An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing

Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes, *Member, IEEE*

Abstract—We present the design and development of the automata processor, a massively parallel non-von Neumann semiconductor architecture that is purpose-built for automata processing. This architecture can directly implement non-deterministic finite automata in hardware and can be used to implement complex regular expressions, as well as other types of automata which cannot be expressed as regular expressions. We demonstrate that this architecture exceeds the capabilities of high-performance FPGA-based implementations of regular expression processors. We report on the development of an XML-based language for describing automata for easy compilation targeted to the hardware. The automata processor can be effectively utilized in a diverse array of applications driven by pattern matching, such as cyber security and computational biology.

Index Terms—Automata, parallel architectures, high performance computing, hardware, accelerator architectures, reconfigurable architectures

1 Introduction

Efficient implementation of automata-based processing has been studied for several decades and has been applied to diverse fields such as network security, computational biology, image processing, and text search [1]. Research has been primarily focused on the use of traditional CPU architectures as the engine for automata processing [2]. In the last few years, this work has been expanded to include multi-core CPUs, network processors, and GPUs (for example, [3], [4], [5], [6]).

Software-based automata usually employs a DFA-based approach over NFA, e.g., see Liu *et al.* [7], trading speed for relatively plentiful memory, though good results using simulated NFA have been reported in some situations [8]. Various techniques to improve the performance of automata processing on von Neumann architectures have been explored. Refinement of DFA [9] or reduction of acuity in pattern matching capability by restricting constructs such as bounded quantifications, Kleene stars and logical operations [10] may reduce the problem of state explosion. Modified NFA, which reduce the problem of backtracking, and hybrid finite automata, have also been proposed ([6], [9], [11], [12], [13]). In von Neumann software-based automata processing systems, DFA and NFA can be thought of as extremities of a spectrum of space and time complexity tradeoffs, as illustrated in Table 1 [14].

Direct implementation of automata in hardware has the potential to be more efficient than software executing on a von Neumann architecture. Hardware-based automata can effect simultaneous, parallel exploration of all possible valid paths in an NFA, thereby achieving the processing complexity of a DFA without being subject to DFA state explosion. FPGAs offer, to some extent, the requisite parallelism and a number of relatively recent efforts have explored this direction ([15], [16], [17], [18], [19]). However, there remain significant limitations in FPGA-based implementations.

We have created an architecture purpose-built for direct implementation of NFA which achieves significantly improved processing efficiency, capacity, expressiveness and computational power. We also expect dramatic improvements in cost and power consumption compared to other approaches from the silicon implementation, in fabrication at the time of writing.

We present the theoretical model of the architecture in relation to the theory of bit parallelism in Section 2 and its semiconductor implementation in Section 3. In Section 4, we evaluate the architecture's ability to implement complex regular expressions and other types of automata in a time- and space-efficient manner. An overview of related work is presented in Section 5. We conclude in Section 6 with a discussion on future work on the architecture. Additional detail on the semiconductor implementation, a more extensive comparison to related work, and videos of example automata presented in the paper is included in supplementary material which is available in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.8.

2 Automata Processor Execution Model

We make use of the formal definition of non-deterministic finite automata, extending its execution model to account for the unique properties of the automata processor. An NFA is described by the 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$where Q is

• *The authors are with Micron Technology, DRAM Solutions Group, Architecture Development Group, Boise, ID, USA. E-mail: {pddlugosch, dbrown, pglendenning, mleventhal, hnoyes}@micron.com.*

Manuscript received 29 Aug. 2013; revised 29 Oct. 2013; accepted 31 Oct. 2013. Date of publication 21 Jan. 2014; date of current version 14 Nov. 2014. Recommended for acceptance by S. Aluru.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.8

TABLE 1
Storage Complexity and per Symbol Processing Complexity of an n-State NFA and Equivalent DFA

	Processing Complexity	Storage Cost
NFA	$O(n^2)$	$O(n)$
DFA	$O(1)$	$O(\Sigma^n)$

the set of automaton states, Σ is the alphabet, δ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of final states. The transition function $\delta(q, \alpha)$ defines the set of states that can be reached from state q when the symbol α is received at the automaton input.

An automaton can be modeled as a labeled graph, where the vertices are labeled with the states and the edges are labeled with the symbols of the transition function.

