Debugging Support for Pattern-Matching Languages and Accelerators

Matthew Casias
University of Virginia
Department of Computer Science
Charlottesville, VA, USA
mkc5dm@virginia.edu

Kevin Angstadt
University of Michigan
Computer Science and Engineering
Ann Arbor, MI, USA
angstadt@umich.edu

Tommy Tracy II
University of Virginia
Department of Computer Science
Charlottesville, VA, USA
tjt7a@virginia.edu

Kevin Skadron
University of Virginia
Department of Computer Science
Charlottesville, VA, USA
skadron@virginia.edu

Westley Weimer
University of Michigan
Computer Science and Engineering
Ann Arbor, MI, USA
weimerw@umich.edu

Abstract

Programs written for hardware accelerators can often be difficult to debug. Without adequate tool support, program maintenance tasks such as fault localization and debugging can be particularly challenging. In this work, we focus on supporting hardware that is specialized for finite automata processing, a computational paradigm that has accelerated pattern-matching applications across a diverse set of problem domains. While commodity hardware enables high-throughput data analysis, direct interactive debugging (e.g., single-stepping) is not currently supported.

We propose a debugging approach for existing commodity hardware that supports step-through debugging and variable inspection of user-written automata processing programs. We focus on programs written in RAPID, a domain-specific language for pattern-matching applications. We develop a prototype of our approach for both Xilinx FPGAs and Micron's Automata Processor that supports simultaneous high-speed processing of data and interactive debugging without requiring modifications to the underlying hardware. Our empirical evaluation demonstrates low clock overheads for our approach across thirteen applications in the ANMLZoo automata processing benchmark suite on FPGAs. Additionally, we evaluate our technique through a human study involving over 60 participants and 20 buggy segments of code. Our generated debugging information increases fault localization accuracy by 22%, or 10 percentage points, in a statistically significant manner ($p = 0.013$).

CCS Concepts • **Software and its engineering** → **Domain specific languages**; *Software testing and debugging*; • **Computer systems organization** → *Reconfigurable computing*.

Keywords automata processing, debugging, human study

ACM Reference Format:
Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. 2019. Debugging Support for Pattern-Matching Languages and Accelerators. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).* ACM, New York, NY, USA, 14 pages. https://doi.org/https://doi.org/10.1145/3297858.3304066

1 Introduction

The amount of data being produced by companies and consumers continues to grow,[1] and business leaders are becoming increasingly interested in analyzing and using this collected information.[2] To keep up with data processing needs, companies and researchers are turning to specialized hardware for increased performance. Accelerators, such as GPUs, FPGAs, and Micron's D480 Automata Processor (AP) [10], trade off general computing capabilities for increased performance on very specific workloads; however, these devices require additional architectural knowledge to effectively program and configure. Despite this added complexity, researchers have successfully used specialized hardware to accelerate data analysis across many domains, including: natural language processing [66], network security [33], graph analytics [32], high-energy physics [53], bioinformatics [30, 31, 45], pseudo-random number generation and simulation [48], data-mining [51, 52], and machine learning [44].

[1] https://web.archive.org/web/20170203000215/http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode

[2] https://www.dnvgl.com/assurance/viewpoint/viewpoint-surveys/big-data.html

ASPLOS'19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/https://doi.org/10.1145/3297858.3304066

Numerous programming models have been introduced to ease the burden on hardware accelerator users, such as OpenCL [41], Stanford's Legion programming system [6], and Xilinx's SDAccel framework.[3] Recently, the RAPID language was proposed to improve the programming of automata processing engines [3]. These processors accelerate the identification of a collection of byte sequences (or patterns) in a stream of data by supporting many comparisons in parallel. RAPID is a C-like language that includes a combined imperative and declarative model for pattern-matching problems, providing intuitive representations for patterns in use cases where regular expressions become cumbersome or exhaustive enumerations. The language provides parallel control structures, admitting concurrent searches for multiple criteria against the data stream. RAPID programs are compiled into finite automata, supporting efficient execution using both automata processing engines, such as Micron's Automata Processor (AP) or Subramaniyan et al.'s Cache Automaton [42], and also general-purpose accelerators, such as Field-Programmable Gate Arrays (FPGAs) and GPUs. However, RAPID abstracts away from the low-level automata or circuit paradigms used by the hardware, thus allowing developers to work with code in a semantically-familiar form.

This focus on new domain-specific languages (DSLs) and accelerators introduces 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 [63, 65]. 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.

