Improving Programming Support for Hardware Accelerators Through Automata Processing Abstractions

PhD Dissertation Proposal

Kevin Angstadt
angstadt@umich.edu

18. December 2018
By 2020, it’s estimated that for every person on earth, 1.7MB of data will be created every second.

DOMO, “Data Never Sleeps 6.0”. 2018
Physical Limits Spark Creativity

Hardware accelerators are seen as a viable path forward for tackling increasing compute demands.


Source: MIT Technology Review

Source: Dazeinfo

18. December 2018  K. Angstadt – PhD Proposal
How Accelerators Help

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018
Significant Interest from Industry

Significant Interest from Industry

Lack of [Good] Programming Models

- Akin to “assembly-level” programming on CPU architectures
  - Are circuit design and digital logic concepts in CS curricula?
- Require low-level knowledge of architectural design to produce performant code
- Difficult to debug and maintain: oscilloscopes and logic analyzers
- Many efforts to improve
  - OpenCL, Xilinx SDAccel, etc.
  - High-level language + annotations + decent performance
  - Non-intuitive impact of high-level implementation on performance
Successful Programming Models

• **Performance and Scalability:** minimize overhead introduced by high-level programming models and tools.

• **Ease of Use:** provide familiar abstractions and a shallow learning curve.

• **Expressive Power:** support the applications that developers wish to accelerate with dedicated hardware.

• **Legacy Support:** support the adaptation of existing software to execute efficiently on hardware accelerators while placing a minimal burden on developers.
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.
Automata Processing in the Big Data World

- Detecting Intrusion Attempts in Network Packets
- Learning Association Rules with an \textit{a priori} approach
- Detecting incorrect POS tags in NLP
- Looking for Virus Signatures in Binary Data
- Detecting Higgs Events in Particle Collider Data
- Aligning DNA Fragments to the Human Genome
Finite Automata: 10,000ft View

Key

Active Searches (Automata)

Target Pattern

Incoming Data

Matching patterns trigger reports

ATCGA

CGGCAT
Homogeneous Finite Automata

- Finite set of states with transitions operating over a finite alphabet
- Input data processed by repeatedly applying transition rules
- **Non-determinism**: multiple transitions on single input
- **Homogeneity**: all incoming transitions occur on the same input character
Homogeneous Finite Automata

State Transition Element (STE): a state in a homogeneous NFA

- Non-determinism: multiple transitions on single input
- Homogeneity: all incoming transitions occur on the same input character
Proposal Overview

• Proposed Research Efforts
  • High-Level Programming Language: RAPID
  • High-Speed, Interactive Debugger for Hardware Accelerators
  • In-Cache Accelerator for Pushdown Automata
  • Adapting Legacy Code for Execution on Hardware Accelerators

• Candidate Schedule

• Conclusion/Discussion
High-Level Languages for Automata Processing

Research Effort 1
Research Effort 1

Goal: establish the feasibility of compiling an imperative, high-level programming language to a set of finite automata for execution on hardware accelerators

Hypothesis: A high-level programming language will improve the conciseness of representing an algorithm while maintaining the performance of hand-crafted applications for hardware accelerators
RAPID at a Glance

• Provides concise, maintainable, and efficient representations for pattern-identification algorithms
• Conventional, C- or Java-style language with domain-specific parallel control structures
• Excels in applications where patterns are best represented as a combination of text and computation
• Compilation strategy supports execution on AP, FPGAs, CPUs, and GPUs
Domain-Specific Code Abstraction

Key

Active Searches (Automata)

Target Pattern

…

G C T G A C C C A T

Incoming Data

Matching patterns trigger reports

…

T

ATCGA

CGGCAT
Domain-Specific Code Abstraction

Network

Macros
Parallel Control Structures

• Concise specification of multiple, simultaneous comparisons against a single data stream
• Support common pattern search paradigms
• Static and dynamic thread spawning for massive parallelism support
• Explicit support for sliding window computations
Multi-Architecture System Overview

Focus of this research effort

RAPID Program → RAPID Compiler → Automata → Hyperscan Compiler → CPU Engine

RAPID Program → RAPID Compiler → Automata → VASim → GPU Output

RAPID Program → RAPID Compiler → Automata → REAPR → FPGA Engine

RAPID Program → RAPID Compiler → Automata → Micron AP Compiler → AP Binary

RAPID Program → RAPID Compiler → Automata → iNFAnt2 → GPU Output

RAPID Program → RAPID Compiler → Automata → Xilinx PAR → FPGA Engine
Code Generation

