Debugging Support for Pattern-Matching Languages and Accelerators

Matthew Casias†, Kevin Angstadt*, Tommy Tracy II†, Kevin Skadron†, Westley Weimer*

†Department of Computer Science, University of Virginia
*Computer Science and Engineering, University of Michigan

This work was funded in part by: NSF grants CCF-1629450, CCF-1619123, and CNS-1619098; the Jefferson Scholars Foundation; the Virginia-North Carolina Louis Stokes Alliance for Minority Participation; and support from CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.
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

Physical Limits Spark Creativity

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

How Accelerators Help

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018
How Accelerators Help

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018
How Accelerators Help

All you need to know about Automata, Micron's revolutionary processor

By Desire Athow  March 28, 2014  World of tech
A CPU in the form of a memory module

Cost per unit

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018
How Accelerators Help

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018
How Accelerators Help

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018
Developers only spend a fraction of their time writing new code
  - Remaining time is spend understanding and modifying code
  - Using a debugger to understand code and localize faults is common

Hardware developers spend even more time debugging and analyzing designs

With the recent focus on HLS (e.g., OpenCL, SDAccel, etc.), we want debuggers that support these languages
  - Simulation/Emulation on CPU is likely too slow—that’s why we’re using an accelerator in the first place!
A Tale of Two Debugging Worlds

What Software Developers Expect

What Hardware Developers Use
Support interactive debugging of high-level languages on hardware accelerators

OVERVIEW

• RAPID Programming Model
• Debugging System
• Technical Approach
• Experimental Evaluation
• Open Challenges
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
• Compilation strategy supports execution on AP, FPGAs, CPUs, and GPUs
  • Finite automata (NFAs)/State machines used as an intermediate representation of computation
Domain-Specific Code Abstraction

Key

\[ \begin{array}{c}
\text{Active Searches (Automata)} \\
\end{array} \]

\[ \begin{array}{c}
\text{Target Pattern} \\
\end{array} \]

\[ \begin{array}{c}
\text{…} \\
\end{array} \]

\[ \begin{array}{cccccccc}
G & C & T & G & A & C & C & A \\
\end{array} \]

\[ \begin{array}{cccccccc}
T & A & C & G & G & \ldots \\
\end{array} \]

Incoming Data

Matching patterns trigger reports

\[ \begin{array}{c}
\text{CGGCAT} \\
\end{array} \]

\[ \begin{array}{c}
\text{ATCGA} \\
\end{array} \]
RAPID (Automata) 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
Houston, we have a problem!

Incoming Data

ATCGA

CGGCAT
Houston, we have a problem!

Unexpected output deep in data processing

Incoming Data

ATCGA

CGGCAT
Houston, we have a problem!

- Unexpected output deep in data processing
- Bug in corner case infrequently activated by input

Incoming Data

ATCGA

CGGCAT
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
“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:** leverage 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
- Cache state vectors to decrease latency
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
Putting it all together

Standard Program Execution

Accelerator processes data

Abnormal behavior observed
Putting it all together

Standard Program Execution

Accelerator processes data  \(\rightarrow\)  Abnormal behavior observed

Debugging Execution

Accelerator processes data  \(\rightarrow\)  User-defined breakpoint
Putting it all together

Standard Program Execution

Accelerator processes data

Abnormal behavior observed

Debugging Execution

Accelerator processes data

User-defined breakpoint

System-calculated breakpoint

Accelerator state vector
Putting it all together

Standard Program Execution

Accelerator processes data → Abnormal behavior observed

Debugging Execution

Accelerator processes data

Accelerator state vector → Simulator caches data → Simulator state vector

User-defined breakpoint

System-calculated breakpoint
Putting it all together

Standard Program Execution

Accelerator processes data

Abnormal behavior observed

Debugging Execution

Accelerator processes data

User-defined breakpoint

System-calculated breakpoint

Accelerator state vector

Simulator caches data

Simulator state vector

Mapping

Interactive Debugger
Evaluation: Hardware and Software

• Measure **performance** and **scalability** of FPGA-based automata debugging
  • ANMLZoo benchmark Suite (14 real-world automata applications)
  • Focus on **overheads** (e.g., hardware resources and clock frequency) introduced by adding debugging hardware

• Measure **ease of use** with a human study
  • Participants given **fault localization** tasks in mock development environment
  • Ten RAPID programs with indicative bugs
  • Focus on **accuracy**, **time**, and **developer experience**
Experimental Methodology

• **Evaluation System**
  - **CPU:** 3.70 GHz 4-core Intel Core i7-4820K (32 GB RAM)
  - **FPGA:** Alphadata board rev 1.0 (Xilinx Kintex-Ultrascale xcku060-ffva1156-2-e)

• **Synthesis:**
  - REAPR modified to produce Verilog with VIO for debugging
  - Xilinx Vivado 2017.2

• **Debugging Simulation:**
  - RAPID compiler modified to generate debugging information
  - VASim modified to produce and capture state vector information
FPGA Hardware Overheads

Overheads are manageable, but vary widely

State Vector Size (KB) vs. Hardware Overhead (x)