Although debugging support for CPUs is mature and fully-featured (including standard tools [40], successful technology transfer [5] and annual conferences [17]), throughput of automata processing applications on CPUs is typically orders of magnitude slower than on hardware accelerators [26, 49], making CPUs too slow for effective debugging of automata processing. Unfortunately, current debugging techniques are limited or nonexistent for most accelerators. For architectures where the sequence of operations is configured or hard-wired in the hardware (i.e., there is no instruction stream, such as in FPGAs), the traditional method of inserting breakpoints is not available. Instead, debugging on FPGAs is often performed at the signal level using logic analyzers or scan chains [4, 21, 43, 55], exposing low-level state to software. The AP also provides no explicit debugging support, but does expose low-level state through APIs.

We propose 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 combined 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 our debugging approach on

[3] https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

the ANMLZoo benchmarks [49] using the REAPR automata-to-FPGA tool [57] and a server-class FPGA. We were able to achieve an average of 81.70% of the baseline clock frequencies. We also discuss the tradeoff between resource overheads and support for debugging.

We evaluate 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.

In summary, this paper 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.

2 Background

In this section, we present background material on our debugging technique and the underlying execution model.

2.1 Homogeneous Finite Automata

In this work, we develop a debugging technique to support algorithms designed for a finite automata computational model. In particular, we consider algorithms represented as homogeneous non-deterministic finite automata (NFAs). An NFA is a state machine that consumes an input string and returns a Boolean value indicating whether the given string matches the pattern encoded in the machine. *Homogeneous* NFAs restrict the definition of the transition function such that, for all states q_1 and q_2, $\delta(q_1, \alpha) = \delta(q_2, \beta) \Rightarrow \alpha = \beta$ [8].

Traditionally, NFAs are represented as a directed graph with vertices representing states and labeled edges representing the transition function. Homogeneity restricts all incoming edges to a given state to be labeled with the same set of symbols. This allows us to label states rather than transition edges, a representation that is often helpful for hardware acceleration, but does not reduce expressiveness. An example homogeneous NFA is given in Figure 1.

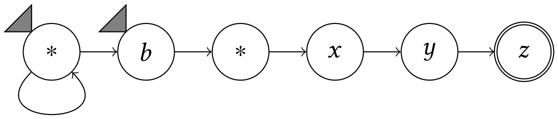


Figure 1. Homogeneous NFA matching a 'b' two characters before the string "xyz" ('*' is a wildcard). Gray triangles are starting states; double circles are accepting states.

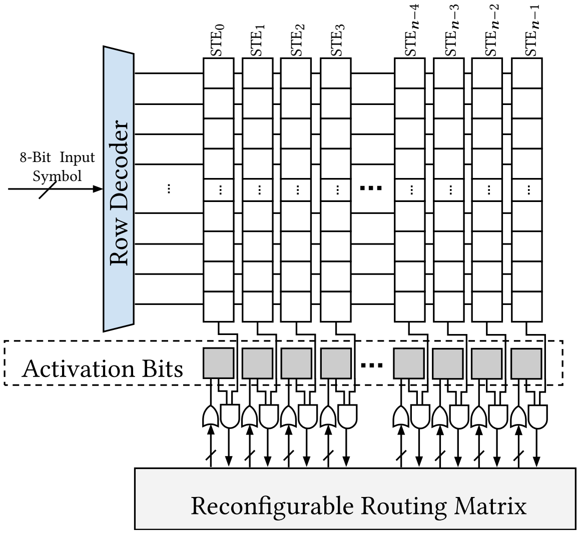


Figure 2. AP architecture. STEs are stored in a memory array, and edges are encoded in a reconfigurable routing matrix.

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 [28, 37]. For example, the use of GPUs and Field-Programmable Gate Arrays (FPGAs) to accelerate general-purpose computation has become commonplace [34]. In fact, cloud computing providers now lease compute time on nodes containing both GPUs and FPGAs.[4] In this work, we focus on two architectures that accelerate *automata processing* applications: commodity FPGAs and Micron's D480 AP. While we focus on the AP and FPGAs, automata processing engines have been developed for other hardware platforms, including CPUs and GPUs [2, 18]. In

[4] https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/

Section 4.3, we describe the functionality needed for such processors to be used with our debugging system.

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. [10]. 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 reconfigurable routing matrix, representing the STEs and edges respectively. The architecture is depicted in Figure 2.

A single column of the memory array is used to represent an STE. For the NFA given in Figure 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 all STEs simultaneously determine whether they 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. 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 this work, we leverage this mapping information to lift hardware runtime state to the semantics of the user-level program.