RAPID Program

- Recursive transformation of RAPID program
- Similar to RegEx $\rightarrow$ NFA transformation
- Adapt strategy from Staged Computation
  - Imperative statements: evaluate at compile time
  - Declarative interaction with input: evaluate at runtime
Experimental Methodology

• Five benchmarks from real-world applications (**expressiveness**)
  • **Success:** Applications can be implemented in RAPID
• Compare LOC in RAPID and hand-crafted baseline automata (**scalability**)
  • **Success:** RAPID size remains constant or grows sublinearly with application instance size
• Measure required hardware resources for AP and FPGA (**performance**)
  • **Success:** overheads within 15% of baseline
RAPID Lines of Code

- ARM
- Brill
- Exact
- Gappy
- MOTOMATA

Lines of Code

- Handcrafted
- RAPID

18. December 2018
K. Angstadt – PhD Proposal
RAPID Hardware Utilization

Automata Processor

FPGA

Handcrafted STEs

RAPID STEs

Handcrafted LUTs

RAPID LUTs

Handcrafted Reg.

RAPID Reg.
• Feasibility of compiling an imperative, high-level programming language to a set of finite automata for execution on hardware accelerators
• RAPID extends C- or Java-like language with domain-specific parallel control structures and code abstraction
• Preliminary results demonstrate low overheads on AP and FPGA; reduced program size
Interactive Debugging for High-Level Languages and Accelerators

Research Effort 2
Houston, we have a problem!

Unexpected output deep in data processing

Bug in corner case infrequently activated by input

Incoming Data

CPU too slow to debug full application, but may be difficult to extract subset of input
Research Effort 2

**Goal:** Design a high-speed interactive debugger for a high-level language (RAPID) executing on a hardware accelerator

**Hypothesis:** The automata abstraction reduces the program state that must be monitored at the signal level on hardware accelerators while still allowing for program semantics to be lifted to higher levels of abstraction and meaningfully supporting debugging
“A debugger is a program designed to help detect, locate, and correct errors in another program. It allows the developer to step through the execution of the process and its threads, monitoring memory, variables, and other elements of process and thread context.”

MSDN Windows Dev Center

Multi-Step Process

1. First, we must halt execution and extract current **program state** from the processor
   
   **Insight:** repurpose existing hardware/signal monitoring

2. Then, we **lift** the extracted state to the semantics of the source language

   **Insight:** generate mapping from expressions/statements to hardware elements during compilation
Where do we stop?

• **Breakpoints** annotate expressions/statements to specify locations to pause execution for inspection
  - Traditional notion relies on instructions stream
  - Mechanism does not apply directly to architectures with no instructions (e.g., FPGAs, AP)

• **Key Insight:** Automata computation driven by input
  - Set breakpoints on input data, not instructions
  - Supports use case of stopping computation at abnormal behavior
  - Can also provide abstraction of traditional breakpoints
Capturing State

- Process input data up to breakpoint
- State of automata is **compact**
  - $O(n)$ in the number of states of the NFA
  - **State vector** captures relevant execution information
- Repurpose existing hardware to capture
  - AP: State vector cache already stores active states
  - FPGA: Integrated Logic Analyzers (ILAs) and Virtual I/O pins (VIOs) allow for probing of activation bits

**Challenge:** Space overhead on FPGAs

ILAs can be dynamically reconfigured to probe different signals, but support additional features, causing bloat.

VIOs are tied to the same clock as the design slow down the design.
Lifting Hardware State to Source-Level

- Modify the RAPID compiler to generate **debugging tables**
  - Approach for the RAPID compiler is similar to traditional compilation
  - For every line, which **NFA states** does it map to?
  - For every line, what variables are in scope and **what are their values** (or which hardware resources hold their values)?
- At breakpoint, apply mappings in reverse
- Now have:
  - Expressions currently executing
  - Values of in-scope variables
Putting it all together

Standard Program Execution

Accelerator processes data → Abnormal behavior observed

Debugging Execution

Accelerator processes data → N → User-defined breakpoint

System-calculated breakpoint

Mapping

Accelerator state vector → Simulator processes data → Simulator state vector
Experimental Methodology

