# Portable Programming with RAPID

Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron, *Fellow, IEEE*

**Abstract**—As the hardware found within data centers becomes more heterogeneous, it is important to allow for efficient execution of algorithms across architectures. We present RAPID, a high-level programming language and combined imperative and declarative model for functionally- and performance-portable execution of sequential pattern-matching applications across CPUs, GPUs, Field-Programmable Gate Arrays (FPGAs), and Micron's D480 AP. RAPID is clear, maintainable, concise, and efficient both at compile and run time. Language features, such as code abstraction and parallel control structures, map well to pattern-matching problems, providing clarity and maintainability. For generation of efficient runtime code, we present algorithms to convert RAPID programs into finite automata. Our empirical evaluation of applications in the ANMLZoo benchmark suite demonstrates that the automata processing paradigm provides an abstraction that is portable across architectures. We evaluate RAPID programs against custom, baseline implementations previously demonstrated to be significantly accelerated. We also find that RAPID programs are much shorter in length, are expressible at a higher level of abstraction than their handcrafted counterparts, and yield generated code that is often more compact.

**Index Terms**—automata processing; sequential pattern matching; heterogeneous (hybrid) systems; concurrent, distributed, and parallel languages; concurrent programming structures, patterns

## 1 Introduction

Data is being collected by companies and researchers alike at increasing rates. The Computer Sciences Corporation projects that data production worldwide will grow to 35 zettabytes by 2020, an amount 44 times greater than the amount produced in 2009 [1]. While processing the growing quantity of data is a technical challenge in and of itself, there is also a demand by business leaders to have real-time analyses [2]. New techniques and algorithms are needed to support these high-speed analyses of growing data sets. New approaches should be able to support high throughputs while processing large data sets, be easy to program and maintain, and be portable across hardware architectures found in data centers (e.g., CPUs, GPUs, FPGAs, etc.).

One technique is to quickly scan the data for "interesting" regions (the definition of interesting varies between applications), and return to these regions to perform a more thorough analysis later, reducing the amount of data being processed by a complex algorithm. The initial scan can often be re-phrased as a pattern-searching problem, in which many searches are conducted against a single stream of data. A pattern defines a sequence of data that should be identified within another collection of data. Nondeterministic finite automata (NFAs) are a common computational paradigm for recognizing patterns in a data stream. As a theoretical framework, NFAs are capable of the highly parallel searches needed for performing several important types of analysis. Additionally, years of research and tool development have resulted in high-throughput *automata processing* architectures and software engines [3]–[9].

While these hardware solutions provide high throughputs for pattern searches, programming them can be challenging. Current programming models are akin to assembly-level development on traditional CPU architectures. Consequently, programs written for these accelerators are tedious to develop and challenging to write correctly. Additionally, these low-level representations do not lend themselves well to debugging and maintenance tasks.

We observe that *automata processing provides a suitable abstraction to support portability across hardware back-ends for pattern-searching problems, but a higher level of abstraction is needed to improve the ease of programming.*

In this article, we first evaluate the extent to which automata processing enables the portability of applications across CPUs, GPUs, and FPGAs. Then we extend our previous work to develop a high-level programming language, RAPID [10], for representing pattern search problems with respect to NFAs, targeting Micron's Automata Processor (AP), CPUs, GPUs, and FPGAs. Together, these two contributions provide a programming model that is portable, reduces code size, and improves maintainability.

### 1.1 Automata Processing Portability

To evaluate the portability of the automata processing paradigm, we consider two questions: 1) do design and optimization choices for finite automata port across architectures? and 2) to what extent does automata processing support high performance across architectures?

We observe that there exist automata processing engines for all mainstream architectures (CPUs, GPUs, FPGAs) as well as specialized architectures such as the UAP [11] and Micron's D480 AP [5]. Further, previous results demonstrate that the D480 AP, UAP, and FPGAs achieve high throughputs for automata applications [11]–[19]. In this article,

- *K. Angstadt, J. Wadden, and W. Weimer are with the Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: {angstadt, wadden, weimerw}@umich.edu.*
- *K. Skadron is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904. E-mail: {skadron}@virginia.edu.*

*Manuscript received August 24, 2017. Manuscript revised May 3, 2018 and August 7, 2018.*

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

### 1.2 RAPID Programming

While automata processing provides a suitable abstraction for performance portability, finite automata programming is tedious and error-prone. Therefore, we extend our previous work on the development of the RAPID programming language to support CPU, GPU, and FPGA back-ends [10]. RAPID is a high-level language that maintains the performance and portability benefits of automata processing while also providing concise, clear, maintainable, and efficient representations of pattern-matching algorithms.

In this article, we present algorithms for converting RAPID programs into NFAs for execution via automata processing. We describe code generation and tool pipelines that are efficient across all target architectures.

We evaluate the efficiency of compiled RAPID programs against handcrafted equivalents, measuring program size, resource utilization, and runtime performance. These programs are based on real-world applications that have significant speedups when executed using specialized hardware accelerators. Our evaluation demonstrates that RAPID programs introduce little overhead compared with applications written at a lower level of abstraction and maintain the performance and functional portability provided by the automata paradigm.

### 1.3 Contributions

In this article, we make the following contributions:

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

The remainder of this article is organized as follows. Section 2 introduces background information and discusses related work. In Section 3, we present our empirical evaluation of automata processing stability with respect to state of the art algorithms. Section 4 describes the RAPID programming language. Next, Section 5 describes the algorithms for generating Finite Automata from a RAPID program. Section 6 discusses the tool pipelines for compiling and executing Finite Automata applications on CPUs, GPUs, FPGAs, and the D480 AP. In Section 7, we evaluate the performance of the RAPID programming language. Finally, we discuss our conclusions in Section 8.

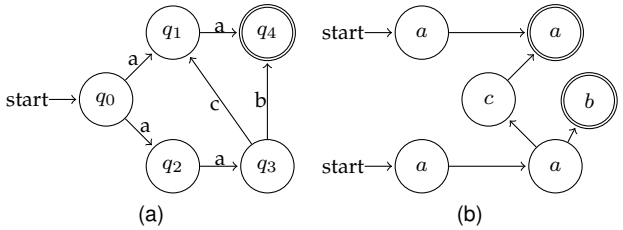


Fig. 1. A behaviorally equivalent NFA and homogeneous NFA (both accept exactly `aa`, `aab`, and `aaca`). Note that there is a singleton start state in (a) (i.e., $Q_{start} = \{q_0\}$), but there are two start states in (b).

## 2 Background and Related Work

In this section, we describe background material related to automata processing, spatial architectures discussed in this article, and programming models for pattern searches and portability across architectures. We also discuss similarities and differences between RAPID and this related work.

### 2.1 Finite Automata

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

An NFA modifies this five-tuple to be $(Q, \Sigma, Q_{start}, \delta, F)$, where $Q_{start} \subseteq Q$ is a set of initial states and $\delta : 2^Q \times \Sigma \rightarrow 2^Q$ is the transition function.[1] Note that non-determinism in terms of finite state machines does not refer to stochastic non-determinism, but rather refers to the transition function, which given a set of active states and symbol, returns a new set of active states. This allows for multiple transitions to occur for every symbol processed, effectively forming a tree of computation. NFAs have the same representative power as DFAs, but have the advantage of being more spatially compact [20].