Field-Programmable Gate Arrays. FPGAs are fabrics 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 fabric. This architectural model allows FPGAs to form arbitrary circuits, which can be useful for prototyping logic circuits.

Recently Xie et al. developed the Reconfigurable Engine for Automata Processing (REAPR), which generates high-performance automata processing and IO cores for Xilinx FPGAs [57]. REAPR generates FPGA configurations that operate very similarly to the AP-style processing model. 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.

Further, REAPR supports notions of reporting and STE labeling, thereby admitting similar operations to lift runtime state to the semantics of the user-level program.

3 The RAPID Programming Language

Before discussing our approach to generating debugging information, we summarize the basics of the subject language. RAPID is a high-level language designed to support highly-parallel pattern searches through a stream of data [3]. RAPID is intended for use cases where hundreds or thousands of pattern searches are executed simultaneously over a large data stream. A pattern defines a sequence of data that may be found within another collection of data.

3.1 Supporting Parallel Pattern Searches

RAPID provides built-in support for parallel inspection of input data. Special constructs allow programmers to perform tasks, such as checking for multiple criteria simultaneously or matching a sequence at any point in the input stream.

Either/Orelse Statement. This structure supports parallel exploration of patterns. An `either/orelse` statement instantiates a static number of parallel comparisons against the input stream with each block in the statement being executed in parallel.

Some Statement. In certain cases the ability to generate a dynamic number of parallel searches (e.g., one for each element of an array) is desirable. RAPID's `some` statement, using syntax that is similar to a for-in loop in Java, instantiates a parallel computation for each element provided.

Whenever Statement. Sliding window searches, in which a pattern could begin on any character within the input stream, are a common operation in pattern-matching. For example, a search might begin after a particular data sequence. The `whenever` statement consists of a Boolean guard and an internal statement. At any point in the data stream 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.

3.2 Worked Example

In Figure 3, we present an example RAPID application that searches for 'b' two characters before the string "xyz" (the pattern matched by the machine in Figure 1). The `network` (akin to the `main` function in a standard C program) is used to instantiate a single search on line 20. In RAPID, `macros` programmatically define portions of a pattern-matching algorithm. In the `macro` *b_xyz*, a `whenever` loop creates a sliding

```
1  macro b_xyz() {
2    // match 'b' two characters before "xyz"
3    whenever( 'b' == input() ) {
4      // match any character
5      // computation stops if a comparison
6      // returns false
7      ALL_INPUT == input();
8
9      // match the string "xyz"
10     foreach(char c : "xyz") {
11       c == input();
12     }
13
14     report; // if we successfully matched everything,
15             // report
16   }
17 }
18
19 network() {
20   b_xyz(); // instantiate a single search with b_xyz
21            // macro
22 }
```

Figure 3. RAPID program matching 'b' two characters before the string "xyz".

window search over the input, searching for a 'b' (the condition is true for every 'b'). Then, we match any character (line 7). Note that comparisons against the input stream are declarative in nature; comparisons evaluating to false terminate computation for the particular parallel search. Finally, a `foreach` loop is used to match "xyz". If all comparisons match successfully, a report event is generated (line 14).

3.3 Compilation

RAPID programs are compiled into a set of finite automata, which can then be executed on an automata processing engine. The compiler employs a staged computation model to perform the conversion: comparisons with the input stream occur at runtime, while all other values are resolved at compile time. Compilation recursively transforms a RAPID program into finite automata in much the same way that regular expressions can be transformed into NFAs [38]. Comparisons with the input stream are transformed into NFA states. The context in which the comparison occurs determines how the NFA states attach to the rest of the automaton. Broadly, each instantiated pattern search in the network generates a stand-alone automaton that is executed in parallel on the processing core. Depending on the targeted automata processing engine being used, further transforms may occur. With the Automata Processor, for example, generated automata are placed and routed to assign states to hardware elements [10]. Similarly, the REAPR toolchain for FPGAs maps generated automata to connected LUTs and FFs, which are then synthesized into a hardware configuration bitmap [57].

4 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.

4.1 Breakpoints

Breakpoints allow a developer to begin interacting with a debugger [19]. 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 [24]. *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 [36], 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 Figure 3, a line breakpoint could be set on line 11 to halt execution for each match of a character in the sequence "xyz".

Input Breakpoints. Automata-based pattern-recognition programs often process large quantities of data, and spurious or incorrect reports[5] 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.