FPGA Hardware Overheads

LUT Overhead

FF Overhead

State Vector Size

Brill
ClamAV
Dotstar
ER
Fermi
Hamming
Levenshtein
PowerEN
Protonata
RF
Short
SPM
BlockRings
GeoMean

Overheads are manageable, but vary widely.
FPGA Performance Overheads

82% clock frequency on average — still fast on debug builds
Human Study Results

• N=61 participants (predominantly UVA students)

• Asked to identify location of bug in source code and describe

• Statistically significant improvement in fault localization accuracy by 22% (p = 0.013)

• Helps novices and experts alike
Open Challenges

Hardware Design

• **Reduce overhead** of program state capture
• **Decouple clocking** for program execution
• **Dynamic selection** of program state to monitor and capture

Software Tools

• Program analyses to **limit captured state size**
• Further study of **acceptable overheads and latencies**
• Additional support for **existing** (e.g., watchpoints) and new debugging abstractions
Conclusion

82% original clock freq.

Low overheads for server-class FPGAs

22% increase in fault localization accuracy

RAPID Programming Model as Stepping Stone

Accelerator-Assisted Debugging

SW

HW

Debugger
Bonus Slides
Example RAPID Program

- **If all symbols in item set match, increment counter**

- **Spawn parallel computation for each item set**

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

- **Trigger report if threshold reached**

```
macro frequent (String set, Counter cnt) {
  foreach(char c : set) {
    while(input() != c);
  }
  cnt.count();
}

network (String[] set) {
  some(String s : set) {
    Counter cnt;
    whenever(START_OF_INPUT == input())
      frequent(s,cnt);
    if (cnt > 128)
      report;
  }
}
```
Traditional Breakpoints

**RAPID Program**

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

**Figure 5:** Transformation of a line breakpoint to an input breakpoint. Reports generated by STEs mapped to RAPID expressions determine input breakpoints.

**Table 1: ANMLZoo Benchmark Overview**

<table>
<thead>
<tr>
<th>Benchmark Family</th>
<th>States Ave</th>
<th>Node Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brill Regex</td>
<td>42,658</td>
<td>1.03287</td>
</tr>
<tr>
<td>ClamA V Regex</td>
<td>49,538</td>
<td>1.00396</td>
</tr>
<tr>
<td>Dotstar Regex</td>
<td>96,438</td>
<td>0.97396</td>
</tr>
<tr>
<td>PowerEN Regex</td>
<td>40,513</td>
<td>0.97601</td>
</tr>
<tr>
<td>Protamata Regex</td>
<td>42,009</td>
<td>0.99110</td>
</tr>
<tr>
<td>Snort Regex</td>
<td>69,029</td>
<td>1.08831</td>
</tr>
<tr>
<td>Hamming Mesh</td>
<td>11,346</td>
<td>1.69672</td>
</tr>
<tr>
<td>Levenshtein Mesh</td>
<td>2,784</td>
<td>3.26724</td>
</tr>
<tr>
<td>Entity Resolution (ER) Widget</td>
<td>95,136</td>
<td>2.28372</td>
</tr>
<tr>
<td>Fermi Widget</td>
<td>40,783</td>
<td>1.41176</td>
</tr>
<tr>
<td>Random Forest (RF) Widget</td>
<td>33,220</td>
<td>1.00000</td>
</tr>
<tr>
<td>SPM Widget</td>
<td>100,500</td>
<td>1.70000</td>
</tr>
<tr>
<td>BlockRings Synthetic</td>
<td>44,352</td>
<td>1.00000</td>
</tr>
<tr>
<td>CoreRings Synthetic</td>
<td>48,002</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

For each benchmark, we generate an FPGA configuration using our modified version of REAPR [55], producing Verilog including both VIOs (for capturing state vector) and also Wadden et al.'s reporting architecture [47] 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. 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. FPGA Results

Performance results for FPGA-based debugging are presented in Table 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.

Three 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 our added 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 2 include this overhead.

Our additional debugging hardware has average LUT and FF overheads of 707 × 707 and 5.925 × 733, 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 specific structures are not provided), but since it is provided for context switching, using it for debugging incurs no extra hardware cost. For FPGAs, the area overhead of 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 (as in Entity Resolution and SPM), which is straightforward and supported by our infrastructure. However, partitioning requires either running multiple passes over the input (end-to-end latency increases as extra passes are added) or using multiple FPGAs (increasing hardware costs, but as of August 2018 cloud computing providers offer instances with access to up to eight...
### Summary Statistics

<table>
<thead>
<tr>
<th>Subset</th>
<th>Average Time (min.)</th>
<th>Average Accuracy</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>8.17</td>
<td>50.3%</td>
<td>61</td>
</tr>
<tr>
<td>Intermediate Undergraduate Students</td>
<td>7.30</td>
<td>49.2%</td>
<td>37</td>
</tr>
<tr>
<td>Advanced Undergraduate Students</td>
<td>10.14</td>
<td>50.0%</td>
<td>21</td>
</tr>
<tr>
<td>Graduate Students and Professional Developers</td>
<td>5.07</td>
<td>66.7%</td>
<td>3</td>
</tr>
</tbody>
</table>