-level languages and low-level hardware.

7.1 SUMMARY OF CONTRIBUTIONS

This dissertation makes five primary contributions: two front-end programming interfaces, an interactive debugger, and two automata-derived architectures. We briefly summarize each.

In Chapter 3 we develop AutomataSynth, a framework for porting certain classes of legacy source code to execute on hardware accelerators, with a focus on FPGAs. AutomataSynth is a fundamentally new approach to solving this problem and overcomes some of the von Neumann-centric limitations of current high-level synthesis techniques. We develop a novel combination of state machine learning algorithms, software verification algorithms, string decision procedures, and high-performance automata processing architectures to learn behaviorally equivalent FPGA hardware descriptions of functions deciding regular expressions. For functions with more expressive power, AutomataSynth will produce an approximate solution that is correct for all inputs shorter than some

fixed bound. Using a benchmark suite of real-world string functions mined from open-source repositories, we find that AutomataSynth constructs equivalent (or near-equivalent) hardware descriptions for more than 80% of benchmarks.

Next, we consider the problem of writing new software for hardware accelerators in Chapter 4. We develop a new, high-level programming language, RAPID, which allows developers to concisely write pattern searches over streams of input data. RAPID extends standard syntax of an imperative programming language with three domain-specific control structures. RAPID programs compile to a set of finite automata, which can then be executed efficiently across a plethora of hardware platforms. We describe mechanisms for executing RAPID programs on FPGAs, Micron's D480 AP, CPUs, and GPUs. We also provide empirical evidence that automata enable portable programming, producing fewer performance inversions across architectures than OpenCL. Using a suite of real-world applications taken from recent publications, we demonstrate that RAPID programs are significantly more compact than hand-crafted and optimized automata while maintaining similar performance and hardware resource characteristics.

We then tackle the challenge of debugging applications executing on hardware accelerators in Chapter 5. For this dissertation, we restrict attention to RAPID programs executing on Micron's AP and FPGAs. By leveraging the automata-based intermediate representation of the RAPID language, we develop (1) a compilation strategy that constructs a mapping from automata states to statements and expressions in the RAPID program, (2) mechanisms to automatically generate logic on FPGAs and repurpose hardware on the AP to extract the set of currently

active automata states on any given clock cycle, and (3) an interactive debugger that combines the previous two developments, thereby allowing a developer to set breakpoints in RAPID programs, pause and single-step RAPID programs, and inspect program variables. Using a suite of common automata processing applications, we find that our debugging hardware requires significant additional hardware resources, but maintains around 80% of clock frequencies, thus supporting high-performance processing during debugging tasks. Additionally, a human subjects study of 61 programmers each debugging ten RAPID applications found a statistically significant improvement in fault localization accuracy when using our debugger ($p = 0.013$).

Finally, in Chapter 6, we develop two new automata-based architectures to support computer security and data parsing applications. Recent discovery of hardware bugs in Intel CPUs leave millions of computers vulnerable to attacks that can leak critical system- and user-information. We design MARTINI, a novel system integration of an automata processing core to monitor memory accesses to detect such hardware attacks. We demonstrate that the automata abstraction enables real-time monitoring with sufficient fidelity to detect recent hardware attacks as well as traditional, software-based attacks. In addition to security concerns, recent trends in data processing mean that significant quantities of structured data must be parsed and analyzed. We design ASPEN, a new in-memory automata accelerator that supports parsing of recursively nested and tree-structured data. We observe that deterministic pushdown automata (DPDA) provide a suitable abstraction for representing many common serialization formats. We develop a

new, five-stage data path for executing DPDA in a tightly coupled data processing loop with a cache automaton (for tokenization) and CPU cores (for later stages of the processing pipeline). Our evaluations of Martini and ASPEN demonstrate that our new architectures (1) can detect attacks with high accuracy and (2) can parse serialized data an average of 14× more quickly than the current state of the art. We also find that reduction in cache capacity needed to implement these architectures has a statistically negligible impact on runtime performance.

Taken collectively, these contributions represent a programming model that can help ease the adoption and use of hardware accelerators for data analysis applications, while also supporting high-performance computation. We argue that the work presented in this dissertation satisfies the requirements for a suitable programming model laid out in Section 1.1:

- performance and scalability. Programs written in RAPID as well as hardware descriptions produced by AutomataSynth maintain the performance of hardware accelerators by introducing minimal overhead. Additionally, our proposed languages, tools, and architectures scale to support real-world applications.

- ease of use. The RAPID debugger improves developers' accuracy in localizing common bugs. Further, RAPID extends a familiar programming language with constructs that are natural for pattern-matching applications. Critically, our model allows developers to reuse and port existing code with AutomataSynth.

- EXPRESSIVE POWER. Both AUTOMATASYNTH and RAPID provide the full expressive power of the underlying automata abstraction, which has been demonstrated to be sufficient for many applications. We also develop a hardware data path to support parsing of tree-structured data, a more expressive computational model.

- LEGACY SUPPORT. AUTOMATASYNTH explicitly supports porting legacy applications to hardware accelerators, such as FPGAs. Additionally, the compiler we develop for ASPEN allows developers to use existing parser grammar descriptions.

7.2 A LOOK TO THE FUTURE

While the work presented in this dissertation provides improved programming support for hardware accelerators, significant room remains for understanding and characterizing the design space as well as supporting even broader classes of applications. Throughout this thesis, we have noted open challenges and future directions; we provide a high-level summary here.

We believe that programming techniques that compile or transform programs written in existing languages to current commodity hardware accelerators (e.g., FPGAs) can have high impact. For techniques that adopt a model learning approach, such as AUTOMATASYNTH in Chapter 3, we identified several open challenges. In this dissertation, we chose to learn models represented as finite automata, but there is significant opportunity to study more expressive models as well as the

principled application of approximation to this problem. Further, we exposed new opportunities for string solver optimizations and richer constraint languages. Improvements to the tooling that supports our approach can result in significantly improved performance, applicability, and scalability.

While developing debugging support for FPGAs in Chapter 5, we observed that hardware resource overheads can be nontrivial. This presents new opportunities for optimized logic design as well as potential for new FPGA architectures. Open challenges on the hardware side include dynamic selection of signals to monitor, decoupling of clocks between debugging and runtime hardware, and dedicated signal monitoring hardware to reduce overheads. On the software side, efforts to analyze programs to minimize capture and inspection of program state as well as building out support for more sophisticated debugging features (e.g., watchpoints) can further improve support for debugging on hardware accelerators. There is a need for additional scientific understanding of human factors associated with debugging on these new platforms. Better understanding of how developers use and perceive tools can help guide the development and design of future iterations of programming support.

Automata-derived architectures have been the subject of significant study in recent years. In particular, there has been interest in embedding such architectures in the memory of a system, and we presented two such architectures in Chapter 6. The work presented in this dissertation represents only an initial foray into understanding the implications of such a system design. There may be many applications that this design approach renders tractable or accelerates. Further, this

Table 7.1: Major Publications Supporting This Dissertation

VENUE	TITLE
ASPLOS '16	*RAPID Programming of Pattern- Recognition Processors* [15]
ASPLOS '19	*Debugging Support for Pattern-Matching Languages and Accelerators* [49]
ASPLOS '20	*Accelerating Legacy String Kernels via Bounded Automata Learning* [14]
CAL '18	*MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem* [12]
MICRO 51	*ASPEN: A Scalable In-SRAM Architecture for Pushdown Automata* [17]
TPDS '19	*Portable Programming with RAPID* [13]
In Preparation	MARTINI: *Memory Access Traces to Detect Attacks*

dissertation presented an initial analysis of the whole system performance impact of reducing cache capacity to embed an automata processing core. However, a systematic exploration of the design space—including prototyping—could reveal key performance trade-offs, improvements, and impacts.