4.2 Hardware Abstractions for Debugging

Unlike traditional (non-parallel) CPU debugging, we explicitly target a setting with a particular kind of parallelism, one

[5] False negatives (missing reports) remain an open challenge.

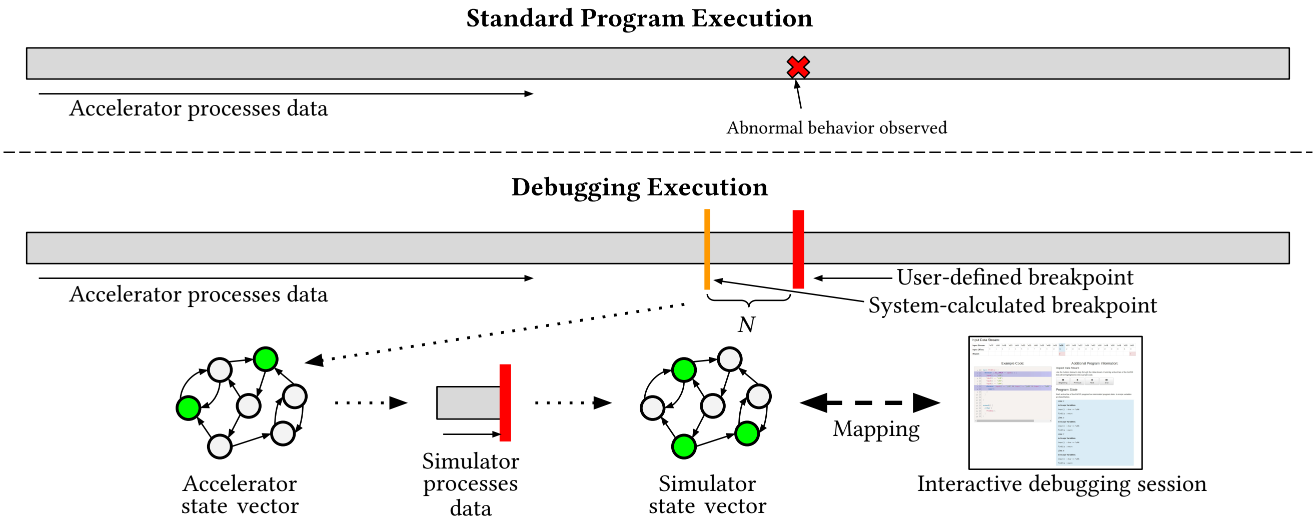

Standard Program Execution
Accelerator processes data
Abnormal behavior observed
Debugging Execution
Accelerator processes data
User-defined breakpoint
System-calculated breakpoint
N
Accelerator state vector
Simulator processes data
Simulator state vector
Mapping
Interactive debugging session

Figure 4. 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.

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 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).

4.3 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 [10]. 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) [58] and virtual IOs (VIOs) [59]. Both of these Xilinx IP (Intellectual Property) blocks are used for runtime debugging the FPGA and come with different design tradeoffs [60].

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 has 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 (cf. Section 5.2) demonstrates that these reductions are less than 20% for most automata applications.

We extend Xie et al.'s REAPR (cf. Section 2.2) 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 (cf. Table 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 [50]) requires iterating through all states in the automaton to capture the active set. Other custom accelerators for automata processing, such as the Cache Automaton [42], also provide direct support for accessing the state vector.

4.4 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 4. 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 B. Updating or selecting new line breakpoints requires regenerating machine B. This transformation is illustrated in Figure 5.