• Measure **performance** and **scalability** of FPGA-based automata debugging
  • ANMLZoo benchmark Suite (14 real-world applications)
  • Measure additional resources used and relative clock frequency
  • **Success:** fit on commercial FPGA and exceed performance of baseline CPU engine

• Measure **ease of use** with a human study
  • Participants given **fault localization** task
  • Ten RAPID programs with indicative bugs
  • Collect implicated lines and measure time taken to answer
  • **Success:** statistically significant improvement in accuracy or time
Preliminary FPGA Results
Human Study Results

• N=61 participants (predominantly UVA students)
• Our debugging tool improves a user’s fault localization accuracy for RAPID programs in a statistically significant manner ($p = 0.013$)
• No statistically significant impact on the time needed to localize faults in RAPID programs
• Debugging information for RAPID programs helps novices and experts alike (there is no interaction between developer experience and the ability to interpret debugging information)
Research Effort 2 Summary

• Design a high-speed interactive debugger for RAPID on hardware accelerators
• Repurpose existing hardware to extract runtime state from accelerator and bridge semantic gap with high-level language
• Preliminary FPGA results demonstrate viability
  • High overhead remain a challenge
• Human Study results demonstrate ease of use
Expressive Power of In-Memory Automata Accelerators

Research Effort 3
Automata/RegEx Processing Platforms

- REAPR (FPGA-Based)
- VASim
- HyperScan
- Becchi, et al.
- PCRE
- DFAGE
- iNFAnt2
- IBM PowerEN
- Cache Automaton
- PAP
- Micron AP
- UAP
- HARE

18. December 2018
K. Angstadt – PhD Proposal
Finite automata are fundamentally limited in the kinds and complexity of analyses they support.
Finite automata are fundamentally limited in the kinds and complexity of analyses they support.
**Goal:** Extend the expressive power of automata-based accelerators to support more common data processing tasks, including the parsing of recursively-nested structures.

**Hypothesis:** An in-cache accelerator architecture supporting pushdown automata computation will support a rich class of applications, and allow for improved performance over state-of-the-art baselines.
Accelerated in-SRAM Pushdown ENgine

Scalable processing engine that uses LLC slices to accelerate Pushdown Automata computation

Custom five-stage datapath using SRAM lookups can process up to one byte per cycle

Optimizing compiler supports existing grammars, packs states efficiently, and reduces the number processing stalls
Pushdown Automata Refresher

Input Symbol Match

Top of Stack Match

Stack Actions

Finite State Control

Stack Memory

STACK
Deterministic Pushdown Automata (DPDA) avoid stack divergence, but still support parsing of most common languages.
Recognizing Palindromes with a Middle Character

01010c01010

18. December 2018

K. Angstadt – PhD Proposal
Recognizing Palindromes with a Middle Character

0
* Pop 0 Push ‘0’

1
* Pop 0 Push ‘1’

\[
\text{0 1 0 1 0 c 0 1 0 1 0}
\]

18. December 2018

K. Angstadt – PhD Proposal
Recognizing Palindromes with a Middle Character

0 * Pop 0 Push ‘0’

1 * Pop 0 Push ‘1’

0 0 Pop 1 No Push

0 0 Pop 1 No Push

ε ⊥ Pop 0 No Push

No Push

No Push

STACK

0

⊥
Recognizing Palindromes with a Middle Character

01010c01010
Recognizing Palindromes with a Middle Character
Recognizing Palindromes with a Middle Character

18. December 2018

K. Angstadt – PhD Proposal
Recognizing Palindromes with a Middle Character

STACK

0
1
0
1
0
⊥

18. December 2018

K. Angstadt – PhD Proposal

55
Recognizing Palindromes with a Middle Character

0 0
Pop 0
Push ‘0’

1 1
Pop 0
Push ‘1’

0
Pop 0
No Push

1
1
Pop 1
No Push

ε
⊥
Pop 0
No Push

STACK

0
1
0
1
0
⊥
Recognizing Palindromes with a Middle Character

STACK

01010c01010
Recognizing Palindromes with a Middle Character

STACK

- ε
- 1
- 1
- 1
- 0
- 0
- 0
- 1
- 1
- 0
- 0
- 1

01010c01010

18. December 2018

K. Angstadt – PhD Proposal
Recognizing Palindromes with a Middle Character

0
  *  
Pop 0  
Push '0'