7.3 FINAL REMARKS

As the adoption of hardware accelerators into data processing pipelines continues to grow, we believe that the work in this dissertation represents only a subset of the exciting area of programming support for emerging technologies. The primary findings of our work have been (or are currently being) published in prominent computer architecture, programming languages, and parallel and distributed systems research venues and journals. Table 7.1 provides a summary of these manuscripts in chronological order. As we consider the results of our research efforts, it is quite phenomenal that the lowly finite automaton—often overlooked due to its perceived limited expressive power—has such a profound and broadly

applicable benefit for bridging the gap between high-level programming languages and the low-level resources of many hardware accelerators. Researchers are only just beginning to understand how we might leverage such abstractions to ease the adoption of accelerators, and we believe there is a bright and bountiful future in their continued study.

APPENDIX A

MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem

Years of research and development have resulted in high-throughput *automata processing* architectures and software engines [75, 79, 111, 133, 204, 218, 235, 241]. This has led to the discovery of new use-cases and application domains for finite automata, such as natural language processing [267], network security [184], graph analytics [183], high-energy physics [240], bioinformatics [185, 186, 220], pseudo-random number generation and simulation [232], data-mining [238, 239], and machine learning [221].

Unfortunately, the software frameworks for the construction, manipulation, and translation of automata are frustratingly fractured (e.g. have inconsistent serialization formats) and restrictively licensed (e.g., Micron licenses a comprehensive SDK, but it is closed-source and specifically targets their D480 Automata Processor, or AP [75]). While these tools are useful for developing applications for the AP, the tools do not allow researchers to easily evaluate designs across hardware platforms, such as CPUs, GPUs, and FPGAs. The tools also cannot be easily extended to support new architectures and automata paradigms. Instead, a general and ex-

tensible framework is needed to enable the development of platform-independent applications and to support experimental automata designs.

Therefore, we have developed a suite of tools for creating, manipulating, and executing finite automata, which we refer to as MNCaRT (the MNRL Network Computation and Research Testbed, pronounced "minecart").[1] MNCaRT collects a diverse set of automata processing tools and algorithms into a central location and will grow as new tools are developed. We currently provide support for compiling state machines from Perl compatible regular expressions (PCRE) to automata, high-speed execution of NFAs and DFAs using Intel Hyperscan [111], and optimization and simulation of experimental automata designs with the Virtual Automata Simulator (VASim) [235]. Further, we provide back-ends for executing on GPUs [231], FPGAs [252], and the AP [75]. Finally, we allow users to explore routing constraints for experimental spatial architectures via the Automata-to-Routing (ATR) tool [234].

To support our ecosystem, we have created MNRL, the MNRL Network Representation Language (pronounced "mineral"), a JSON-based, open-source language to support the development of, and experimentation with, new automata-based applications and architectures. MNRL allows a user to define a *network* (or collection) of MNRL *nodes*, which represent the states within automata. Each node stores configuration information (such as node type, name, etc.) and connections to other nodes within the network. The language specification is general, allowing state machines other than finite automata to be represented. We provide initial

1 https://github.com/kevinaangstadt/mncart

definitions for traditional finite automata states, homogeneous states, up-counters, and Boolean logic in the MNRL specification; additional node types may be defined by the user for specific applications. Both MNRL and the tools in MNCaRT are publicly available (typically under BSD licenses), allowing both academics and industry experts to contribute to, and use, the ecosystem.

In summary, this appendix presents the following:

- MNCaRT, an comprehensive repository of compatible tools for developing and experimenting with automata processing on CPUs, GPUs, and FPGAs.
- MNRL, an extensible, open-source JSON specification for representing state machines.
- Python and C++ APIs for reading, creating, manipulating, and writing MNRL files.
- Extensions to Intel's Hyperscan PCRE engine, supporting compilation to and execution of MNRL files.
- An extended version of VASim, which supports reading and writing of MNRL files.

A.1 MNRL: A NEW AUTOMATA LANGUAGE

We have developed MNRL, an extensible, open-source automata representation language, which allows for the topological specification of a collection of finite

state machines using JSON syntax. While JSON is supported by most common general-purpose programming languages, we provide C++ and Python bindings to support additional validation checks.

It is important to note that the MNRL format specifies the layout of a machine but does not specify how elements behave, allowing many types of state machines to be represented, including traditional NFAs [199] and homogeneous NFAs [48].[2] Behavior is left for the execution engine to specify and implement (allowing MNRL to be an extremely flexible file format). Therefore, MNRL is similar in intent to the Unified Modeling Language (UML), in which developers describe and design software systems while eliding implementation details [85].

A.1.1 *MNRL Format*

A MNRL file contains a single MNRL *network*—a collection of one or more state machines that are executed in parallel using the same input. The file contains an array of MNRL nodes, which define each element in the network. A node consists of:

- A unique identifier
- A node type (state, homogeneous state, up counter, boolean, etc.)
- How the node is enabled

2 MNRL is general enough to represent more powerful machines (e.g. push-down automata, cellular automata, and Turing machines).

- Whether the node reports (generates an output signal) when activated
- An array of input ports, each with a unique ID and specified width (number of wires)
- An array of output ports, each with a unique ID, specified width, and list of connected nodes
- Custom attributes, specific to each element type

A developer can encode the topological layout of the state machines within the network and to specify the sort of behavior the underlying execution engine should assign to each node. The implementation of behavior is *not* defined in the MNRL file; instead, the computation engine that processes a MNRL network is responsible for specifying the semantics for each node type. Therefore, node types and execution engines are typically co-designed. If an engine needs information (e.g. symbol sets for matching against an input stream) to process a node, this configuration can be embedded in a MNRL node's attributes. For the standard node types, we have specified additional attributes to support their respective expected behaviors.

We provide the specification of MNRL as a JSON schema [112], which allows for validation of file syntax. The MNRL schema defines four node types: standard automata states (*state*), homogeneous automata states (*hState*), saturating up-counters (*upCounter*), and combinatorial logic (*boolean*). Custom attributes for each of these node types are described in Table A.1. Each of these node types defines a *reportId* attribute, which allows an additional string or integer to be

```
1 {
2   "id": "0t_15l_5r",
3   "type": "hState",
4   "enable": "onActivateIn",
5   "report": true,
6   "inputDefs": [
7     {
8       "width": 1,
9       "portId": "i"
10    }
11  ],
12  "outputDefs": [
13    {
14      "width": 1,
15      "activate": [],
16      "portId": "o"
17    }
18  ],
19
20  "attributes": {
21    "reportId": 5,
22    "latched": false,
23    "symbolSet": "[\\xFF]"
24  }
25 }
```

Listing A.1: Sample MNRL homogeneous hState node. The node is enabled (performs computation) only after an incoming edge is active (line 4), and this node matches against the input character \xFF (line 23). When this occurs, the node generates a report signal (line 5). Lines 6-11 define a single input port for incoming edges. Lines 12-18 define a single output port for outgoing edges. The array on line 15 is empty, indicating that there are no outgoing edges.