In this article, we use an alternate form of NFAs known as *homogeneous* NFAs. These automata restrict the possible transition rules such that all incoming transitions to a state must occur on the same symbol. Because all transitions to a state occur on the same symbol, we can label states with

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

symbols rather than labeling the transitions. We refer to these combined states and labels as *state transition elements* (STEs), following the nomenclature adopted by Dlugosch et al. [5]. An STE *accepts* the symbols in its label, which we we refer to as the *character class* of the STE. Figure 1 depicts an NFA and a behaviorally equivalent homogeneous NFA. Additionally, we relax the definition of machine acceptance. Instead of accepting if an accepting state is active at the end of input, whenever an accepting state is active we report the relative offset in the input stream. This allows for pattern-recognition in streams of data symbols.

## 2.2 Architectural Support for Automata Processing

There are several custom and customizable platforms that support automata processing. In this section, we provide additional background for two: field-programmable gate arrays, and Micron's D480 AP.

### 2.2.1 Field-Programmable Gate Arrays

FPGAs are reconfigurable fabrics of look-up tables (LUTs). Individual LUTs can be configured to perform the computation of arbitrary logic gates. LUTs can be connected together via the reconfigurable fabric to form arbitrary circuits.

Traditionally, FPGAs have been used for prototyping logic circuits. However, FPGAs are increasingly used as co-processors to accelerate computation that does not map well to von Neumann execution. Usually, computations that can be represented as streaming systolic arrays perform extremely well when mapped to FPGA fabrics in contrast to von Neumann architectures like CPUs and GPUs.

Prior work has investigated implementing finite automata processing on FPGAs [8], [9], [21]–[24]. Because automata can be thought of as circuits—where each state transition element is a specialized logic gate—they can be naturally implemented in an FPGA fabric. Although FPGAs are a natural and successful fit for acceleration of automata processing and have been the subject of significant study, the ability to port software to FPGAs while maintaining performance remains an open research problem.

### 2.2.2 Micron's D480 Automata Processor

The AP, as described by Dlugosch et al. [5], is a hierarchical, memory-derived architecture for direct execution of homogeneous non-deterministic finite automata. State Transition Elements (STEs) are stored in a memory array, and transitions between STEs are encoded in a reconfigurable routing matrix. The memory array and routing matrix form the basis of the AP architecture.

An SDRAM memory array serves as a computational medium in the AP, a stark contrast from its traditional role as main memory in a von Neumann computer system. Arranged as a two dimensional grid, SDRAM data is accessed via row and column addresses. STEs consist of a single column of memory and a detection cell used for storing whether the given STE is active. The design in Figure 1b would require seven columns of SDRAM, one for each of the STEs. The symbol or symbols accepted by an STE are stored in the column of memory, each row representing a symbol in the alphabet. At runtime, a symbol from the input stream is decoded and drives one of the rows in the memory array.

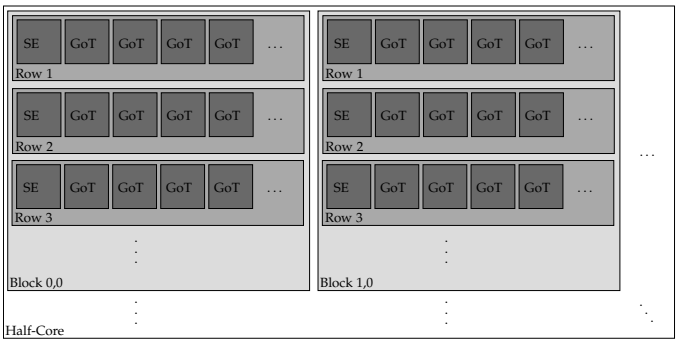


Fig. 2. Hierarchical relationships between AP components: *groups of two* (GoT) and a *special purpose element* (SE) form a *row*, rows form *blocks*, and blocks form a *half-core*

TABLE 1
Resources on the first-generation AP board, containing 32 chips

| Total STEs | Total Counters | Total Boolean Logic Elements | Total Blocks | Half-Cores / Chip |
|---|---|---|---|---|
| 1,572,864 | 24,576 | 73,728 | 6,144 | 2 |
| **STEs / Row** | **Rows / Block** | **Counters / Block** | **Boolean / Block** | **Blocks / Half-Core** |
| 16 | 16 | 4 | 12 | 96 |

Simultaneously, all STEs (columns of memory) determine whether they accept that symbol. Accepting STEs (i.e., those currently active as determined via their detection cells) then generate an output signal that is passed through the routing matrix to activate the connected STEs.

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

A hierarchical, reconfigurable routing matrix is used to route activation signals between STEs and special elements. Groupings of two STEs form a *GoT*. Several GoTs and a special purpose element (SE) are then connected to form a *row* in the routing matrix. Groupings of rows form *blocks*, and groupings of blocks form a *half-core*. There is no routing between the two half-cores that make up an AP *chip*. An AP *board* consists of several AP chips, and this allows for many pattern-recognition searches to occur in parallel. Figure 2 depicts this hierarchy for a half-core. Table 1 provides resource information for the first-generation AP board. We can take advantage of this hierarchical routing to produce efficient automata from RAPID programs.

## 2.3 Programming Models for Pattern Searches

Next, we consider current programming models for identifying patterns within a stream of data.

### 2.3.1 Graph-Based Representations

Traditionally, NFAs are often represented as a directed graph with states as vertices and the transition function en-

coded as edges. While capable of specifying search patterns, NFAs are difficult to write and maintain. NFA formats, such as the XML-based Automata Network Markup Language (ANML) and Becchi's transition table representation [25], are extremely verbose. For example, measuring the pairwise difference of characters between an input string and a fixed five-character string requires 62 lines of ANML to represent [26]. Maintenance tasks on this code are also cumbersome: changing such an automaton to compare against a string of length 12 requires modification of 65% of the code. NFAs can be challenging and tedious to write correctly, especially for developers lacking familiarity with automata theory. In research areas such as program verification, the task of specifying automata is automated [27].

#### 2.3.2 Regular Expressions

Regular expressions are another common option for representing a search pattern; however, these also suffer from similar maintainability challenges. For many of our target applications, such as motif searches, particle tracking, and rule mining, the regular expression representing the search is non-intuitive and may simply be an exhaustive enumeration of all possible strings that should be matched (much in the same way an overfit machine learning classifier might directly encode a lookup table of the training data). Additionally, programming of regular expressions can be extremely error-prone due to variations in regular expression syntax, which leads to high rates of runtime exceptions [28].

#### 2.3.3 Languages for Streaming Applications.

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

Languages for streaming applications, such as StreamIt [29], have been studied in great detail. StreamIt provides structures for stream pipelining, splitting and joining, and feedback loops. StreamIt objects may peek and pop from the input stream, store input, and perform computations before outputting a result. Automata processing, however, does not readily admit this computational model. Finite automata have no inherent memory, and cannot generally peek at the input stream. Many of the operations allowed by StreamIt are thus not applicable in this domain, and it is not evident how to extend the StreamIt model to describe complex automata nor non-deterministic execution. Additionally, data and control are treated differently in StreamIt and automata processing-based languages such as RAPID (see [10] and Section 4). A StreamIt program specifies a stream graph: data always enters the program and is transformed and passed downstream until reaching an output. In RAPID, instructions describe the next step to be taken to identify a pattern: stream data enters the program in the location(s) where the program is currently active, causing control to shift to another statement in the program. Ultimately, StreamIt and RAPID target different computational abstractions, and are not directly comparable (e.g., it is not clear how to compile StreamIt programs to finite automata).