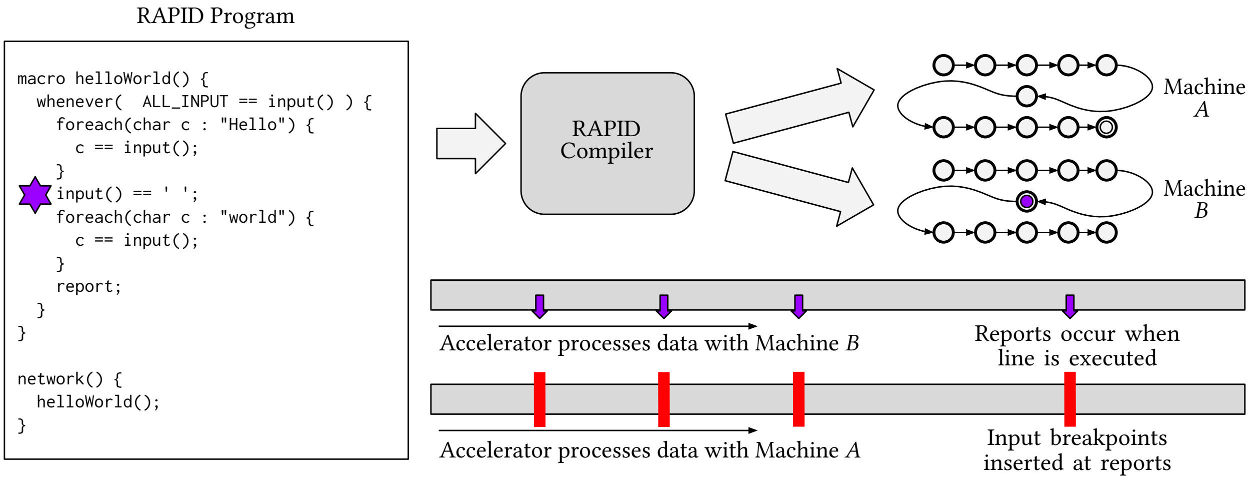
While the double compilation and execution steps do incur a minimum of a 2× overhead[6] for line breakpoints over execution containing no line breakpoints, we note that *current hardware supports this approach.* A more efficient approach would be to support hardware-based debugging signals. On a straightforward 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 [36]. 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.

4.5 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 [36].

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 3.3); 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.

[6] 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.


RAPID Program
RAPID Compiler
Machine A
Machine B
Accelerator processes data with Machine B
Reports occur when line is executed
Accelerator processes data with Machine A
Input breakpoints inserted at reports

```
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.

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 [15]). 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.

4.6 Time-Travel Debugging

Many debuggers provide the ability to step backward in a program, a functionality often referred to as time-travel debugging [20]. 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 [50], 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.[7] 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 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

[7] Modified version available at https://github.com/kevinaangstadt/VASim/tree/statevec.

Table 1. ANMLZoo Benchmark Overview

Benchmark	Family	States	Ave 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

our evaluation on the space and time overheads incurred for the additional FPGA hardware needed in our system.

5.1 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 [49]. The benchmarks are varied, including both regular expression-based and hand-crafted automata. We present a summary of the applications in Table 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 [57], producing Verilog including both VIOs (for capturing state) 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. 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 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.

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 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 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[8] for $13.20 an hour[9]). 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 6) and reduce maintenance time (e.g., [27, 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 (cf. Section 4.3) 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 [49]. Therefore, we expect our FPGA-accelerated system to provide better performance than a CPU-only approach.

[8] https://aws.amazon.com/ec2/instance-types/f1/

[9] https://aws.amazon.com/ec2/pricing/on-demand/

Table 2. FPGA-Based Debugging System Performance Results

Benchmark	Without Debugging				With Debugging				Num. VIOs	LUT Over-head	FF Over-head	Percent Orig. Freq.	Power Over-head
	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.

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.

6 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.

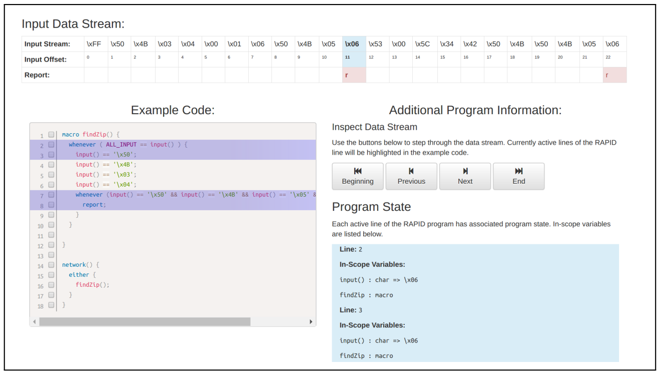
6.1 Experimental Methodology

Our IRB-approved human study[10] 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 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 [49]. 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 [66], frequent subset mining [52], and string alignment for DNA/Protein sequencing [7, 31]. We seeded a variety of defects into the code for our fault localization tasks, based on RAPID developer mistakes reported by Angstadt et al. [3]. 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 6 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,

[10] UVA IRB for Social and Behavioral Sciences #2016-0358-00.



Input Data Stream:

Input Stream:	\xFF	\x50	\x4B	\x03	\x04	\x00	\x01	\x06	\x50	\x4B	\x05	**\x06**	\x53	\x00	\x5C	\x34	\x42	\x50	\x4B	\x50	\x4B	\x05	\x06
Input Offset:	0	1	2	3	4	5	6	7	8	9	10	**11**	12	13	14	15	16	17	18	19	20	21	22
Report:												r											r

Example Code:

```
1  macro findZip() {
2    whenever ( ALL_INPUT == input() ) {
3      input() == '\x50';
4      input() == '\x4B';
5      input() == '\x03';
6      input() == '\x04';
7      whenever (input() == '\x50' && input() == '\x4B' && input() == '\x05' &
8        report;
9      }
10   }
11
12 }
13
14 network() {
15   either {
16     findZip();
17   }
18 }
```

Additional Program Information:

Inspect Data Stream

Use the buttons below to step through the data stream. Currently active lines of the RAPID line will be highlighted in the example code.

Program State

Each active line of the RAPID program has associated program state. In-scope variables are listed below.

Line: 2

In-Scope Variables:

`input() : char => \x06`

`findZip : macro`

Line: 3

In-Scope Variables:

`input() : char => \x06`

`findZip : macro`

Figure 6. A question from the human study including generated debugging information. Task text and program state information are elided for space.

Table 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

Theory of Computation, and Programming 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 3.

6.2 Statistical Analysis

Next, we present statistical analyses of the responses to our human study. We address the following research questions:

1. Does our technique improve fault localization accuracy?
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 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 [27, 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 [61]. 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 [27]) 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 [11]), 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 [56]. 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.

6.3 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 [49].

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.

7 Related Work

The development of debugging tools has a lengthy history [14, 23, 35, 62], and software debuggers are commonplace in development toolchains [24]. Ungar et al. argue that immediacy is important for debugging tasks and developed a step-through debugger [46]. There has also been significant effort devoted to improving the efficiency of debugging tools, such as quickly transferring control when a breakpoint is reached [19] and efficiently supporting large numbers of watchpoints [64]. 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 [39] and energy-harvesting systems [9]. 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 [16]. Similarly, there are approaches for debugging FPGA applications [1, 13]; 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 [12, 25]. 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.

Human studies 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 [54]. Ko and Myers demonstrated that their debugging tool, Whyline, allowed study participants to perform debugging tasks more quickly [22]. Fry and Weimer found that localization accuracy is not uniform across various bug types [11]. Romero et al. found that debugging performance is related to balanced use of available information in programming systems that provide multiple representations of state [29]. Our results complement these findings by demonstrating our debugging system improves fault localization accuracy for the domain of pattern-matching automata processing applications.

8 Conclusions

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.

Acknowledgments

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.

References

[1] H. Angepat, G. Eads, C. Craik, and D. Chiou. 2010. NIFD: Non-intrusive FPGA Debugger – Debugging FPGA 'Threads' for Rapid HW/SW Systems Prototyping. In *International Conference on Field Programmable Logic and Applications*. 356–359. https://doi.org/10.1109/FPL.2010.77

[2] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron. 2018. MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 84–87. https://doi.org/10.1109/LCA.2017.2780105

[3] Kevin Angstadt, Westley Weimer, and Kevin Skadron. 2016. RAPID Programming of Pattern-Recognition Processors. In *Architectural Support for Programming Languages and Operating Systems*. 593–605. https://doi.org/10.1145/2872362.2872393

[4] Z. K. Baker and J. S. Monson. 2009. In-situ FPGA Debug Driven by On-Board Microcontroller. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. 219–222. https://doi.org/10.1109/FCCM.2009.9

[5] Tom Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. 2004. *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft*. Technical Report. 22 pages. https://www.microsoft.com/en-us/research/publication/slam-and-static-driver-verifier-technology-transfer-of-formal-methods-inside-microsoft/

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *International Conference on High Performance Computing, Networking, Storage and Analysis*. Article 66, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389086

[7] Chunkun Bo, Ke Wang, Yanjun Qi, and Kevin Skadron. 2015. String Kernel Testing Acceleration using the Micron Automata Processor. In *Workshop on Computer Architecture for Machine Learning*.

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