1
  *  
Pop 0  
Push '1'

0
  c  
Pop 0  
No Push

0
  0  
Pop 1  
No Push

1
  1  
Pop 1  
No Push

ε
  ⊥  
Pop 0  
No Push

STACK

0
1
0
⊥

01010c01010
Recognizing Palindromes with a Middle Character

0
*  
Pop 0
Push ‘0’

1
*  
Pop 0
Push ‘1’

0
Pop 0
No Push

1
Pop 0
No Push

ε
⊥
Pop 0
No Push

STACK

1
0
⊥
Recognizing Palindromes with a Middle Character

- Pop 0, Push '0'
- Pop 0
- Pop 0, Push '1'
- Pop 0
- No Push
- Pop 1
- No Push
- Pop 1
- No Push
- Pop 0
- No Push
- Pop 0
- No Push
- Pop 0, Push 'c'
- Pop 0
- No Push
- Pop 1
- No Push
- Pop 1
- No Push
- Pop 0
- No Push
- Pop 0
- No Push

STACK

01010c01010
Recognizing Palindromes with a Middle Character

01010c01010
Mapping DPDA Efficiently to Hardware

• ASPEN supports **homogeneous** DPDA
  • All transitions to a state occur on the same input character, top of stack comparison, and stack operation
  • Similar in nature to homogeneous NFAs

• Equal expressive power as standard DPDA

• State increase is **quadratic in the worst case** with a fixed alphabet

• Allows for **efficient mapping** to hardware resources
  • Transitions decoupled from input/stack matches
Five Steps of DPDA Execution Per Cycle

0  *  Pop 0  Push ‘0’
1  *  Pop 0  Push ‘1’

0  *  Pop 0  No Push
1  *  Pop 0  No Push

ε  ↓  Pop 0  No Push

STACK

01010c01010

18. December 2018
K. Angstadt – PhD Proposal
Five Steps of DPDA Execution Per Cycle

1. Input Match $\epsilon$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition

Stack:

101010c01010
Five Steps of DPDA Execution Per Cycle

1. Input Match $\epsilon$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition
Five Steps of DPDA Execution Per Cycle

1. Input Match $\epsilon$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition
Five Steps of DPDA Execution Per Cycle

1. Input Match\(\varepsilon\)
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition
Five Steps of DPDA Execution Per Cycle

1. Input Match $\varepsilon$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition

01010c01010c
Where is ASPEN?

- ASPEN uses 2 arrays per bank
- 240 states per bank
- Full connectivity within bank
- Global switch and stack in CBOX for large DPDA
ASPEN Datapath — 240 States in 2 SRAM Arrays

1. Input Match
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition
Optimizations

Epsilon Merging

Goal: Reduce the number of stalls while processing input

Multipop

• Average of 65% reduction in epsilon states
Experimental Methodology

• Compile existing grammars (*expressive power* and *legacy support*)
  • Cool, JSON, XML, Dot, etc.
  • Measure hardware utilization, including optimization improvement

• Real-world application case studies (*performance and scalability*)
  • Stress different aspects of architecture
  • Large DPDA with Global Stack (high connectivity)
  • Small DPDA with Local Stacks (sparse connectivity)
  • Compare runtime performance to state-of-the-art baselines

• **Success**: Outperform state-of-the-art for indicative applications while not exceeding power thresholds for modern CPUs
Benchmarking:

- Parabix, Ximpleware, UW XML

ASPEN is 13-18x faster (on average) than popular CPU Parsers.

Performance did not vary significantly with complexity of XML.

Optimizations and tokenization hide ε-stalls.