#### 2.3.4 Non-Deterministic Languages.

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

Dijkstra's Guarded Command Language [30] introduces non-deterministic alternative and repetitive constructs. These constructs are predicated with a Boolean "guard" that must be true for the encapsulated statements to execute. The alternative construct chooses one command with a satisfied guard and executes it. In the repetitive construct, the program loops, choosing one command with a satisfied guard to execute, until no guards are satisfied. Rather than proposing a concrete language, the Guarded Command Language presents guiding formalisms for supporting non-determinism. We provide similar constructs in RAPID, but a notion of parallel exploration is incorporated directly into the semantics, allowing RAPID to be more concise than the Guarded Command Language when processing stream or pattern data.

An additional non-deterministic programming language is Alma-0 [31], a declarative extension of Modula-2. Alma-0 supports the use of Boolean expressions as statements, an `ORELSE` statement allowing for execution of multiple paths through the program, and a `SOME` statement that is the non-deterministic dual of the `FOR` statement. While RAPID also treats Boolean expressions as statements (see Section 4.1), it differs from Alma-0 in the computational model supported by the language's semantics. In Alma-0, `ORELSE` and `SOME` are defined via backtracking. Execution is single threaded: when an `ORELSE` statement or a `SOME` statement is encountered, the program will choose a single option to execute. If an exploration fails, the program backtracks to the last choice point, restoring all program state, and attempts a different option. Rather than choosing a single option to explore and backtracking if computation fails, RAPID programs explore all paths in parallel.

### 2.4 Programming Models for Portability

The holy grail of programming for heterogeneous environments is to "write once and run anywhere." Research into portability dates back decades and has its origins in attempts to support multiple mainframe computer architectures. For example, the Parallel Programming Language (PPL) was a strongly typed language that abstracted away from machine-dependent types to support multiple architectural targets [32]. More recently, the focus has been on supporting portability across different coprocessors.

The OpenCL language boasts support for CPUs, GPUs, FPGAs, and other microprocessors [33]. The language provides an abstract notion of computational devices and processing elements, which allow for task- and data-parallel applications to be executed in heterogeneous environments. While the same code can be run on multiple types of hardware, code written for one architecture rarely performs well on another architecture. To execute efficiently on both GPUs and FPGAs, for example, a developer must often write two copies of the application, crafting the code to

TABLE 2
Performance stability of OpenCL programs

| Benchmark | CPU | GPU | FPGA | Stable |
|---|---|---|---|---|
| CFD | ↓ | ↓ | ↑ | ✗ |
| Hotspot | ↓ | ↓ | ↑ | ✗ |
| LUD | ↑ | ↑ | ↓ | ✗ |
| NW | ↓ | ↓ | ↑ | ✗ |
| Pathfinder | ↓ | ↓ | ↑ | ✗ |
| SRAD | ↓ | ↓ | ↑ | ✗ |

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

TABLE 3
Performance stability of Automata Processing optimizations

| Optimization | CPU | GPU | FPGA | AP | Stable |
|---|---|---|---|---|---|
| Automata Folding | ↑ | ↑ | ↑ | ↑ | ✓ |
| Counters | ↓ | n/a | ↓ | ↑ | ✗ |
| DRM | — | — | — | ↑ | ✓ |
| Prefix Collapsing | ↑ | ↑ | ↑ | ↑ | ✓ |
| Race Logic | ↓ | ↓ | ↓ | ↓ | ✓ |
| Striding | ↑ | ↑ | ↑ | ↑ | ✓ |

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

make use of the particular strengths of each platform. Our goal with RAPID is to avoid this rewriting step, allowing the developer to write an application using a computational abstraction that performs well *across architectures*.

High-level constructs, such as Map-Reduce, have been demonstrated to provide portability across architectures [34], [35]. RAPID also makes use of high-level constructs, but constructs in our language are more specific to sequential pattern identification tasks.

## 3 Automata Processing Stability

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

### 3.1 Performance Stability

We first compare the stability of design choices in automata processing applications with the stability of those in OpenCL. OpenCL supports execution across a variety of architectures [33]. However, code written for one processor may not compile for another target or may require significant re-writing to be performant on the new architecture [36]. Given two implementations of the same application and two hardware architectures, if one implementation outperforms the other on the first architecture and the opposite is true for the second architecture, we say that there is a *performance inversion*. Performance *stability* is the lack of observable performance inversions.

The OpenCL language has many observable performance inversions and is therefore not stable across architectures. We demonstrate such inversions using applications in the Rodinia HPC benchmark suite, which were optimized for multi-threaded execution [37]. Zohouri et al. have developed a second implementation based on an iterative approach [36]. For each benchmark, we time both implementations on the CPU, GPU, and FPGA. Table 2 presents high-level relative performance results for loop- and thread-based OpenCL Rodinia benchmarks; performance is stable if arrows within a row do not reverse direction. We find that all six benchmarks demonstrate performance inversions. That is, for all benchmarks we consider, the design decisions needed for performant code vary with each architecture.

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

- **Automata folding [16]**: reducing automata states by combining non-overlapping input comparisons.
- **Counters [5]**: reducing states by rewriting automata to use saturating counters.
- **Disjoint Report Merging (DRM) [38]**: reducing data transfer overheads on spatial automata processors.
- **Prefix collapsing [21]**: combining common automata states to form a trie-like structure.
- **Race Logic [39]**: providing general support for dynamic programming at the cost of performance.
- **Striding [21]**: transforming automata to support compressed input streams.

For each, we select an arbitrary application that supports the optimization[2] from the ANMLZoo automata processing benchmark suite [40]. Using one implementation with the optimization and one without, we measure relative performance (i.e., relative time-to-solution) across CPUs, GPUs, FPGAs, and the AP. These results are presented in Table 3. We observe only a single performance inversion (counters) in our experiments and believe this inversion is an artifact of current implementation support.[3] These results provide initial evidence that the automata processing abstraction provides stable performance across architectures for many implementations and optimizations. Design decisions for performant code in automata processing do not appear to vary as much across architectures as they do with OpenCL.

### 3.2 Automata Processing Performance

In addition to stability, performant code across architectures is a desirable quality of a portable programming model. Recent studies by Wadden et al. and Nourian et al. investigate (and demonstrate) the performance of automata processing on several hardware accelerators, including GPUs, FPGAs, and the AP [40], [41]. Therefore, we restrict our attention in

2. No support results in the optimization being a no-op and thus has no impact on stability.

3. Nourian et al. support counters on GPUs [41], but their software artifacts have not been made public. Performance on the CPU and FPGA is degraded due to the complexity of circuit simulation (CPU) and routing constraints (FPGA). AP counters are discussed in Section 7.