Table A.1: Custom Attributes for MNRL Node Types

Node Type	Attribute	Required?	Attribute Type	Description
state	symbolSet	YES	object	Mapping from each output port name to a symbol set string representing the matched character set that enables the outgoing connections from the given port
	latched	NO	Boolean	Determines whether a state remains enabled after the first enable signal
hState	symbolSet	YES	string	Represents the matched character set that enables the outgoing connections
	latched	NO	Boolean	Determines whether a state remains enabled after the first enable signal
upCounter	threshold	YES	number	The internal value at which the counter enables outgoing connections
	mode	YES	enum	"trigger": enable the outgoing connections for one clock cycle when the threshold is reached "high": enable the outgoing connections for all subsequent clock cycles while the internal value if at the threshold "rollover": similar to trigger, but also reset the internal value
boolean	gateType	YES	enum	Must be one of the following values: 'and', 'or', 'nor', 'not', or 'nand'

Table A.2: Modes for Enabling MNRL Nodes

Enable Mode	Description
always	The node is enabled on every cycle
onActivateIn	The node is enabled on the clock cycle following a high signal to an input port
onStartAndActivateIn	The node is enabled on the first clock cycle and then follows the "onActivateIn" mode
onLast	The node is only enabled for the final clock cycle

associated and returned with any reporting event during execution. MNRL states and hStates map directly to notions of NFA states and homogeneous NFA states. We provide upCounter and boolean node types to maintain compatibility with Micron's D480 Automata Processor [75]; however these element types are general and similar elements are in use in other engines and automata styles [29, 166]. Additionally, the MNRL schema defines four valid modes for *enabling* a node, which are similar to those used by both the AP [75] and Intel Hyperscan [111]. An enabled node performs a predefined computation on a given clock cycle. These modes are described in Table A.2.

A.1.2 *Extending the MNRL Schema*

MNRL is designed to be extensible, enabling research on new, custom automata functionality and allows researchers to quickly define custom attributes for new node types. Because custom node types become part of the JSON schema, prototype extensions to the MNRL format can still be statically checked with minimal

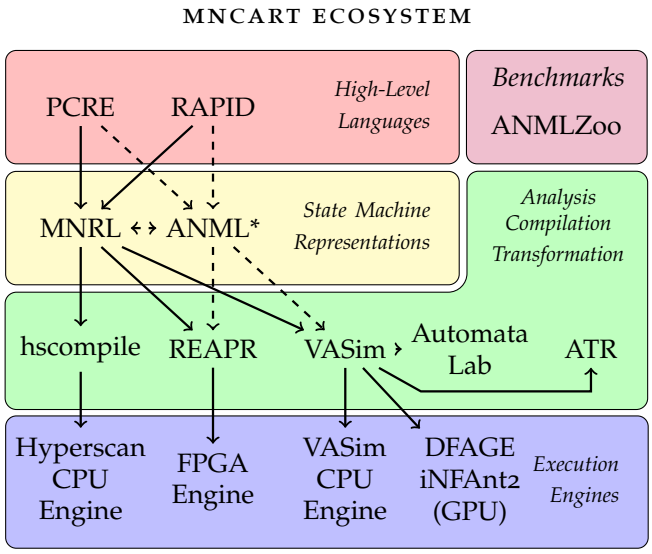
effort from the developer. The MNRL file format could easily be extended to support additional node types such as non-deterministic counters [29], and stacks (to support push-down automata). Because MNRL supports variable-width ports, it is also possible to represent elements that share more than a single bit of data with elements downstream.

A.2 THE MNCART ECOSYSTEM

Our goal with the MNRL language is to enable the development of a rich, vibrant ecosystem of compatible tools for manipulating and executing automata. We are collecting these tools in an umbrella repository, the MNRL Network Computation and Research Testbed (or MNCaRT). By keeping tools catalogued in a single location, we hope to maintain the interoperability of tools and reduce fracturing in the ecosystem. We also provide a Linux container configured to use all of the MNCaRT tools.[3]

Figure A.1 describes the interaction between tools provided with MNCaRT. Our ecosystem supports workflows beginning with high-level languages, such as PCRE, and ending with execution on CPUs, GPUs, and FPGAs. We also support execution on Micron's Automata Processor via conversion to ANML. Additionally, we provide compatible benchmarks for testing experimenting with tools in MNCaRT. In this section, we briefly describe to tools that make up the initial release of MNCaRT.

3 `https://hub.docker.com/r/kevinaangstadt/mncart`



While ANML is not officially part of MNCaRT, we indicate where this alternate representation falls within the ecosystem using dashed lines.

Figure A.1: Tools supplied as part of MNCaRT. These fall into four categories: front-end representations (both high-level and representation languages), benchmarks, transformation and compilation tools, and hardware and software execution engines.

A.2.1 *High-Level Languages*

Our framework supports programming models that represent pattern searches at a higher level of abstraction.

We compile PCRE to MNRL files using Intel Hyperscan's parsing and compilation routines [111]. Hyperscan is an open-source, high-performance regular expression processing library supported by Intel. The tool returns a graph representation of the compiled state machine, which we traverse to generate a MNRL file. Our regular expression compiler (`pcre2mnrl`) reads a file of regular expres-

sions and compiles the given set of patterns to a single MNRL file. The line number of each given PCRE pattern is used as the report ID to allow for easy identification of matched patterns in processing output.

RAPID is a high-level programming language for execution of sequential pattern-matching applications (see Chapter 4 for a full introduction to the language). This C-like language is extended with three keywords to support parallel matching of patterns against a single data stream as well as sliding window pattern recognition. The RAPID compiler can emit MNRL files, allowing for high-level programming within the MNCaRT ecosystem.

A.2.2 *Benchmarks*

The ANMLZoo benchmark suite contains a diverse set of automata applications and associated input stimuli [231]. Applications range from configurable, synthetic benchmarks to algorithms not easily represented by regular expressions and can therefore demonstrate vastly different execution characteristics. We have generated MNRL for all benchmarks in the suite.

A.2.3 *Analysis, Transformation, and Compilation*

HYPERSCAN COMPILATION. We provide an extension to Hyperscan (`hscompile`) that parses MNRL files and compiles the finite automata to a serialized Hyperscan pattern database, allowing offline compilation. Additionally, our tool serializes a

mapping from MNRL node IDs and report IDs to Hyperscan's internal naming for each state machine element. This mapping enables human-readable output when processing input data using Hyperscan.

VASIM. We have extended VASim [235] to support parsing of MNRL files. VASim is a general-purpose framework for automata simulation, optimization, transformation, and performance modeling. The tool enables prototyping, debugging, simulation, and analysis of automata-based applications and architectures. Additionally, VASim can parse Micron ANML files, allowing for conversion with MNRL.

VASim also provides a common codebase for applying state-of-the-art optimizations, transformations, and static and dynamic analyses to finite automata. This platform allows researchers to easily and quickly share new algorithms and perform fair apples-to-apples comparisons to prior work, accelerating automata processing research. We provide several optimizations in the core of VASim, including common prefix merging [28] and a literal matching engine [111].

AUTOMATA LAB. Automata Lab is a web-based graphical environment for visualizing, editing, and simulating finite automata [132]. The tool uses VASim to manipulate automata, and the resulting state machines are displayed graphically, allowing for user interaction. Users may upload MNRL files or choose from applications in the ANMLZoo benchmark suite.

REAPR. We adapt REAPR [252], a tool for generating highly efficient FPGA automata accelerator kernels, to support MNRL. The tool generalizes prior work [75, 196, 255] to be applicable for automata processing applications other than just regular expressions. Hardware automata accelerator engines such as REAPR take advantage of the one-to-one mapping between the spatial distribution of automaton states and hardware resources such as lookup tables (LUTs), block RAM (BRAM), and wires.

In REAPR, there are two main types of RTL elements to consider: 1) the state transition element (STE), which contains state activation information and transition logic; and 2) the wiring between all of the STEs in the automaton. Von Neumann automata engines iterate over every active STE and check whether the current input symbol will activate outgoing transition(s). If so, the next cycle's activation state is updated with the list of STEs that the current state affects. In an FPGA circuit generated by REAPR, STEs that affect each other are physically connected with wires, and if a single STE has multiple incoming transitions, they are combined in an OR gate so that any incoming transition can change the activation state of an STE.

AUTOMATA-TO-ROUTING. We extend the Automata-to-Routing (ATR) [234] tool to support placement and routing of MNRL state machines. ATR utilizes the Versatile Place and Route (VPR) tool to model spatial automata-processing architectures [32]. We use VASim to emit VPR-readable circuits of MNRL networks and provide guidance to construct custom, parameterizable, spatial architecture

description files to accept these custom state machine circuits. ATR is thus capable of modeling spatial architectures that are purpose-built to accept MNRL state machines.