**Application 1: XML Parsing**

<table>
<thead>
<tr>
<th>Markup Density</th>
<th>Xerces</th>
<th>Expat</th>
<th>ASPEN</th>
<th>ASPEN-MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low (&lt; 0.3)</td>
<td></td>
<td></td>
<td><img src="image" alt="Graph" /></td>
<td></td>
</tr>
<tr>
<td>Medium (0.3-0.7)</td>
<td></td>
<td></td>
<td><img src="image" alt="Graph" /></td>
<td></td>
</tr>
<tr>
<td>High (&gt;0.7)</td>
<td></td>
<td></td>
<td><img src="image" alt="Graph" /></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td><img src="image" alt="Graph" /></td>
<td></td>
</tr>
</tbody>
</table>
Application 2: Anomaly Intrusion Detection

• Identify abnormal program executions by monitoring memory access patterns
• Hypothesis: normal executions will exhibit consistent, similar memory access patterns
• Cache is perfect!
  • Transparently snoop on memory access
  • Automata accelerators repurpose address lines for data input
• Revisits “a sense of self” work by Forrest et al. using system calls for signatures of normal behavior
MemIDS Basic Approach: Sliding Windows

- Successful approaches (syscalls) use simple automata
- Build dictionary of observed behavior
- Sliding window of sequences to determine normal/abnormal

Training Memory: AABACCCBBBC
MemIDS Basic Approach: Sliding Windows

- Successful approaches (syscalls) use simple automata
- Build dictionary of observed behavior
- Sliding window of sequences to determine normal/abnormal

Training Memory: AABACCCBBC
Testing Memory: AAECDC

Diagram:
- A
- B
- C
- D
- E
- A
- B
- C
- D
- E
MemIDS Research Questions

• Can we successfully detect the execution of different programs?
• How many bits of address are needed for a meaningful dictionary?
• What classes of abnormal behavior can we detect?
  • Speculative execution-based attacks
  • Stealthy malware
  • Information leaks
• What do we do with this information?
  • Block access
  • Delay access
  • Return bogus values
Research Effort 3 Summary

• Expand expressive power of automata accelerators with new in-cache architecture for DPDA
• New homogeneous DPDA variant for efficient hardware implementation
• Optimizing compiler to support existing grammars
• Preliminary results demonstrate improved performance for XML parsing
• Ongoing work to leverage accelerator for intrusion detection applications
Adapting Legacy Code for Execution on Hardware Accelerators

Research Effort 4
Legacy Code in the Age of Hardware Accelerators

• Legacy code typically cannot be directly compiled for accelerators
• Learning a new programming model is costly and slows rate of adoption of new accelerators
• May want to “try out” new hardware with existing software
  • No training on new hardware
  • Limited time or resources to allocate
Research Effort 4

**Goal:** Reduce the burden on developers tasked with porting legacy code to execute on hardware accelerators

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

• Input: function kernel : string -> bool

• Assumptions:
  • Function decides a regular language
  • Source code for function is available

• Output: finite automaton with the same behavior on “all” inputs as kernel
Angluin-Style Learning (L*)

- **Learner**
  - Membership Query: $s \in L$
  - Equivalent Query: $L(A) \equiv L$
  - Yes or Counterexample

- **Teacher**
  - Yes/No Answer

- **Oracle**
  - Yes/No Answer

Diagram:
- Learner connected to Teacher via Membership Query
- Learner connected to Oracle via Equivalent Query
- Oracle connected to Teacher via Yes/No Answer
- Automaton $A$ connected to Learner via Yes or Counterexample
Angluin-Style Learning (L*)

Learner

Teacher

Oracle

Membership Query

Yes/No Answer

Equivalence Query

L(A) \overset{?}{=} L

Automaton

A

L \in \mathcal{L}

s \in L

Yes or Counterexample
Angluin-Style Learning (L*)

Learner

\( A \)

Teacher

Oracle

Membership Query

\( s \in L \)

Yes/No Answer

Equivalence Query

\( L(A) = L \)

Yes or Counterexample

Kernel

Software Model Checker

RE4

18. December 2018 K. Angstadt – PhD Proposal
Equality Checking as Software Verification

- Explores control flow graph looking for property violations
  - Success finding variety of bugs (e.g., double-free, locking violations, etc.)
  - Used in industry for driver verification
- Properties specified as automata
  - **Research**: How do we leverage this?
  - Adapt L*-learned candidate automata to specifications
  - Verify source kernel: violations are counterexamples
- **Research**: identify appropriate verification strategies
Mitigating Risk: Speculative Research

- Restrict kernel functions
  - Decide regular languages
  - Streaming access to input data
  - Time permitting: relax these assumptions

- Allow for approximate solutions
  - Measure accuracy with respect to time
  - Reduce impact of error with examples