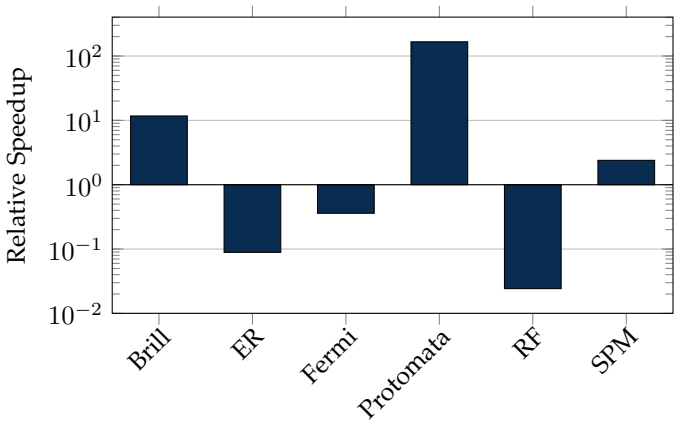


Fig. 3. Relative performance of automata processing vs. application-specific algorithms on the CPU. Higher bars indicate better performance of the automata-based algorithm. Note that the y-axis is log-scale.

this article to CPU-based automata processing. We compare the performance of applications mapped to the automata processing paradigm with state-of-the-art CPU algorithms.

We evaluate all applications from the ANMLZoo benchmark suite [40] that were adapted from state-of-the-art, non automata-based algorithms. These applications are:

- **Brill [12]**, a rule-writing processor for part of speech tagging in natural language processing.
- **Entity Resolution (ER) [42]**, an algorithm for detecting duplicated (or similar) names from a list.
- **Fermi [13]**, a path recognition algorithm for particle physics experiments.
- **Protomata [43]**, a protein motif signature classification application.
- **Random Forest (RF) [16]**, a random forest ensemble classifier for handwriting recognition.
- **SPM [15]**, a sequential pattern mining application.

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

For each experiment, we executed the state-of-the-art implementation ten times and measured the average throughput of the core algorithm. Then, we averaged ten runs of an automata engine running the same application. We executed the benchmark automata using the Intel Hyperscan framework supplied as part of MNCaRT [44]. Experiments were performed on an Intel Core i7-5820K (3.30 GHz) processor with six physical cores and 32GB of RAM.

Figure 3 shows the relative speedup of automata engines over application-specific algorithms on the CPU. For three applications (Brill, Protomata, and SPM), the automata-based algorithm outperforms the state-of-the-art in terms of average throughput. By representing Brill and Protomata as automata, new opportunities for optimization are exposed, allowing for orders of magnitude increased performance. For Fermi, the automata algorithm is within $3\times$ of the application-specific algorithm. Entity Resolution and Random Forest are an order of magnitude slower primarily due to the increased accuracy and/or work [45] of the automata implementations. When adapting a new application to the automata paradigm, researchers should consider carefully how this might impact the work—the time or steps needed for a serial processor to complete the task—performed by the algorithm. Large increases in work may not be suitable for performant automata processing algorithms across architectures.

### 3.3 Discussion

We observe that automata processing is more stable across disparate architectures relative to design choices and optimizations than the OpenCL programming model. We also observe that four of our six automata benchmarks perform within $3\times$ application-specific algorithms on the CPU, and two of these state machine-based implementation are at least an order of magnitude faster than the state-of-the-art. Additionally, automata processing is already a widely used computational model in areas such as network security [46], computational finance [47], and software engineering [27], [48]. There has been significant development of new optimizations for state machine performance on CPUs [25], and we anticipate continued improvement of automata processing performance on von Neumann architectures.

We conclude that automata processing provides *stability* (Section 3.1) and *performance* (Section 3.2) across architectures and implementations, including CPUs, GPUs, FPGAs, and the AP. That is, the performance of an automata-based algorithm is stable across architectures and often similar to the performance of an application-specific algorithm on the same hardware platform. Note that this performance portability includes applications that go beyond traditional regular expression-based algorithms, even on the CPU. We believe that portability across these architectures is beneficial to both the research and end-user communities. In particular, there is a lower overhead and risk incurred by developers who learn a programming model that is usable on multiple architectures. We believe that automata processing provides a *suitable abstraction* for representing and porting computation across multiple, dissimilar computer architectures.

## 4 The RAPID Language

While automata processing provides a suitable abstraction for performance portable execution of algorithms, current programming models for pattern searches have significant drawbacks (see Section 2.3). In this section, we discuss a new programming language, RAPID, which allows developers to write concise, clear, and maintainable algorithms for use with automata engines. In particular, RAPID supports searching a stream of data for many patterns in parallel. Programs are written in a combined imperative and declarative style using a C-like syntax. In this section, we present a high-level overview of the control structures and data representations in the RAPID programming language.

### 4.1 Program Structure

**Macros and Networks.** RAPID programs consist of one or more *macros* and a *network*. The basic unit of computation in a RAPID program is a *macro*, which defines a reusable

```
1 macro hamming_distance (String s, int d) {
2     Counter cnt;
3     foreach (char c : s)
4         if(c != input()) cnt.count();
5     cnt <= d;
6     report;
7 }
8 network (String[] comparisons) {
9     some(String s : comparisons)
10        hamming_distance(s,5);
11 }
```

Fig. 4. A RAPID program for computing Hamming distances

pattern-matching algorithm. Macros in RAPID share similarities with both C-style macros and ANML macros, allowing code to be written once and then used as a "rubber stamp." RAPID macros admit more customized usage than their namesakes in C and ANML; the same macro can generate all designs for a particular problem.

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

Additionally, macros can instantiate other macros. When a macro is called, control shifts to the called macro; all of its statements are executed, and then control returns to the calling macro. While the macro code defines how to identify a pattern in the input stream, the macro parameters can specify the particular characters to match, allowing for comparisons of varying lengths. Consider the macro in Figure 4, which performs a Hamming distance computation between a string parameter, `s`, and the input stream. Changing from comparison against a string of length five to a string of length twelve only requires passing a different string argument to the macro. As noted in Section 2, more than half of the code in the corresponding ANML implementation must be modified to make an identical change.

The *network* represents the highest level of pattern-matching within a RAPID program, and statements within a network definition are executed in parallel. The most common use of the network is to define a collection of macros for instantiation, which are executed in parallel at runtime to identify patterns in the input data stream. The network may also have parameters to specify certain values at runtime. Figure 4 contains a RAPID program that computes the Hamming distance for a number of given strings and reports on input within a distance of five. The network is parameterized on an array of strings, which is used at runtime to specify the comparisons being made.

**Reporting.** RAPID programs passively observe the input data stream; they cannot modify the stream. Programs can indicate interesting regions within the stream by using the `report` statement, which generates a *report event*. These events provide the offset in the input data stream where the report occurred and additional identifying meta data, such as the reporting macro. For the program in Figure 4, reports indicate offsets where the input stream is within a Hamming distance of five from the strings in `comparisons`.

```
1 Counter cnt;
2 foreach(char c : "rapid") {
3     if( c == input() ) cnt.count();
4 }
5 if( cnt >= 3 ) report;
```

Fig. 5. The above code counts the number of characters matched in "`rapid`" and reports if the count is at least three

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

### 4.2 Types and Data in RAPID

There are five primary data types in RAPID: `char`, `int`, `bool`, `String`, and `Counter`. Both `String` and `Counter` are lightweight objects, while the remaining three are primitive types. Additionally, there is support for nested arrays of these types.

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

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

Figure 5 demonstrates the usage of counters and interacting with the input stream. The `foreach` loop iterates over each character in the string "`rapid`" sequentially. If that character matches the next character from the input stream, the counter is incremented. After iterating over the entire string, the program checks if the counter is at least three and reports if so. For example, if the stream contained "`tepid`," the count would be three, and there would be a report, but "`party`" results in a count of one and no report.