A.2.4 *Execution Engines*

HYPERSCAN CPU ENGINE. We provide a tool (`hsrun`) for processing MNRL files against an input stream using the Hyperscan execution core. This tool deserializes the Hyperscan pattern database and node mapping produced by `hscompile`. The tool then scans the given input file against the database and prints out human-readable reporting information (e.g. MNRL ID and input stream offset). If multiple compiled MNRL files and/or input files are passed to `hsrun`, the tool will execute all pairings of the files using a supplied number of threads.

VASIM CPU ENGINE. In addition to support for transformation and analysis of finite automata, VASim supports simulation of a diverse set of finite automata models. While Hyperscan achieves higher throughput, VASim's modular design allows for quick prototyping to test new automata elements and designs, such as those including custom compute units.

FPGA ENGINE. In addition to generating hardware NFA kernels, REAPR can also generate a full platform execution environment for certain automata applications. The REAPR platform has been demonstrated to offer up to 183× speedup

over best-effort CPU implementations[252]. We are also actively developing a general-purpose reporting architecture to support execution environments for all automata kernels.

GPU ENGINES (DFAGE AND INFANT2). MNCaRT contains both a GPU-based DFA engine (DFAGE) and NFA engine (iNFAnt2). The NFA engine was described previously by Wadden et al.[231]; we therefore focus on describing DFAGE in this article. Use of DFAGE first requires compilation to one or more DFAs using VASim. Note that the compilation process is performed offline by the CPU. Often, compiling to a single DFA is inefficient. Therefore, users may partition rulesets into several DFAs, and each DFA consists of a state transition table and an acceptance vector. State transition tables corresponding to different DFAs are stored consecutively in the GPU's global memory. The same layout is applied for acceptance vectors. It should be noted that each transition table is represented by a 2-D array containing the next state identifiers for every pair of current state identifier and input symbol. Similar to previous implementations, our DFA matching engine supports multi-packets processing to take advantage of the extreme parallelism of GPU architectures. Input packets also reside in the GPU's global memory.

Workloads are mapped to a 2-D grid of threads. Similar to Yu et al. [261], different packets are mapped to different blocks on the x-dimension of grid. Each thread within the block processes a different DFA for the assigned packet. However, for large datasets in our benchmark suite where the number of DFAs

can exceed the block size, different blocks on the y-dimension of the grid will also be used.

A.3 APPENDIX SUMMARY

MNRL is a general and extensible format for representing state machines. The language specification and associated tools are released with open-source licenses to promote collaboration and usage within both academia and industry. MNRL is supported by general-purpose programming languages because it is based off of the JSON format. Further, we provide MNRL-specific APIs for Python and C++ to perform more direct manipulation and validation of networks.

MNRL is a component of MNCaRT, a suite of tools for analyzing, executing, and transforming automata processing applications. We support execution of MNRL networks on CPUs, GPUs, and FPGAs, and we provide a workflow for execution on Micron's AP. Support for high-level pattern-matching languages, such as PCRE and RAPID is also provided as part of MNCaRT. Finally, we allow for design space exploration through analysis functionality in the VASim and ATR tools.

In this dissertation, we leverage aspects of MNRL and MNCaRT for each of our main contributions. We use the MNRL format internally and as the output from AutomataSynth in Chapter 3, as output of the RAPID compiler in Chapter 4, and in an extended form to represent hDPDA in Chapter 6. We employ several resources from MNCaRT to evaluate our FPGA-based debugging framework in

Chapter 5, including the ANMLZoo benchmark suite and a customized version of REAPR. Further, research challenges encountered representing non-traditional finite automata (e.g., hDPDA in Chapter 6) informed the design of MNRL. In summary, the success of the research detailed in this dissertation was significantly supported by the development of MNRL and MNCaRT.

BIBLIOGRAPHY

[1] *2017 IEEE International Workshop on Program Debugging (IWPD), Symposium on Software Reliability Engineering Workshops, ISSRE Workshops*. IEEE Computer Society. Toulouse, France: IEEE, 2017.

[2] F. Aarts, J. De Ruiter, and E. Poll. "Formal Models of Bank Cards for Free." In: *Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013, pp. 461–468.

[3] Alfred V. Aho and Margaret J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search." In: *Commun. ACM* 18.6 (June 1975), pp. 333–340.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[5] Gustavo Alonso. "FPGAs in Data Centers." In: *Queue* 16.2 (Apr. 2018), 60:52–60:57.

[6] 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*. 2013, pp. 1–8.

[7] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. "Synthesis of Interface Specifications for Java Classes." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Long Beach, California, USA, 2005, pp. 98–109.

[8] 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*. 2010, pp. 356–359.

[9] Dana Angluin. "A note on the number of queries needed to identify regular languages." In: *Information and Control* 51.1 (1981), pp. 76 –87.

[10] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples." In: *Information and Computation* 75.2 (Nov. 1987), pp. 87–106.

[11] Dana Angluin. "Computational Learning Theory: Survey and Selected Bibliography." In: *Symposium on Theory of Computing*. 1992, pp. 351–369.

[12] 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." In: *IEEE Computer Architecture Letters* 17.1 (2018), pp. 84–87.

[13] K. Angstadt, J. Wadden, W. Weimer, and K. Skadron. "Portable Programming with RAPID." In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 939–952.

[14] Kevin Angstadt, Jean-Baptiste Jeannin, and Westley Weimer. "Accelerating Legacy String Kernels via Bounded Automata Learning." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: ACM, 2020.

[15] Kevin Angstadt, Westley Weimer, and Kevin Skadron. "RAPID Programming of Pattern-Recognition Processors." In: *Architectural Support for Programming Languages and Operating Systems*. 2016, pp. 593–605.

[16] Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron. *MNRL and MNCaRT: An Open-Source, Multi-Architecture State Machine Research and Execution Ecosystem*. Tech. rep. CS2017-01. University of Virginia, 2017.

[17] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. "ASPEN: A Scalable in-SRAM Architecture for Pushdown Automata." In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-51. Fukuoka, Japan: IEEE Press, 2018, pp. 921–932.

[18] Apache Software Foundation. *Xerces C++ XML Parser*. `http://xerces.apache.org/xerces-c/`.

[19] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. *Alma-0: An Imperative Language That Supports Declarative Programming*. Tech. rep. 1997.

[20] Krste Asanović et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 2006.

[21] 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*. 2009, pp. 219–222.

[22] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. "Relative Completeness of Abstraction Refinement for Software Model Checking." In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. 2002, pp. 158–172.

[23] Thomas Ball and Sriram K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces." In: *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*. SPIN '01. Toronto, Ontario, Canada: Springer-Verlag, 2001, pp. 103–122.

[24] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft." In: *Integrated Formal Methods*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–20.

[25] S. W. Beal. "Rapid design implementation with field-programmable gate arrays." In: *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*. 1989, pp. 487–490.

[26] David Beazley. *PLY (Python Lex-Yacc)*. `http://www.dabeaz.com/ply/index.html`.

[27] Michela Becchi. *Retular Expression Processor*. `http://regex.wustl.edu`. Accessed 2017-04-06. 2011.

[28] Michela Becchi and Patrick Crowley. "Efficient Regular Expression Evaluation: Theory to Practice." In: *Proceedings of Architectures for Networking and Communications Systems*. ANCS '08. San Jose, California, 2008, pp. 50–59.

[29] Michela Becchi and Patrick Crowley. "Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions." In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies*. CoNEXT '08. Madrid, Spain, 2008, 25:1–25:12.

[30] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[31] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. "Z3str3: A string solver with theory-aware heuristics." In: *Formal Methods in Computer Aided Design*. FMCAD 2017. 2017, pp. 55–59.