[9] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. In *Architectural Support for Programming Languages and Operating Systems*. 577–589. https://doi.org/10.1145/2872362.2872409

[10] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8

[11] Z. P. Fry and W. Weimer. 2010. A human study of fault localization accuracy. In *International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609691

[12] J. Goeders and S. J. E. Wilton. 2014. Effective FPGA debug for high-level synthesis generated circuits. In *24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927498

[13] P. Graham, B. Nelson, and B. Hutchings. 2001. Instrumenting Bitstreams for Debugging FPGA Circuits. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*. 41–50.

[14] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. 2009. VIDA: Visual interactive debugging. In *International Conference on Software Engineering*. 583–586. https://doi.org/10.1109/ICSE.2009.5070561

[15] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Programming Language Design and Implementation*. 32–43. https://doi.org/10.1145/143095.143114

[16] Qiming Hou, Kun Zhou, and Baining Guo. 2009. Debugging GPU Stream Programs Through Automatic Dataflow Recording and Visualization. In *SIGGRAPH Asia*. Article 153, 11 pages. https://doi.org/10.1145/1661412.1618499

[17] IEEE Computer Society 2017. *IEEE International Workshop on Program Debugging (IWPD), Symposium on Software Reliability Engineering Workshops, ISSRE Workshops*. IEEE Computer Society, IEEE, Toulouse, France. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8108700

[18] Intel. 2017. Hyperscan. https://01.org/hyperscan. Accessed 2017-04-07.

[19] Peter B. Kessler. 1990. Fast Breakpoints: Design and Implementation. In *Programming Language Design and Implementation*. 78–84. https://doi.org/10.1145/93542.93555

[20] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-travel Debugging with First-class Traces. In *International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 352–361. http://dl.acm.org/citation.cfm?id=2486788.2486835

[21] G. Knittel, S. Mayer, and C. Rothlaender. 2008. Integrating Logic Analyzer Functionality into VHDL Designs. In *2008 International Conference on Reconfigurable Computing and FPGAs*. 127–132. https://doi.org/10.1109/ReConFig.2008.77

[22] Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *International Conference on Software Engineering*. 301–310. https://doi.org/10.1145/1368088.1368130

[23] T. J. Leblanc and J. M. Mellor-Crummey. 1987. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.* C-36, 4 (April 1987), 471–482. https://doi.org/10.1109/TC.1987.1676929

[24] Norman Matloff and Peter Jay Salzman. 2008. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, San Francisco, CA, USA.

[25] J. S. Monson. 2016. *Using Source-to-Source Transformations to Add Debug Observability to HLS-Synthesized Circuits*. Ph.D. Dissertation. Brigham Young University.

[26] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu chun Feng, and Michela Becchi. 2017. Demystifying Automata Processing: GPUs, FPGAs, or Micron's AP?

[27] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *International Symposium on Software Testing and Analysis*. 199–209. https://doi.org/10.1145/2001420.2001445

[28] B. Reagen, R. Adolf, Y. S. Shao, G. Y. Wei, and D. Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *International Symposium on Workload Characterization*. 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[29] P. Romero, B. du Boulay, R. Lutz, and R. Cox. 2003. The effects of graphical and textual visualisations in multi-representational debugging environments. In *Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*. 236–238. https://doi.org/10.1109/HCC.2003.1260234

[30] Indranil Roy. 2015. *Algorithmic Techniques for the Micron Automata Processor*. Ph.D. Dissertation. Georgia Institute of Technology.

[31] Indranil Roy and Srinivas Aluru. 2014. Finding Motifs in Biological Sequences Using the Micron Automata Processor. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*. 415–424. https://doi.org/10.1109/IPDPS.2014.51

[32] I. Roy, N. Jammula, and S. Aluru. 2016. Algorithmic Techniques for Solving Graph Problems on the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*. 283–292. https://doi.org/10.1109/IPDPS.2016.116

[33] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru. 2016. High Performance Pattern Matching Using the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*. 1123–1132. https://doi.org/10.1109/IPDPS.2016.94

[34] S. Sarkar, T. Majumder, A. Kalyanaraman, and P. P. Pande. 2010. Hardware accelerators for biocomputing: A survey. In *International Symposium on Circuits and Systems*. 3789–3792. https://doi.org/10.1109/ISCAS.2010.5537736

[35] E. Satterthwaite. 1972. Debugging tools for high level languages. *Software: Practice and Experience* 2, 3 (1972), 197–217. https://doi.org/10.1002/spe.4380020303

[36] M.L. Scott. 2015. *Programming Language Pragmatics*. Elsevier Science. https://books.google.com/books?id=jM-cBAAAQBAJ

[37] Y. S. Shao, B. Reagen, G. Y. Wei, and D. Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *International Symposium on Computer Architecture*. 97–108. https://doi.org/10.1109/ISCA.2014.6853196