The input data stream in RAPID is privileged and is accessed via the `input()` function. A call to this function returns a single character from the head of the data stream. Access to the input data is destructive—no peeking or insertion is allowed during program execution. Calls to `input()` act as synchronization points across active threads

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

in a RAPID program. Similar to how active states in an NFA process the same input symbol, all active threads execute up to an `input()` statement and then receive the same character from the input stream. For example, if the stream contains "`abcd...`," `input()` would return '`a`' to all active threads of computation, and the stream would now contain "`bcd...`" There is no required number of calls to `input()` across threads and also no communication between threads. Threads with fewer calls to input() than another thread will simply terminate earlier. This data model supports the heterogeneity of MISD computations.

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

### 4.3 Parallel Control Structures

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

**`Either/Orelse` Statements.** This structure provides basic support for parallel exploration. An `either/orelse` statement consists of two or more blocks, which allows for an arbitrary, static number of parallel computations. Computation splits when an `either/orelse` statement is encountered during execution, and each of the blocks is executed in parallel. When the end of a block is reached, computation continues with the next statement in the program. No blocking or joining occurs, meaning that different paths in the `either/orelse` statement may begin executing the following statement at different times. This behavior is desirable because it allows for the matching of different length patterns containing the same suffix.

As an example usage of the `either/orelse` statement, consider the code fragment in Figure 6, adapted from the *MOTOMATA* benchmark [17] evaluated in Section 7. Candidates in the input stream are separated by the control character '`y`'. The computation should report the candidates within a Hamming distance of `d` from the string stored in variable `s`. We use an `either/orelse` statement to ensure that computation continues to the next candidate when the current candidate does not fall within the threshold. The first block of the `either/orelse` statement performs the Hamming distance comparison, while the second block consumes input until the control character is reached, always preparing the program to check the next candidate.

```
1 either {
2     hamming_distance(s,d); //hamming distance
3     'y' == input();        //next input is 'y'
4     report;                //report candidate
5 } orelse {
6     while('y' != input()); //consume until 'y'
7 }
```

Fig. 6. An example usage of an either/orelse statement

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

This statement is the parallel dual of a `foreach` loop. During execution, the program iterates over a provided array or string and instantiates a parallel thread of execution for each item. Similar to an `either/orelse` statement, the execution of each parallel thread continues with the subsequent statement in the program; different threads in the some statement may reach this next statement at disjoint times. The `some` statement in Figure 4 instantiates a Hamming distance macro for each string in the `comparisons` array. The number of parallel threads executed depends on the number of entries in `comparisons`.

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

The code fragment in Figure 7 will perform a sliding window search for the string "`rapid`." The predicate within the guard will return true on any input, and therefore the block of code will begin execution at every character in the input stream. The `whenever` statement can also perform restricted sliding window searches depending on the predicate in the guard. For example, an application searching through HTTP transactions might use the predicate matching "`GET`" before matching specific URLs.

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

## 5 Code Generation

In this section, we present techniques for converting RAPID programs into automata for execution with automata pro-

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

Fig. 7. Execution of a sliding window search over the entire input stream for the string "rapid"

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

We employ a staged computation model to convert RAPID programs: comparisons with the input stream and counters occur at runtime, while all other values are resolved at compile time. To aid in partitioning, we annotate expressions with their return type during type checking. Allowable annotations include the five types listed in Section 4.2 as well as an internal `Automata` type, which denote expressions interacting with the input stream. Expressions annotated with `Automata` or `Counter` are converted into ANML (allowing for runtime execution), while the remaining expressions are evaluated during compilation.

Our conversion algorithm recursively transforms RAPID programs into finite automata in much the same way that regular expressions can be transformed into NFAs. Comparisons with the input stream are transformed into STEs. The statement in which the comparison occurs determines how the STEs attach to the rest of the automaton. Rules for transforming automata expressions determine the structure of the STEs within a given statement. We describe the conversion of expressions, statements and counters in turn.

### 5.1 Converting Expressions

Expression transformation results in the formation of a chain of STEs. No cycles are generated by expressions, but chains may include bifurcations. Figure 8 provides examples of transformations from RAPID expressions to automata structures. The most basic transformation is a comparison between a character and the input stream, generating a single STE. AND expressions behave as concatenation because reading from the input stream is destructive. The conversion of an OR expression generates a bifurcation in the generated automaton. A special case occurs when both sides of the OR expression contain input comparisons of length one. In these instances, we take advantage of STE character classes to specify multiple accepting symbols for a single STE.

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

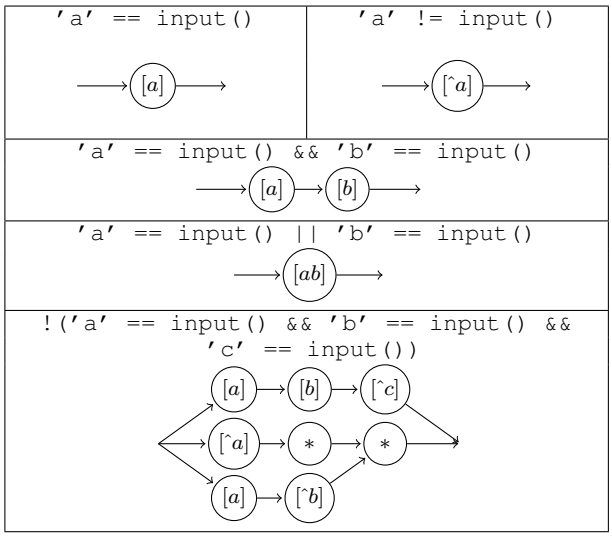


Fig. 8. Example transformations of RAPID expressions into automata

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

### 5.2 Converting Statements

Statements in RAPID are transformed into the high-level automaton structures, allowing for additional pipelining, feedback loops, and parallel exploration of patterns. We present the overall structures in Figure 9.

A `foreach` loop is unrolled into straight-line pattern-matching. Parallel `either/orelse` and `some` statements are transformed by generating the code for each statement and connecting these structures in parallel into the overall design. This mirrors the language semantics that the `some` statement is the parallel dual of `foreach`. Note that `some` statements typically depend on compile-time parameters (via input annotations on the network) while `either/orelse` statements do not (see Section 4.3).

There is also a similarity between `while` loops and `whenever` statements. `While` loops alternately perform predicate checks and execute the body code. This generates a feedback loop structure in the automaton. In a `whenever` statement, predicate checking begins on every character consumed. To support this, we generate a self-activating STE that accepts all symbols (see $*$ node in Figure 9d). This added STE maintains an active transition into the predicate, allowing matching to begin on every symbol consumed. Once the predicate accepts, the body of the `whenever` statement will begin to execute (although the predicate is still checked again in parallel on subsequent input characters).