[32] Vaughn Betz and Jonathan Rose. "VPR: A new packing, placement and routing tool for FPGA research." In: *Proceedings of the International Workshop on Field Programmable Logic and Applications*. Springer. 1997, pp. 213–222.

[33] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification." In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190.

[34] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. "Predicate Abstraction with Adjustable-block Encoding." In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD '10. Lugano, Switzerland: FMCAD Inc, 2010, pp. 189–198.

[35] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The software model checker Blast." In: *International Journal on Software Tools for Technology Transfer* 9.5 (2007), pp. 505–525.

[36] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications." In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: ACM, 2008, pp. 72–81.

[37] Armin Biere. "Bounded Model Checking." In: *Handbook of Satisfiability*. 2009, pp. 457–481.

[38] M. Bishop. "What is computer security?" In: *IEEE Security Privacy* 1.1 (2003), pp. 67–69.

[39] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System." In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA: ACM, 1995, pp. 207–216.

[40] Chunkun Bo, Ke Wang, Yanjun Qi, and Kevin Skadron. "String Kernel Testing Acceleration using the Micron Automata Processor." In: *Workshop on Computer Architecture for Machine Learning*. 2015.

[41] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. "Angluin-Style Learning of NFA." In: *International Joint Conference on Artificial Intelligence*. 2009.

[42] William J. Bowhill et al. “The Xeon® Processor E5-2600 v3: a 22 nm 18-Core Product Family.” In: *J. Solid-State Circuits* 51.1 (2016), pp. 92–104.

[43] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling.” In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. 2011, pp. 70–87.

[44] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical.” In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM’03. Washington, DC: USENIX Association, 2003, p. 1.

[45] Randal E. Bryant and David R. O’Hallaron. *Computer Systems: A Programmer’s Perspective*. 2nd. USA: Addison-Wesley Publishing Company, 2010.

[46] Janusz A. Brzozowski. “Derivatives of Regular Expressions.” In: *Journal of the ACM* 11.4 (Oct. 1964), pp. 481–494.

[47] Cristian Cadar, Daniel Dunbar, and Dawson 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. San Diego, California: USENIX Association, 2008, pp. 209–224.

[48] Pascal Caron and Djelloul Ziadi. “Characterization of Glushkov automata.” In: *Theoretical Computer Science* 233.1 (2000), pp. 75–90.

[49] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. “Debugging Support for Pattern-Matching Languages and Accelerators.” In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: ACM, 2019, pp. 1073–1086.

[50] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. “Active Learning for Extended Finite State Machines.” In: *Form. Asp. Comput.* 28.2 (Apr. 2016), pp. 233–263.

[51] Adrian M. Caulfield et al. “A Cloud-scale Acceleration Architecture.” In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 7:1–7:13.

[52] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. “Rodinia: A benchmark suite for heterogeneous computing.” In: *International Symposium on Workload Characterization*. 2009, pp. 44–54.

[53] Wei Chen, Szu-Liang Chen, Siufu Chiu, Raghuraman Ganesan, Venkata Lukka, Wei Wing Mar, and Stefan Rusu. "A 22nm 2.5 MB slice on-die L3 cache for the next generation Xeon® processor." In: *Symposium on VLSI Technology*. 2013, pp. C132–C133.

[54] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. "A survey of model compression and acceleration for deep neural networks." In: *arXiv preprint arXiv:1710.09282* (2017).

[55] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Optimizing Database-Backed Applications with Query Synthesis." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 3–14.

[56] Noam Chomsky and George A. Miller. "Introduction to the Formal Analysis of Natural Languages." In: *Handbook of Mathematical Psychology*. Vol. 2. 1963. Chap. 11, pp. 269–322.

[57] Robert G. Clapp, Haohuan Fu, and Olav Lindtjorn. "Selecting the right hardware for reverse time migration." In: *The Leading Edge* 29.1 (2010), pp. 48–58. eprint: https://doi.org/10.1190/1.3284053.

[58] James Clark. *The Expat XML Parser*. http://expat.sourceforge.net.

[59] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Wenyuan Xu, and Kevin Fu. "WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices." In: *Presented as part of the 2013 USENIX Workshop on Health Information Technologies*. Washington, D.C.: USENIX, 2013.

[60] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided Abstraction Refinement for Symbolic Model Checking." In: *Journal of the ACM* 50.5 (Sept. 2003), pp. 752–794.

[61] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. "An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems." In: *Architectural Support for Programming Languages and Operating Systems*. 2016, pp. 577–589.

[62] Computer Sciences Corporation. *Big Data Universe Beginning to Explode*. http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode. 2012.

[63] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. "Inception: System-Wide Security Testing of Real-World Embedded Systems Software." In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 309–326.

[64] *CortexTM-M1 Technical Reference Manual*. r1p0. ARM Limited. 2008.

[65] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. "A Large-scale Analysis of the Security of Embedded Firmwares." In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 95–110.

[66] William Craig. "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory." In: *The Journal of Symbolic Logic* 22.3 (1957), pp. 269–285.

[67] 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.

[68] Zefu Dai, Nick Ni, and Jianwen Zhu. "A 1 cycle-per-byte XML parsing accelerator." In: *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2010, pp. 199–208.

[69] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security." In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[70] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. "RAPL: Memory power estimation and capping." In: *International Symposium on Low-Power Electronics and Design*. 2010.

[71] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[72] Joeri De Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations." In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Washington, D.C.: USENIX Association, 2015, pp. 193–206.

[73] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. "On the feasibility of online malware detection with performance counters." In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 559–570.

[74] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs." In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[75] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing." In: *IEEE Transactions on Parallel and Distributed Systems* 25.12 (2014), pp. 3088–3098.

[76] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. "Tappan zee (north) bridge: mining memory accesses for introspection." In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 839–850.

[77] ECMA Technical Committee 39. *The JSON Data Interchange Format*. Tech. rep. ECMA-404 1st Edition. ECMA, Oct. 2013.

[78] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. "Graphviz — open source graph drawing tools." In: *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 483–484.

[79] 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 '15. 2015, pp. 533–545.

[80] Yuanwei Fang, Chen Zou, Aaron J Elmore, and Andrew A Chien. "UDP: a programmable accelerator for extract-transform-load workloads and more." In: *International Symposium on Microarchitecture*. ACM. 2017, pp. 55–68.

[81] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostiundefined. "Make the Most out of Last Level Cache in Intel Processors." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019.

[82] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. "Combining Model Learning and Model Checking to Analyze TCP Implementations." In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 454–471.

[83] Michael Flynn. "Flynn's Taxonomy." In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 689–697.

[84] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. "A Sense of Self for Unix Processes." In: *Proceedings of the IEEE Symposium on Security and Privacy*. SP '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 120–.

[85] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, 2004.

[86] Z. P. Fry and W. Weimer. "A human study of fault localization accuracy." In: *International Conference on Software Maintenance*. 2010, pp. 1–10.

[87] Matthew M. Geller, Michael A. Harrison, and Ivan M. Havel. "Normal forms of deterministic grammars." In: *Discrete Mathematics* 16.4 (1976), pp. 313 –321.

[88] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis." In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461.

[89] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. "Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach." In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 139–153.

[90] J. Goeders and S. J. E. Wilton. "Effective FPGA debug for high-level synthesis generated circuits." In: *24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8.

[91] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. "HARE: Hardware accelerator for regular expressions." In: *International Symposium on Microarchitecture*. 2016, pp. 1–12.

[92] 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*. 2001, pp. 41–50.

[93] Sheila A. Greibach. "A New Normal-Form Theorem for Context-Free Phrase Structure Grammars." In: *J. ACM* 12.1 (Jan. 1965), pp. 42–52.

[94] Sumit Gulwani. "Programming by Examples: Applications, Algorithms, and Ambiguity Resolution." In: *Proceedings of the 8th International Joint Conference on Automated Reasoning*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 9–14.