- User-provided annotations
  - Guide software verification
  - Note “transitions” in kernel
Experimental Methodology

- Seek benchmark suite of existing kernel functions
- Is it possible for our algorithm to adapt kernels? (legacy support)
- Measure time needed to learn automata and their size in states/hardware resources (scalability)
  - **Success**: Run over the weekend and fit on commercial FPGA
- Measure accuracy of approximate solutions
  - Use provided test suites and augment with test input generation
  - **Success**: use of example inputs improves accuracy
Research Effort 4 Summary

- Design algorithm to adapt legacy kernels for execution on hardware accelerators
- Adopt Angluin-style learning approach
- Convert equivalence queries to software verification tasks
- Speculative research
  - Limit kernels to decide regular languages
  - Annotations to guide software verification
  - Approximate solutions and guiding example inputs
Proposal Overview

• Proposed Research Efforts
  • High-Level Programming Language: RAPID
  • High-Speed, Interactive Debugger for Hardware Accelerators
  • In-Cache Accelerator for Pushdown Automata
  • Adapting Legacy Code for Execution on Hardware Accelerators

• Candidate Schedule
• Conclusion/Discussion
Today
Graduation
Research Period
Publication Lag
Other

Figure 4: Proposed dissertation work schedule.

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

Figure 4 outlines the research timeline for the proposed dissertation. We anticipate completing the proposed research efforts in 1.5 years, with an expected graduation in May 2020. We have completed the research described in Section 3.1 to develop a high-level language for automata processing. The results of this effort have been published in [11, 12]. We have developed a working prototype and conducted an initial human study for our research on interactive debugging (Section 3.2), and a manuscript detailing our findings has been accepted for publication [24]. For the expressive power of in-cache automata processors (Section 3.3), we have completed design of a prototype compiler and architecture, which has been published in [8]. We are currently...
Typical Venues

**Computer Architecture**
- MICRO (March/April)
- HPCA (July/August)
- ASPLOS (August)
- TPDS (Journal)

**PL and Software Engineering**
- ASE (April)
- POPL (July)
- ICSE (August/September)
Publications Supporting Proposed Research


Additional Publications


Invited Papers and Tech Reports


Proposal Summary

- Hardware accelerators more commonplace—need for programming models and maintenance tools
- Four components to improve programming support for hardware accelerators using automata abstractions
  - High-level programming language (RAPID)
  - High-speed, interactive debugging for RAPID on AP and FPGA
  - In-cache accelerator for pushdown automata (ASPEN)
  - Adapt legacy kernels for execution on hardware accelerators
- Evaluation w.r.t. Performance & scalability, ease of use, expressive power, and legacy support
Discussion Cache

• Space overheads for FPGA debugging probes
  • Dynamic probing or decoupling clock signals or some other approach?

• Memory-based intrusion detection evaluations and applications
  • What classes of abnormal behavior can we detect?
  • What do we do with this information?

• Automata synthesis approach and evaluation
  • Thoughts on representing equivalence queries?
  • What will computer architects want to know?

• Potential interest overlap and collaboration?

• Integrating with pedagogy...undergraduate mentorship?
Programming Support

- High-Level Language
- Low-Level (Assembly) Lang.
- Operating System
- Hardware Abstraction Layer
- Micro Architecture
- Logic
- Transistors
- Geometry

New programming languages better-suited for specific application domains/hardware
Software systems to adapt/transform/improve existing (legacy) code
Low-level representations to bridge the gap between software and hardware
Architectural modifications to support applications and features
The AP at a High Level

Row Address
(Input Symbol)

Automata Processor

Routing Matrix

Row Access results in 49,152 match & route operations
Executing NFA in DRAM

- Columns in DRAM store STE labels (Each STE is a single column)
- Reconfigurable routing matrix connects the STEs

Columns with “1”: STEs that accept input symbol

Active States

Active States for Next Clock Cycle

Input: Drives a Row
Program Structure

• Macro
  • Basic unit of computation
  • Sequential control flow
  • Boolean expressions as statements for terminating threads of computation

• Network
  • High-level pattern matching
  • Parallel control flow
  • Parameters to set run-time values
Either/Orelse Statements

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

• Perform parallel exploration of input data
• Static number of parallel operations
Some Statements