### 5.3 Converting Counters

Counters in RAPID are challenging to implement because the state of a hardware counter on the AP cannot be directly accessed. Therefore, counter comparisons in RAPID programs are transformed into a pattern-matching operation

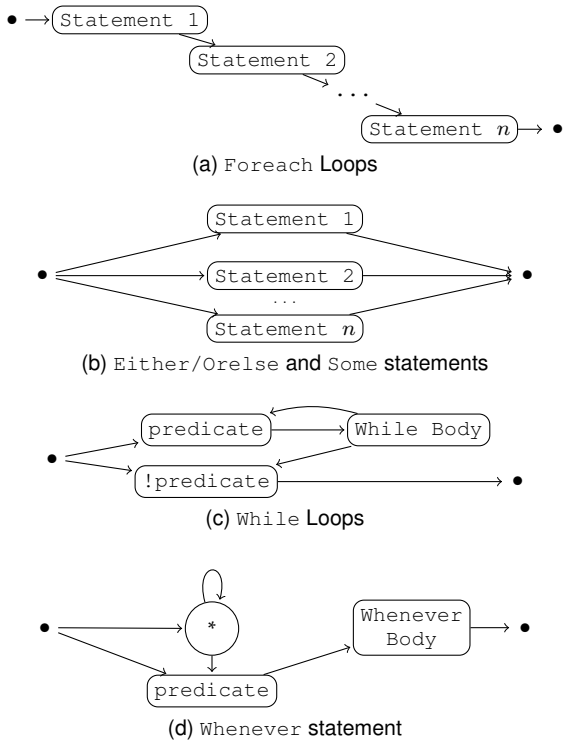


Fig. 9. Automaton designs for RAPID statements

using a combination of one or more saturating counters and Boolean gates. The basic structure consists of a saturating counter set to latch (once the threshold is reached, the output signal remains active) and an inverter, which allows for detection of the counter target not being reached.

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

We follow the set of rules for determining the threshold and outputs of a `Counter` shown in Table 4. Equality checking with a `Counter` requires the use of two physical counter elements. While traversing the program, we note which `Counter` objects are used for equality checking and during code generation emit two counter elements for each.

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

We must also support the use of `Counter` variables as predicates in a `whenever` statement. For the body of a `whenever` statement to execute, the `Counter` must have reached its threshold, and the statement itself must have been reached within the control flow of the RAPID program. We use a self-activating STE matching all symbols to track when the statement is reached. An AND gate checks both of these conditions before executing the body of the `whenever` statement. This design is demonstrated in Figure 10.

TABLE 4
Rules for thresholds and outputs on counters

| Comparison | Threshold | True Output |
|---|---|---|
| `< x` | `x` | inverted |
| `<= x` | `x+1` | inverted |
| `> x` | `x+1` | non-inverted |
| `>= x` | `x` | non-inverted |
| `== x` | convert to `<= x && >= x` | |
| `!= x` | convert to `< x \|\| > x` | |

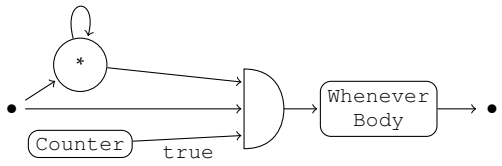


Fig. 10. Structure of `whenever` statement with counters

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

We attempt to automatically determine the pattern for inserting the count threshold symbol into the input stream. An example pattern is "insert the symbol after every 25 characters in the input stream." Often, the compiler can infer the pattern by counting the number of symbols consumed before the counter check occurs. When certain `while` loops are included in the program, however, it may not be possible to determine where in the input stream to inject the symbols. In these cases, we currently output a warning at compile time and rely on the developer to provide the pattern for inserting the control character into the data stream.

## 6 Executing RAPID Programs

A primary goal of the RAPID programming language is to support cross-platform portability of pattern searching applications. This allows an application to be tested on a developer's machine, which might not contain high-performance hardware, and be easily deployed into a heterogeneous hardware environment. Finite automata provide a portable, intermediate computation form that can be ported to many hardware back-ends, including CPUs, GPUs, FPGAs, and Micron's D480 AP. We achieve this by developing and adapting automata engines for each platform.

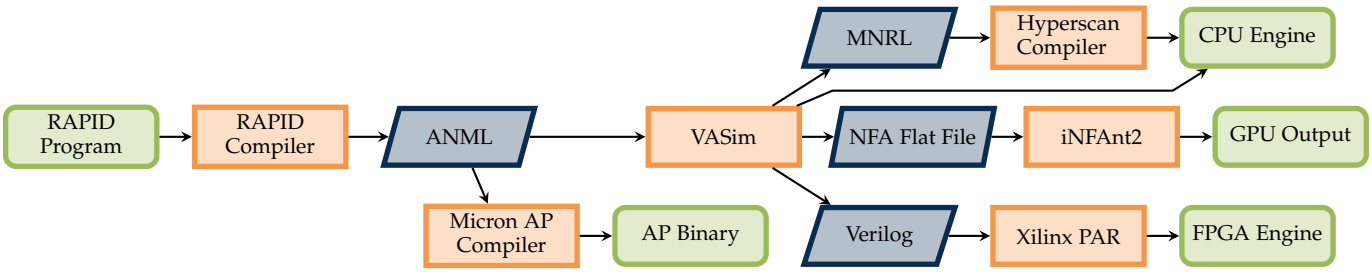


Fig. 11. Supported pipelines for executing RAPID programs. RAPID programs can be executed on CPUs (using VASim or Hyperscan), GPUs (using iNFAnt2), FPGAs, and Micron's D480 AP. Rounded green boxes indicate the input and output of the pipeline. Orange rectangles are software tools used to generate intermediate and output files. Blue parallelograms are intermediate files generated by our pipeline.

As discussed in Section 3, automata processing provides a suitable abstraction for efficient execution of applications across architectures. Such an approach effectively decouples high-level application development from low-level optimizations. Any advances in automata processing performance (e.g. new optimizations and new computational approaches) can be beneficial for all high-level applications.

In the previous section, we described the process for compiling a high-level RAPID program to finite automata. Now, we discuss workflows for executing automata across common computer architectures. Figure 11 outlines our workflow for targeting CPUs, GPUs, FPGAs, and the AP.

### 6.1 Targeting the Automata Processor

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

### 6.2 Targeting CPUs

We have developed and collected a set of algorithms for optimizing and transforming finite automata. These algorithms are implemented in VASim, a tool we created to facilitate automata research and experimentation [7]. This framework supports easy prototyping, debugging, simulation, and analysis of automata-based applications and architectures. We use VASim to optimize the automaton from the RAPID compiler using *common prefix collapsing* [21]. This process merges states that match the same input symbols, beginning with the starting states, producing a functionally-equivalent NFA with fewer states. In our Brill tagging benchmark, for example, prefix collapsing results in a 57% reduction in the number of states. Additionally, VASim contains a multi-threaded simulation core, which is capable of executing automata on an input stream. The simulator was designed specifically to execute ANML files, making VASim an excellent candidate for a RAPID CPU back-end.

While VASim is currently $4\times$–$694\times$ faster than existing simulation tools for Micron's AP, regular expression processors, such as Hyperscan [6] outperform VASim for pure NFA applications. When a compiled RAPID program contain no counters, we choose to execute with Hyperscan, using the compilation and runtime tools supplied as part of the MNCaRT ecosystem [44]. The compiler takes MNRL, an open-source state machine representation, as input. We use VASim to convert the ANML emitted by the prototype RAPID compiler, and then use the Hyperscan compiler to generate a serialized *pattern dictionary* and perform Hyperscan-specific optimizations to the automata. We then execute the pattern dictionary against a supplied input stream using the *hsrun* tool provided with MNCaRT.