[95] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. "A Survey of FPGA-Based Neural Network Inference Accelerators." In: *ACM Trans. Reconfigurable Technol. Syst.* 12.1 (2019).

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

[97] Tegze P. Haraszti. "Sense Amplifiers." In: *CMOS Memory Circuits*. Boston, MA: Springer US, 2002, pp. 163–275.

[98] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell, Third Edition*. O'Reilly Media, Inc., 2004.

[99] Michael A. Harrison and Ivan M. Havel. "Real-Time Strict Deterministic Languages." In: *SIAM Journal on Computing* 1.4 (1972), pp. 333–349. eprint: `https://doi.org/10.1137/0201024`.

[100] Douglas M. Hawkins. "The Problem of Overfitting." In: *Journal of Chemical Information and Computer Sciences* 44.1 (2004). PMID: 14741005, pp. 1–12. eprint: `https://doi.org/10.1021/ci0342472`.

[101] Urs Hölzle, Craig Chambers, and David Ungar. "Debugging Optimized Code with Dynamic Deoptimization." In: *Programming Language Design and Implementation*. San Francisco, California, USA, 1992, pp. 32–43.

[102] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. "MapCG: Writing Parallel Program Portable Between CPU and GPU." In: *Parallel Architectures and Compilation Techniques*. Vienna, Austria, 2010, pp. 217–226.

[103] Pieter Hooimeijer and Westley Weimer. "A decision procedure for subset constraints over regular languages." In: *Programming Language Design and Implementation (PLDI)*. 2009, pp. 188–198.

[104] Qiming Hou, Kun Zhou, and Baining Guo. "Debugging GPU Stream Programs Through Automatic Dataflow Recording and Visualization." In: *SIGGRAPH Asia*. 2009, 153:1–153:11.

[105] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. "An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel® Xeon® Processor E5 Family." In: *J. Solid-State Circuits* 48.8 (2013), pp. 1954–1962.

[106] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. "Verifying web applications using bounded model checking." In: *International Conference on Dependable Systems and Networks*. IEEE. 2004, pp. 199–208.

[107] Jennifer Huffstetler. *Intel Processors and FPGAs——Better Together*. `https://itpeernetwork.intel.com/intel-processors-fpga-better-together`. Accessed 2020-02-04. 2018.

[108] INRIA. *Lexer and parser generators (ocamllex, ocamlyacc)*. `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html`. accessed 2018-04-06.

[109] *Integrated Logic Analyzer v6.2: LogiCORE IP Product Guide*. PG172. Xilinx Inc. San José, CA, 2016.

[110] Intel. *Cache Allocation Technology*. 2017.

[111] Intel. *Hyperscan*. `https://01.org/hyperscan`. Accessed 2017-04-07. 2017.

[112] Internet Engineering Task Force. *JSON Schema: core definitions and terminology*. json-schema-core. Jan. 2013.

[113] M Isberner. "Foundations of active automata learning: an alorithmic perspective." PhD thesis. Technical University of Dortmund, 2015.

[114] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning." In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Cham: Springer International Publishing, 2014, pp. 307–322.

[115] Ranjit Jhala and Rupak Majumdar. "Software Model Checking." In: *ACM Computing Surveys* 41.4 (Oct. 2009), 21:1–21:54.

[116] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013.

[117] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 1–12.

[118] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. "Verified Lifting of Stencil Computations." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 711–726.

[119] Yusaku Kaneta, Shingo Yoshizawa, SI Minato, and Hiroki Arimura. "High-Speed String and Regular Expression Matching on FPGA." In: *Proceedings of the Asia-Pacific Signal and Information Processing Association*. 2011.

[120] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. "JavaSMT: A Unified Interface for SMT Solvers in Java." In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Sandrine Blazy and Marsha Chechik. Cham: Springer International Publishing, 2016, pp. 139–148.

[121] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs." In: *SIAM J. Scientific Computing* 20.1 (1998), pp. 359–392.

[122] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. "Quantifying and improving the efficiency of hardware-based mobile malware detectors." In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press. 2016, p. 37.

[123] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. Cambridge, MA, USA: MIT Press, 1994.

[124] Peter B. Kessler. "Fast Breakpoints: Design and Implementation." In: *Programming Language Design and Implementation*. White Plains, New York, USA, 1990, pp. 78–84.

[125] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher 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. Carlsbad, CA, USA: USENIX Association, 2018, pp. 107–127.

[126] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. "Expositor: Scriptable Time-travel Debugging with First-class Traces." In: *International Conference on Software Engineering*. San Francisco, CA, USA: IEEE Press, 2013, pp. 352–361.

[127] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. "HAMPI: a solver for string constraints." In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM. 2009, pp. 105–116.

[128] G. Knittel, S. Mayer, and C. Rothlaender. "Integrating Logic Analyzer Functionality into VHDL Designs." In: *2008 International Conference on Reconfigurable Computing and FPGAs*. 2008, pp. 127–132.

[129] Andrew J. Ko and Brad A. Myers. "Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior." In: *International Conference on Software Engineering*. Leipzig, Germany, 2008, pp. 301–310.

[130] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[131] Ron Kohavi. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence—Volume 2*. IJCAI'95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.

[132] Dan Kramp, Jack Wadden, and Kevin Skadron. *Automata Lab: An Open-Source Automata Visualization, Simulation, and Manipulation Tool*. Tech. rep. CS2017-03. University of Virginia, 2017.

[133] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. "Hardware acceleration in the IBM PowerEN processor: Architecture and performance." In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. 2012, pp. 389–400.

[134] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. "Are We There Yet? A Study on the State of High-Level Synthesis." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (2019), pp. 898–911.

[135] A. C. Lear. "XML seen as integral to application integration." In: *IT Professional* 1.5 (1999), pp. 12–16.

[136] T. J. Leblanc and J. M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay." In: *IEEE Transactions on Computers* C-36.4 (1987), pp. 471–482.

[137] D. Lee and M. Yannakakis. "Principles and methods of testing finite state machines-a survey." In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123.

[138] John Levine and Levine John. *Flex & Bison*. 1st. O'Reilly Media, Inc., 2009.

[139] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. "Intrusion detection system: A comprehensive review." In: *Journal of Network and Computer Applications* 36.1 (2013), pp. 16 –24.

[140] Anthony W Lin and Pablo Barceló. "String solving with word equations and transducers: towards a logic for analysing mutation XSS." In: *ACM SIGPLAN Notices*. Vol. 51. 1. ACM. 2016, pp. 123–136.

[141] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. "Parabix: Boosting the efficiency of text processing on commodity processors." In: *International Symposium on High Performance Computer Architecture*. 2012, pp. 373–384.

[142] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown." In: *arXiv preprint arXiv:1801.01207* (2018).

[143] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. "Last Level Cache side-channel attacks are practical." In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 605–622.

[144] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation." In: *ACM SIGPLAN Notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[145] Robert W.P. Luk, H.V. Leong, Tharam S. Dillon, Alvin T.S. Chan, W. Bruce Croft, and James Allan. "A survey in indexing and searching XML documents." In: *Journal of the American Society for Information Science and Technology* 53.6 (2002), pp. 415–437. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/asi.10056.

[146] Jan van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. "Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator." In: *International Symposium on Microarchitecture*. 2012, pp. 461–472.

[147] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. "Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms." In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 517–528.

[148] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. "Efficient Test-based Model Generation for Legacy Reactive Systems." In: *Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*. HLDVT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 95–100.

[149] Norman Matloff and Peter Jay Salzman. *The Art of Debugging with GDB, DDD, and Eclipse*. San Francisco, CA, USA: No Starch Press, 2008.