- Parallel exploration may depend on candidate patterns
- Iterates over items, dynamically spawn computation

```java
network (String[] comparisons) {
    some(String s : comparisons)
        hamming_distance(s, 5);
}
```
Whenever Statements

```java
whenever( ALL_INPUT == input() ) {
    foreach(char c : "rapid")
        c == input();
    report;
}
```

- Body triggered whenever guard becomes true
- `ALL_INPUT`: any symbol in the input stream
## Parallel Control Structures

<table>
<thead>
<tr>
<th>Sequential Structure</th>
<th>Parallel Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>if...else</td>
<td>either...orelse</td>
</tr>
<tr>
<td>foreach</td>
<td>some</td>
</tr>
<tr>
<td>while</td>
<td>whenever</td>
</tr>
</tbody>
</table>
Example RAPID Program

Association Rule Mining
Identify items from a database that frequently occur together
Example RAPID Program

- **If all symbols in item set match, increment counter**
  ```java
  macro frequent (String set, Counter cnt) {
    foreach(char c : set) {
      while(input() != c);
    }
    cnt.count();
  }
  ```

- **Spawn parallel computation for each item set**
  ```java
  network (String[] set) {
    some(String s : set) {
      Counter cnt;
      whenever(START_OF_INPUT == input())
        frequent(s,cnt);
      if (cnt > 128)
        report;
    }
  }
  ```

- **Sliding window search calls frequent on every input**

- **Trigger report if threshold reached**

Houston, we have a problem!

- Accelerator applications often target large datasets
- CPUs typically too slow to debug full applications
- Abnormal behavior may not manifest itself in testing inputs
  - Low quality/coverage test suites
  - Ex. ANMLZoo (automata processing benchmark suite) contains two inputs per application and no gold standard output!
- May be difficult to extract subset of input for debugging
- Low-level debugging support exists—tedious to use and abstraction mismatch
Traditional Breakpoints

RAPID Program

```cpp
macro helloWorld() {
  whenever( ALL_INPUT == input() ) {
    foreach(char c : "Hello") {
      c == input();
    }
  
    input() == ' '; 
    foreach(char c : "world") { 
      c == input();
    } 
    report;
  }
}
network() {
  helloWorld(); 
}
```

Network

Accelerator processes data with Machine A

Reports occur when line is executed

Input breakpoints inserted at reports
Human Study Results

- ~60 participants (students and professional developers)
- Participants shown 10 RAPID programs with seeded defects
  - 5 have interactive debugger
  - 5 have no debugging information
- Asked to identify location of bug in source code and describe
- Recorded time needed to perform each task
Implementing ASPEN in LLC

- ASPEN repurposes LLC slices for pushdown automata computation
- Location in LLC supports tighter coupling with CPU operations than dedicated accelerator
  - PDA often part of a larger workflow
  - ASPEN similar to auxiliary functional unit in CPU (similar to FPU or vector unit)
- SRAM arrays in LLC already support necessary operations for DPDA execution
Stack Match in SRAM

- Check **all states** against top of stack
  - One column of SRAM/state
  - Input TOS as row address
  - “1”: match; ”0”: no match
- **Intersect** with currently active states
Stack Match in SRAM

• Check all states against top of stack
  • One column of SRAM/state
  • Input TOS as row address
  • “1”: match; ”0”: no match

• Intersect with currently active states
Preliminary Results: XML Parsing

<table>
<thead>
<tr>
<th>Component</th>
<th>Max Frequency</th>
<th>Operating Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPEN</td>
<td>880 MHz</td>
<td>850 MHz</td>
</tr>
<tr>
<td>Cache Automaton</td>
<td>4 GHz</td>
<td>3.4 GHz</td>
</tr>
</tbody>
</table>

• **Baseline Evaluation**
  • **CPU:** 2.6 GHz dual-socket Intel Xeon E5-2697-v3 (28 cores total)
  • **Performance and Power:** PAPI, Intel RAPL
  • **ASPEN Simulation:**
    • METIS graph partitioning framework
    • VASim modified for cycle-accurate DPDA simulation
Angluin-Style Learning

- Attempts to learn a finite automaton from a held-out model
- Requires set membership queries (Is string in language? Returns yes/no answer)
  - Run legacy code for answer
- Requires equivalence queries (Is the automaton equivalent to model? Returns yes/counterexample)
  - Software model checking to find differences