### 6.3 Targeting GPUs

We support the execution of pure NFAs with a GPU back-end. RAPID programs that do not use counters can therefore be executed on GPUs. We use iNFAnt2, the optimized GPU-based NFA engine used by Wadden et al. with the ANMLZoo benchmark suite [40]. The iNFAnt2 engine reads in a transition table and uses individual SIMD threads to compute possible transitions on a given input symbol.

We use VASim to convert the ANML produced by the RAPID compiler to the transition tables needed by iNFAnt2. Similar to the CPU target, we optimize the ANML using VASim's optimization framework. Next, we output the NFA transition table using the Becchi-style format [25]. To execute on the GPU, we provide both this transition table and an input stream to iNFAnt2, which produces reporting output.

### 6.4 Targeting FPGAs

When targeting an FPGA, we first optimize the compiled automata and then convert to a hardware description using VASim. VASim transforms the optimized NFA into a Verilog hardware description. Our tool generates a module with inputs for clock, reset, and an 8-bit input symbol and outputs for report events. Within the module, activations of states in the automaton are stored in registers, which are updated on every clock cycle. A state becomes *active* if it is enabled (a state with an incident edge to the current state is active) and the current input symbol matches. Using this update rule, it is possible to execute the NFA directly in hardware. Finally, we target Xilinx FPGAs by synthesizing the hardware description produced by VASim. Additional optimization of automata kernel generation for FPGAs using this same technique has been explored by Xie et al. [19].

## 7 Evaluation

We evaluate RAPID against hand-crafted designs for five real-world benchmark applications, which were selected based upon previous research demonstrating significant acceleration using Micron's AP [12], [14], [17], [18].

TABLE 5
Description of benchmarks

| Benchmark | Description | Generation Method | Sample Instance Size |
|---|---|---|---|
| *ARM/FIS* [14] | Association rule mining / Frequent itemset | Python + ANML | 24 Item-Set |
| *Brill* [12] | Rule re-writing for Brill part of speech tagging | Java | 219 Rules |
| *Exact* [18] | Exact match DNA sequence search | Workbench | 25 Base Pairs |
| *Gappy* [18] | DNA string search with gaps between characters | Workbench | 25-bp, Gaps $\leqslant 3$ |
| *MOTOMATA* [17] | Fuzzy matching for bioinformatics planted motif search | Workbench | (17,6) Motifs |

Table 5 provides descriptions of the benchmarks used. For each benchmark, we chose an instance size representative of a real-world problem. These sizes come either directly from previous work or from conversations with the authors of the previous work. The generation method column indicates the technique used to create the hand-crafted code, which ranged from custom Java or Python programs for generating an ANML design to the use of a GUI design tool (Workbench) for crafting automata by hand. The authors of the *ARM* [14] and *Brill* [12] benchmarks provided us with their original code, including a collection of regular expressions for performing the *Brill* benchmark. We recreated the remaining designs, using algorithms and specifications published in previous work.

Table 6 lists design statistics for the benchmarks. We compare the lines of code needed to generate ANML For *ARM*, the RAPID code requires six times fewer lines to represent, and *Brill* requires about half of the lines of the hand-crafted solution. The regular expression representation for *Brill* is more compact than RAPID.

We created the *Gappy*, *Exact*, and *MOTOMATA* benchmarks using a GUI design tool. For these, we present the lines of code in ANML, which is roughly equivalent to the number of actions taken within the design tool. ANML file sizes are dependent on the specific instance of a problem, and the numbers we present are for a single instance of the problem listed in Table 5. In all cases, the RAPID program is significantly more compact than the ANML it generates.

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

Due to the lock-step execution of automata on the AP, runtime performance of loaded designs is linear in the length of a given input stream. Therefore, we focus on evaluating the space efficiency of RAPID programs. In Table 7, we present the performance of RAPID programs

TABLE 6
Comparison between RAPID and hand-crafted code with respect to lines of code (LOC) and STE usage

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

*R – RAPID H – Hand-coded Re – Regular Expression*
† The GUI tool does not have a LOC equivalent metric.
‡ No ANML statistics are provided by the regular expression compiler.

compared to hand-crafted ANML based on placement and routing statistics for the AP, using version 1.4-11 of the AP SDK to generate the placement and routing information. The total blocks column measures the number of routing matrix blocks (see Section 2.2.2) needed to accommodate the design; lower numbers represent a more compact design. STE utilization indicates the percent of used STEs within the routed blocks; high numbers indicate a design with fewer unused STEs. Mean BR allocation (AP MBRA) is a metric provided by the AP SDK that approximates the routing complexity of the design. Here, a lower number is better, signifying lower congestion within the routing matrix. The AP Clk column indicates whether the clock cycle of the AP must be reduced to accommodate a design. In one instance (the RAPID *MOTOMATA* program), the clock cycle must be halved due to a limitation in signal propagation between counters and combinatorial elements in the current generation AP. However, the RAPID version is four times more compact. Although this is a performance loss for a single instance, it is a net performance gain for a full problem, which will fill the AP board: four times as many instances execute in parallel at half the speed, for a net improvement factor of two. Although RAPID provides a higher level of abstraction than ANML, the final device binaries are more compact, using fewer resources on the AP.

We also evaluate the space efficiency of the FPGA engines our tools produce. We synthesize our designs for a Xilinx Kintex UltraScale XCKU060. Table 7 also lists the

TABLE 7
Space utilization on AP and FPGA targets. Lower values for AP States, FPGA LUTs and FPGA Registers indicate a smaller footprint; lower values for AP MBRA indicate less stress on the routing network.

| Benchmark | | AP STEs | AP MBRA | AP Clk | FPGA LUTs | FPGA Reg |
|---|---|---|---|---|---|---|
| *ARM* | H | 58 | 20.8% | 1 | 73 | 76 |
| | R | 56 | 20.8% | 1 | 83 | 65 |
| *Brill* | H | 1,514 | 65.4% | 1 | 201 | 1483 |
| | R | 1,429 | 60.6% | 1 | 358 | 1360 |
| *Exact* | H | 27 | 4.2% | 1 | 6 | 25 |
| | R | 27 | 4.2% | 1 | 28 | 27 |
| *Gappy* | H | 123 | 77.1% | 1 | 73 | 123 |
| | R | 399 | 70.8% | 1 | 52 | 399 |
| *MOTOMATA* | H | 149 | 75.0% | 0.5 | 114 | 148 |
| | R | 72 | 75.0% | 1 | 85 | 60 |

*H – Handcrafted R – RAPID*

number of LUTs and registers needed to implement the hardware description of the benchmark. Lower numbers indicate smaller footprints for the circuits, which allows for more widgets to be run in parallel on the FPGA. As with the AP results, RAPID programs do not incur significant space overheads on the FPGA. A complete timing analysis and comparison with other FPGA engines falls outside the scope of this article, but is examined by Xie et al. [19].

## 8 Conclusions

As data sets continue to grow in size, new hardware and software approaches are needed to quickly process and analyze available data. This article explores the viability of automata processing as an intermediate computational representation to support high-throughput processing across computer architectures. Additionally, we present extended results for RAPID, a language that provides a high-level representation of pattern-matching (automata) algorithms.