[150] Sergey Maximov. *Performance Implications of the Meltdown and Spectre Fixes*. https://www.virtuozzo.com/connect/details/blog/view/performance-implications-of-the-meltdown-and-spectre-fixes.html. 2018.

[151] Kenneth L. McMillan. "Lazy Abstraction with Interpolants." In: *Proceedings of the 18th International Conference on Computer Aided Verification*. CAV'06. Seattle, WA: Springer-Verlag, 2006, pp. 123–136.

[152] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 691–701.

[153] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. "Helium: Lifting High-Performance Stencil Kernels from Stripped X86 Binaries to Halide DSL Code." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 391–402.

[154] *MicroBlaze Micro Controller System v3.0*. PG116. Xilinx Inc. San José, CA, 2019.

[155] Micron Technoloy. *Calculating Hamming Distance*. `http://www.micronautomata.com/documentation/cookbook/c_hamming_distance.html`.

[156] Sparsh Mittal. "A Survey of Techniques for Approximate Computing." In: *ACM Comput. Surv.* 48.4 (Mar. 2016), 62:1–62:33.

[157] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. "Learning nominal automata." In: *Principles of Programming Languages (POPL)*. 2017, pp. 613–625.

[158] Martin Monperrus. "Automatic software repair: a bibliography." In: *ACM Computing Surveys (CSUR)* 51.1 (2018), p. 17.

[159] J. S. Monson. "Using Source-to-Source Transformations to Add Debug Observability to HLS-Synthesized Circuits." PhD thesis. Brigham Young University, 2016.

[160] Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie D. Enright Jerger, and Adrian Sampson. "A Taxonomy of General Purpose Approximate Computing Techniques." In: *Embedded Systems Letters* 10.1 (2018), pp. 2–5.

[161] R. Nane et al. "A Survey and Evaluation of FPGA High-Level Synthesis Tools." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604.

[162] R. Nathuji, C. Isci, and E. Gorbatov. "Exploiting Platform Heterogeneity for Power Efficient Data Centers." In: *Fourth International Conference on Autonomic Computing (ICAC'07)*. 2007, pp. 5–5.

[163] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs." In: *Compiler Construction*. Ed. by R. Nigel Horspool. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 213–228.

[164] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. "SemFix: Program Repair via Semantic Analysis." In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE, 2013, pp. 772–781.

[165] Zhenyu Ning and Fengwei Zhang. "Understanding the security of ARM debugging features." In: *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[166] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. "Demystifying Automata Processing: GPUs, FPGAs, or Micron's AP?" In: *Proceedings of the International Conference on Supercomputing*. ICS '17. Chicago, Illinois: ACM, 2017, 1:1–1:11.

[167] Oracle. *OpenSPARC T1*. `https://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html`. Accessed 2020-02-04.

[168] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES." In: *Cryptographers' track at the RSA conference*. Springer. 2006, pp. 1–20.

[169] Peizhao Ou and Brian Demsky. "Checking Concurrent Data Structures Under the C/C++11 Memory Model." In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '17. Austin, Texas, USA: Association for Computing Machinery, 2017, pp. 45–59.

[170] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. "FlexJava: language support for safe and modular approximate programming." In: *Foundations of Software Engineering (ESEC/FSE)*. 2015, pp. 745–757.

[171] Chris Parnin and Alessandro Orso. "Are Automated Debugging Techniques Actually Helping Programmers?" In: *International Symposium on Software Testing and Analysis*. Toronto, Ontario, Canada, 2011, pp. 199–209.

[172] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[173] Douglas L. Perry. *VHDL (2nd Ed.)* USA: McGraw-Hill, Inc., 1993.

[174] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. 2nd. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[175] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services." In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 13–24.

[176] 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*. 2014, pp. 110–119.

[177] David Reinsel, John Gantz, and John Rydning. *Data Age 2025: The Digitization of the World from Edge to Core*. White Paper US44413318. IDC, 2018.

[178] Martin C. Rinard. "Acceptability-oriented computing." In: *Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*. 2003, pp. 221–239.

[179] R.L. Rivest and R.E. Schapire. "Inference of Finite Automata Using Homing Sequences." In: *Information and Computation* 103.2 (1993), pp. 299 –347.

[180] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes. "Features, Design Tools, and Application Domains of FPGAs." In: *IEEE Transactions on Industrial Electronics* 54.4 (2007), pp. 1810–1823.

[181] Antonio Roldao and George A. Constantinides. "A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation for Dense Matrices." In: *ACM Trans. Reconfigurable Technol. Syst.* 3.1 (Jan. 2010).

[182] P. Romero, B. du Boulay, R. Lutz, and R. Cox. "The effects of graphical and textual visualisations in multi-representational debugging environments." In: *Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*. 2003, pp. 236–238.

[183] 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. 2016, pp. 283–292.

[184] 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. 2016, pp. 1123–1132.

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

[186] Indranil Roy and Srinivas Aluru. "Finding Motifs in Biological Sequences Using the Micron Automata Processor." In: *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*. 2014, pp. 415–424.

[187] S. Sarkar, T. Majumder, A. Kalyanaraman, and P. P. Pande. "Hardware accelerators for biocomputing: A survey." In: *International Symposium on Circuits and Systems*. 2010, pp. 3789–3792.

[188] E. Satterthwaite. "Debugging tools for high level languages." In: *Software: Practice and Experience* 2.3 (1972), pp. 197–217.

[189] Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. "Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report." In: *Proceedings of the 12th International Conference on Integrated Formal Methods - Volume 9681*. IFM 2016. Reykjavik, Iceland: Springer-Verlag, 2016, pp. 311–325.

[190] M.L. Scott. *Programming Language Pragmatics*. Elsevier Science, 2015.

[191] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[192] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors." In: *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. Oakland, 2001.

[193] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the Effectiveness of Address-space Randomization." In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: ACM, 2004, pp. 298–307.

[194] J. M. Shalf and R. Leland. "Computing beyond Moore's Law." In: *IEEE Computer* 48.12 (2015), pp. 14–23.

[195] 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*. 2014, pp. 97–108.

[196] 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)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 227–238.

[197] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. "On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters." In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '17. Abu Dhabi, United Arab Emirates: ACM, 2017, pp. 483–493.

[198] Jawar Singh, Saraju P. Mohanty, and Dhiraj K. Pradhan. "Introduction to SRAM." In: *Robust SRAM Designs and Analysis*. New York, NY: Springer New York, 2013, pp. 1–29.

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

[200] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. "Sketching Concurrent Data Structures." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 136–148.

[201] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial Sketching for Finite Programs." In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 404–415.

[202] Anil Somayaji and Stephanie Forrest. "Automated Response Using System-Call Delays." In: *Proceedings of the 9th USENIX Security Symposium*. Denver, CO, 2000.

[203] Tamim Sookoor, Timothy Hnat, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. "Macrodebugging: Global Views of Distributed Program Execution." In: *Conference on Embedded Networked Sensor Systems*. Berkeley, California, 2009, pp. 141–154.

[204] Ioannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis. "Regular Expression Matching in Reconfigurable Hardware." In: *Journal of Signal Processing Systems* 51.1 (2008), pp. 99–121.

[205] Eric Spishak, Werner Dietl, and Michael D. Ernst. "A Type System for Regular Expressions." In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. FTfJP '12. Beijing, China, 2012, pp. 20–26.

[206] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*. Free Software Foundation, 2002.

[207] Bernhard Steffen, Falk Howar, and Maik Merten. "Introduction to Active Automata Learning from a Practical Perspective." In: *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Ed. by Marco Bernardo and Valérie Issarny. SFM 2011. Bertinoro, Italy: Springer Berlin Heidelberg, 2011, pp. 256–296.

[208] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems." In: *Computing in Science Engineering* 12.3 (2010), pp. 66–73.

[209] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. "Cache Automaton." In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50. Cambridge, Massachusetts: ACM, 2017, pp. 259–272.