[38] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology.

[39] Tamim Sookoor, Timothy Hnat, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. 2009. Macrodebugging: Global Views of Distributed Program Execution. In *Conference on Embedded Networked Sensor Systems*. 141–154. https://doi.org/10.1145/1644038.1644053

[40] Richard Stallman, Roland Pesch, and Stan Shebs. 2002. *Debugging with GDB*. Free Software Foundation.

[41] J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* 12, 3 (May 2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[42] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 259–272. https://doi.org/10.1145/3123939.3123986

[43] A. Tiwari and K. A. Tomko. 2003. 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.* 705–711. https://doi.org/10.1109/ASPDAC.2003.1195112

[44] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards Machine Learning on the Automata Processor. In *Proceedings of ISC High Performance Computing*. 200–218. https://doi.org/10.1007/978-3-319-41321-1_11

[45] Tommy Tracy II, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. 2015. Nondeterministic Finite Automata in Hardware—the Case of the Levenshtein Automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[46] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (April 1997), 38–43. https://doi.org/10.1145/248448.248457

[47] Jack Wadden, Kevin Angstadt, and Kevin Skadron. 2018. Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 749–761.

[48] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron. 2016. Generating efficient and high-quality pseudorandom behavior on Automata Processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 622–629. https://doi.org/10.1109/ICCD.2016.7753349

[49] J. Wadden, V. Dang, N. Brunelle, T. Tracy, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *International Symposium on Workload Characterization (IISWC '16)*. 1–12. https://doi.org/10.1109/IISWC.2016.7581271

[50] Jack Wadden and Kevin Skadron. 2016. *VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research*. Technical Report CS2016-03. University of Virginia.

[51] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential Pattern Mining with the Micron Automata Processor. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2903150.2903172

[52] Ke Wang, Mircea Stan, and Kevin Skadron. 2015. Association Rule Mining with the Micron Automata Processor. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*. http://www.cap.virginia.edu/sites/cap.virginia.edu/files/kwang_arm_submitted.pdf

[53] Michael H.L.S. Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. 2016. Using the Automata Processor for fast pattern recognition in high energy physics experiments - A proof of concept. *Nuclear Instruments and Methods in Physics Research* (2016). https://doi.org/10.1016/j.nima.2016.06.119

[54] Mark Weiser. 1982. Programmers Use Slices when Debugging. *Communications of the ACM ACM* 25, 7 (July 1982), 446–452. https://doi.org/10.1145/358557.358577

[55] Timothy Wheeler, Paul Graham, Brent E. Nelson, and Brad Hutchings. 2001. 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)*. Springer-Verlag, London, UK, UK, 483–492. http://dl.acm.org/citation.cfm?id=647928.739887

[56] Jacob O. Wobbrock, Leah Findlater, Darren Gergle, and James J. Higgins. 2011. The Aligned Rank Transform for Nonparametric Factorial Analyses Using Only Anova Procedures. In *Conference on Human Factors in Computing Systems*. 143–146. https://doi.org/10.1145/1978942.1978963

[57] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.23919/FPL.2017.8056759

[58] Xilinx Inc. 2016. *Integrated Logic Analyzer v6.2: LogiCORE IP Product Guide* (PG172 ed.). Xilinx Inc., San José, CA.

[59] Xilinx Inc. 2018. *Virtual Input/Output v3.0: LogiCORE IP Product Guide* (PG159 ed.). Xilinx Inc., San José, CA.

[60] Xilinx Inc. 2018. *Vivado Design Suite User Guide: Programming and Debugging* (ug908(v2018.1) ed.). Xilinx Inc., San José, CA.

[61] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. 2011. How do fixes become bugs?. In *Foundations of Software Engineering*. 26–36.

[62] Polle Trescott Zellweger. 1984. *Interactive Source-level Debugging for Optimized Programs (Compilation, High-level)*. Ph.D. Dissertation. University of California, Berkeley.

[63] Tianyi Zhang and Miryung Kim. 2017. Automated Transplantation and Differential Testing for Clones. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 665–676. https://doi.org/10.1109/ICSE.2017.67

[64] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. 2008. How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation. In *Compiler Construction*. Berlin, Heidelberg, 147–162. https://doi.org/10.1007/978-3-540-78791-4_10

[65] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API Mapping for Language Migration. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 195–204. https://doi.org/10.1145/1806799.1806831

[66] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Brill tagging on the Micron Automata Processor. In *Proceedings of the 9th IEEE International Conference on Semantic Computing*. 236–239. https://doi.org/10.1109/ICOSC.2015.7050812