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

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

## Acknowledgments

We acknowledge the partial support of the NSF (CCF-0954024, CCF-1629450, CCF-1116289, CCF-1116673, CCF-1619123, CDI-1124931, CNS-1619098); Air Force (FA8750-15-2-0075); Virginia Commonwealth Fellowship; Jefferson Scholars Foundation; Achievement Rewards for College Scientists (ARCS) Foundation; Xilinx; and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We would like to thank Micron Technology and Xilinx for their support and expertise.

## References

[1] Computer Sciences Corporation, "Big data universe beginning to explode," http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode, 2012.
[2] Capgemini, "Big & fast data : The rise of insight-driven business," http://www.capgemini.com/resource-file-access/resource/pdf/big_fast_data_the_rise_of_insight-driven_business-report.pdf, 2015.
[3] Titan IC Systems, "RXP regular eXpression processor soft IP," http://titanicsystems.com/Products/Regular-eXpression-Processor-(RXP).
[4] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware acceleration in the IBM PowerEN processor: Architecture and performance," in *International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 389–400.
[5] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.
[6] Intel, "Hyperscan," https://01.org/hyperscan, 2017, accessed 2017-04-07.
[7] J. Wadden and K. Skadron, "VASim: An open virtual automata simulator for automata processing application and architecture research," University of Virginia, Tech. Rep. CS2016-03, 2016.
[8] I. Sourdis, J. Bispo, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching in reconfigurable hardware," *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 99–121, 2008.
[9] X. Wang, "Techniques for efficient regular expression matching across hardware architectures," Master's thesis, University of Missouri-Columbia, 2014.
[10] K. Angstadt, W. Weimer, and K. Skadron, "Rapid programming of pattern-recognition processors," in *Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 593–605.
[11] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *International Symposium on Microarchitecture*, 2015, pp. 533–545.
[12] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the Micron Automata Processor," in *International Conference on Semantic Computing*, 2015, pp. 236–239.
[13] M. H. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda, "Using the Automata Processor for fast pattern recognition in high energy physics experiments - a proof of concept," *Nuclear Instruments and Methods in Physics Research*, vol. 832, pp. 219–230, 2016.
[14] K. Wang, M. Stan, and K. Skadron, "Association rule mining with the Micron Automata Processor," in *International Parallel & Distributed Processing Symposium*, 2015.
[15] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the Micron Automata Processor," in *International Conference on Computing Frontiers*, 2016, pp. 135–144.
[16] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the Automata Processor," in *Proceedings of ISC High Performance Computing*, 2016, pp. 200–218.
[17] I. Roy and S. Aluru, "Finding motifs in biological sequences using the Micron Automata Processor," in *International Parallel and Distributed Processing Symposium*, 2014, pp. 415–424.
[18] C. Bo, K. Wang, Y. Qi, and K. Skadron, "String kernel testing acceleration using the Micron Automata Processor," in *Workshop on Computer Architecture for Machine Learning*, 2015.
[19] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. R. Stan, "REAPR: Reconfigurable engine for automata processing," in *the International Conference on Field-Programmable Logic and Applications*, 2017.
[20] M. Sipser, *Introduction to the Theory of Computation*. Thomson Course Technology, 2006, vol. 2.
[21] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proceedings of Architectures for Networking and Communications Systems*, 2008, pp. 50–59.
[22] Y. Kaneta, S. Yoshizawa, S. Minato, and H. Arimura, "High-Speed String and Regular Expression Matching on FPGA," in *Proceedings of the Asia-Pacific Signal and Information Processing Association*, 2011.

[23] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 227–238.
[24] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact Architecture for High-throughput Regular Expression Matching on FPGA," in *Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 30–39.
[25] M. Becchi, "Regular expression processor," http://regex.wustl.edu, 2011, accessed 2017-04-06.
[26] Micron Technoloy, "Calculating Hamming distance," http://www.micronautomata.com/documentation/cookbook/c_hamming_distance.html.
[27] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in *Principles of Programming Languages*, 2005, pp. 98–109.
[28] E. Spishak, W. Dietl, and M. D. Ernst, "A type system for regular expressions," in *Workshop on Formal Techniques for Java-like Programs*, 2012, pp. 20–26.
[29] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*, 2002, pp. 179–196.
[30] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.
[31] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf, "Alma-0: An imperative language that supports declarative programming," Tech. Rep., 1997.
[32] P. J. L. Wallis, "The design of a portable programming language," *Acta Informatica*, vol. 10, no. 2, pp. 157–167, Jun 1978.
[33] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.
[34] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, "Map-reduce as a programming model for custom computing machines," in *International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 149–159.
[35] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: Writing parallel program portable between cpu and gpu," in *Parallel Architectures and Compilation Techniques*, 2010, pp. 217–226.
[36] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 35:1–35:12.
[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *International Symposium on Workload Characterization*, Oct 2009, pp. 44–54.
[38] J. Wadden, K. Angstadt, and K. Skadron, "Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures," in *High Performance Computer Architecture*, Feb 2018, pp. 749–761.
[39] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 517–528.
[40] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *International Symposium on Workload Characterization*, Sept 2016, pp. 1–12.
[41] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying automata processing: Gpus, fpgas or micron's ap?" in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1:1–1:11.
[42] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity resolution acceleration using the automata processor," in *International Conference on Big Data*, Dec 2016, pp. 311–318.
[43] I. Roy, "Algorithmic techniques for the micron automata processor," Ph.D. dissertation, Georgia Institute of Technology, 2015.
[44] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron, "MNCaRT: An open-source, multi-architecture automata-processing research and execution ecosystem," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 84–87, Jan 2018.
[45] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216.
[46] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, "High performance pattern matching using the automata processor," in *International Parallel and Distributed Processing Symposium*, 2016, pp. 1123–1132.
[47] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.
[48] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *SPIN Workshop on Model Checking of Software*, 2001, pp. 103–122.

**Kevin Angstadt** received a BS in computer science, mathematics, and German studies from St. Lawrence University in 2014 and a Masters degree in computer science from the University of Virginia in 2016. He is now a PhD student at the University of Michigan. His research focuses on improving programming support for emerging hardware technologies, including both the development of new programming models as well as automated techniques for adapting existing software.

**Jack Wadden** is a post-doctoral fellow at the University of Michigan. Jack received his BA from Williams College in 2011 and a PhD from the University of Virginia in 2018. He studies application specific accelerators, with a focus on spatial-reconfigurable computing, automata processing, and genomics, and is also interested in software/hardware co-design for architectural reliability.

**Westley Weimer** received a BA degree in computer science and mathematics from Cornell University and MS and PhD degrees from the University of California, Berkeley. He is currently a full professor at the University of Michigan. His main research interests include static and dynamic analyses to improve software quality and fix defects, as well as medical imaging and human studies of programming.

**Kevin Skadron** is the Harry Douglas Forsyth professor and chair of the Department of Computer Science at the University of Virginia, where he has been on the faculty since 1999. His research focuses on heterogeneous architecture, design and applications of novel hardware accelerators, and design for physical constraints such as power, temperature, and reliability. Skadron and his colleagues have developed a number of open-source tools to support this research, including the HotSpot thermal model, the Rodinia GPU benchmark suite, the ANMLZoo automata benchmark suite, the MNCaRT automata processing design framework, and the RAPID programming language. Skadron is a Fellow of the IEEE and the ACM, and recipient of the 2011 ACM SIGARCH Maurice Wilkes Award.