[210] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. "Secure Program Execution via Dynamic Information Flow Tracking." In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XI. Boston, MA, USA: ACM, 2004, pp. 85–96.

[211] Audie Sumaray and S. Kami Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform." In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12. Kuala Lumpur, Malaysia: Association for Computing Machinery, 2012.

[212] Prateek Tandon, Faissal M. Sleiman, Michael J. Cafarella, and Thomas F. Wenisch. "HAWK: Hardware support for unstructured log processing." In: *International Conference on Data Engineering*. 2016, pp. 469–480.

[213] J. Teich. "Hardware/Software Codesign: The Past, the Present, and Predicting the Future." In: *Proceedings of the IEEE* 100.Special Centennial Issue (2012), pp. 1411–1430.

[214] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting Performance Data with PAPI-C." In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[215] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. "StreamIt: A Language for Streaming Applications." In: *International Conference on Compiler Construction*. Springer-Verlag, 2002, pp. 179–196.

[216] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[217] Cesare Tinelli, Clark Barrett, and Pascal Fontaine. *SMT-LIB 2.6 Strings Theory: Draft 2.1*. Tech. rep. Department of Computer Science, The University of Iowa, 2019.

[218] Titan IC Systems. *Helios RXPF Soft IP for FPGA Security Analytics Acceleration*. `http://titan-ic.com/products/helios-rxpf`. Accessed 2017-04-05. 2017.

[219] 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.* 2003, pp. 705–711.

[220] Tommy Tracy II, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. "Nondeterministic Finite Automata in Hardware—the Case of the Levenshtein Automaton." In: *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[221] Tommy Tracy II, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. "Towards Machine Learning on the Automata Processor." In: *Proceedings of ISC High Performance Computing*. 2016, pp. 200–218.

[222] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. "S3: A Symbolic String Solver for Vulnerability Detection in Web Applications." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1232–1243.

[223] L. Di 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)*. 2017, pp. 423–426.

[224] David Ungar, Henry Lieberman, and Christopher Fry. "Debugging and the Experience of Immediacy." In: *Communications of the ACM* 40.4 (Apr. 1997), pp. 38–43.

[225] Frits Vaandrager. "Model Learning." In: *Communications of the ACM* 60.2 (Jan. 2017), pp. 86–95.

[226] L. G. Valiant. "A Theory of the Learnable." In: *Commun. ACM* 27.11 (Nov. 1984), pp. 1134–1142.

[227] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution." In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 991–1008.

[228] *Virtex-4 Family Overview*. DS112 (v3.1). Xilinx Inc. San José, CA, 2010.

[229] *Virtual Input/Output v3.0: LogiCORE IP Product Guide*. PG159. Xilinx Inc. San José, CA, 2018.

[230] *Vivado Design Suite User Guide: Programming and Debugging*. UG908(v2018.1). Xilinx Inc. San José, CA, 2018.

[231] J. Wadden et al. "ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures." In: *International Symposium on Workload Characterization*. IISWC '16. 2016, pp. 1–12.

[232] 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)*. 2016, pp. 622–629.

[233] Jack Wadden, Kevin Angstadt, and Kevin Skadron. "Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures." In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 749–761.

[234] 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.

[235] Jack Wadden and Kevin Skadron. *VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research*. Tech. rep. CS2016-03. University of Virginia, 2016.

[236] Peter J. L. Wallis. "The design of a portable programming language." In: *Acta Informatica* 10.2 (1978), pp. 157–167.

[237] Steven Walton. *Patched Desktop PC: Meltdown and Spectre Benchmarked*. `https://www.techspot.com/article/1556-meltdown-and-spectre-cpu-performance-windows/page4.html`. 2018.

[238] 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 '16. Como, Italy: ACM, 2016, pp. 135–144.

[239] Ke Wang, Mircea Stan, and Kevin Skadron. "Association Rule Mining with the Micron Automata Processor." In: *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*. 2015.

[240] 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." In: *Nuclear Instruments and Methods in Physics Research* (2016).

[241] Xiang Wang. "Techniques for Efficient Regular Expression Matching Across Hardware Architectures." MA thesis. University of Missouri-Columbia, 2014.

[242] Z. Wang and R. B. Lee. "Covert and Side Channels Due to Processor Architecture." In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 2006, pp. 473–482.

[243] Shijia Wei, Aydin Aysu, Michael Orshansky, Andreas Gerstlauer, and Mohit Tiwari. "Using Power-Anomalies to Counter Evasive Micro-Architectural Attacks in Embedded Systems." In: *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2019, pp. 111–120.

[244] Mark Weiser. "Programmers Use Slices when Debugging." In: *Communications of the ACM ACM* 25.7 (July 1982), pp. 446–452.

[245] Gail Weiss, Yoav Goldberg, and Eran Yahav. "Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples." In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, 2018, pp. 5247–5256.

[246] Timothy Wheeler, Paul Graham, Brent E. Nelson, and Brad 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. London, UK, UK: Springer-Verlag, 2001, pp. 483–492.

[247] Titus Winters, Hyrum Wright, and Tom Manshreck. *Software Engineering at Google: Lessons Learned from Programming over Time*. O'Reilly Media, 2020.

[248] Loring Wirbel. *Xilinx SDAccel: A Unified Development Environment for Tomorrow's Data Center*. Tech. rep. The Linley Group, 2014.

[249] Jacob O. Wobbrock, Leah Findlater, Darren Gergle, and James J. Higgins. "The Aligned Rank Transform for Nonparametric Factorial Analyses Using Only Anova Procedures." In: *Conference on Human Factors in Computing Systems*. Vancouver, BC, Canada, 2011, pp. 143–146.

[250] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. "A survey on software fault localization." In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

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

[252] 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. 2017, pp. 1–8.

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

[254] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. "ReplayConfusion: Detecting Cache-based Covert Channel Attacks Using Record and Replay." In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 39:1–39:14.

[255] Y. H. Yang and V. Prasanna. "High-Performance and Compact Architecture for Regular Expression Matching on FPGA." In: *IEEE Transactions on Computers* 61.7 (2012), pp. 1013–1025.

[256] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. "Compact Architecture for High-throughput Regular Expression Matching on FPGA." In: *Symposium on Architectures for Networking and Communications Systems*. 2008, pp. 30–39.

[257] Yuval Yarom and Katrina Falkner. "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack." In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

[258] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong. "Map-reduce as a Programming Model for Custom Computing Machines." In: *International Symposium on Field-Programmable Custom Computing Machines*. 2008, pp. 149–159.

[259] Cemal Yilmaz, Amit Paradkar, and Clay Williams. "Time Will Tell: Fault Localization Using Time Spectra." In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 81–90.

[260] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. "How do fixes become bugs?" In: *Foundations of Software Engineering*. 2011, pp. 26–36.

[261] Xiaodong Yu and Michela Becchi. "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space." In: *Proceedings of the ACM International Conference on Computing Frontiers*. CF '13. Ischia, Italy: ACM, 2013, 18:1–18:10.

[262] Polle Trescott Zellweger. "Interactive Source-level Debugging for Optimized Programs (Compilation, High-level)." PhD thesis. University of California, Berkeley, 1984.

[263] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. "Using Hardware Features for Increased Debugging Transparency." In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 55–69.

[264] Tianyi Zhang and Miryung Kim. "Automated Transplantation and Differential Testing for Clones." In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 665–676.

[265] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. "How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation." In: *Compiler Construction*. Berlin, Heidelberg, 2008, pp. 147–162.

[266] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. "Mining API Mapping for Language Migration." In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 195–204.

[267] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. "Brill tagging on the Micron Automata Processor." In: *Proceedings of the 9th IEEE International Conference on Semantic Computing*. 2015, pp. 236–239.

[268] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs." In: *High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, Utah, 2016, 35:1–35:12.

[269] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press, 2010.