The transition function can be extended to multiple states in any class $C \subseteq Q$, by the following definition:

$$\delta(C, \alpha) = \bigcup_{q \in C} \delta(q, \alpha).$$

The closure of the class C is the set of states that are reached from any member of C, on any input symbol. The closure is defined as

$$\delta(C) = \bigcup_{\alpha \in \Sigma} \bigcup_{q \in C} \delta(q, \alpha).$$

A homogeneous automaton is a restriction on the automaton definition above, such that all transitions entering a state must occur on the same input symbol(s), i.e. for any two symbols $(\alpha, \beta) \in \Sigma$ and any two states $(p, q) \in Q$, then $\delta(p, \alpha) \cap \delta(q, \beta) = \delta(q, \alpha) \cap \delta(p, \beta)$. This definition has its roots in Gluskov's position automata [20]. The position automaton obtained from an $n - length$ regular expression (excluding operators) is an $(n + 1)$ state homogeneous automaton. Each unique symbol and position pair (or character class), within the regular expression, maps to a homogeneous state in Q.

We now introduce the concept of symbol acceptance. A homogeneous automaton state q is said to accept the symbol α, if α labels a transition entering the state, or is a subset of any transition label entering the state. Formally, we define the symbol acceptance function as $symbols(\alpha) = \cup_{q \in Q} \delta(q, \alpha)$, i.e., state q accepts symbol α iff $q \in symbols(\alpha)$. We can now define the transition function as:

$$\delta(C, \alpha) = \delta(C) \cap symbols(\alpha).$$

A sequence of alphabet symbols, on the input of the homogeneous automaton, will cause the automaton to transition through a path of states in Q. The automaton runtime state $C \subseteq Q$ is defined as the states that will potentially transition on the next input symbol. $T = C \cap symbols(\alpha)$ are the states that accept the next input symbol. The automaton is said to match the sequence, presented to its input, when $T \cap F \neq \emptyset$. The automaton execution model for an input string S is shown below.

```
1: C = δ(q0)
2: if q0 ∈ F then
3:   match the empty string
4: end if
5: for each input character α in S do
6:   T = C ∩ symbols(α)
7:   if T ∩ F ≠ ∅ then
8:     we have a match
9:   end if
10:  if T is empty then
11:    stop processing S
12:  end if
13:  C = δ(T)
14: end for
```

The runtime complexity of the execution model depends on how efficiently we can execute lines 6-13. A DFA is a special case where $|T| \leq 1$ and $|\delta(T)| \leq |\Sigma|$, however the space cost $|Q|$ can be exponential, in the worst case.

We now describe the theoretical foundation of the automata processor in relation to bit parallelism. Bit-parallelism and its application to string matching was first introduced by Richard L. Baeza-Yates [21]. The general concept is to encode states as an m-bit word. Typically m is chosen to be a multiple of the machine word size of the CPU executing the bit-parallel algorithm. For this reason bit-parallelism is usually reserved for small m-values. The bit parallel homogeneous automaton is defined by the 5-tuple $\langle 2^Q, \Sigma, \Delta, 2^I, 2^F \rangle$. 2^Q is an m-bit word, where each bit position represents a state in $Q = \{0, \ldots, m-1\}$. $2^Q = |_{q \in Q} 2^q$ and the $|$ operation, over all Q, is equivalent to a bitwise OR. $2^I = |_{q \in \delta(0)} 2^q$ is an m-bit word, where each bit position represents the set of initial states reachable from the start state $q_0 = 0$. $2^F = |_{q \in F} 2^q$ is an m-bit word, where each bit position represents a final state. Δ is the transition function such that $\Delta(2^q, \alpha)$ is an m-bit word, where each bit represents the state bits reachable from bit position 2^q on symbol α.

The bit parallel closure on a state q is defined as:

$$\Delta(2^q) = \bigm|_{\alpha \in \Sigma} \Delta(2^q, \alpha).$$

For small m, the transition function for a bit parallel homogeneous automata can be implemented efficiently in O(1) time by the bitwise AND (designated as operator &) of two lookups.

$$\Delta(2^q, \alpha) \equiv follow[q] \ \& \ symbols[\alpha]$$

where $follow[q] = \Delta(2^q)$ and $symbols[\alpha] = |_{q \in Q} \Delta(2^q, \alpha)$

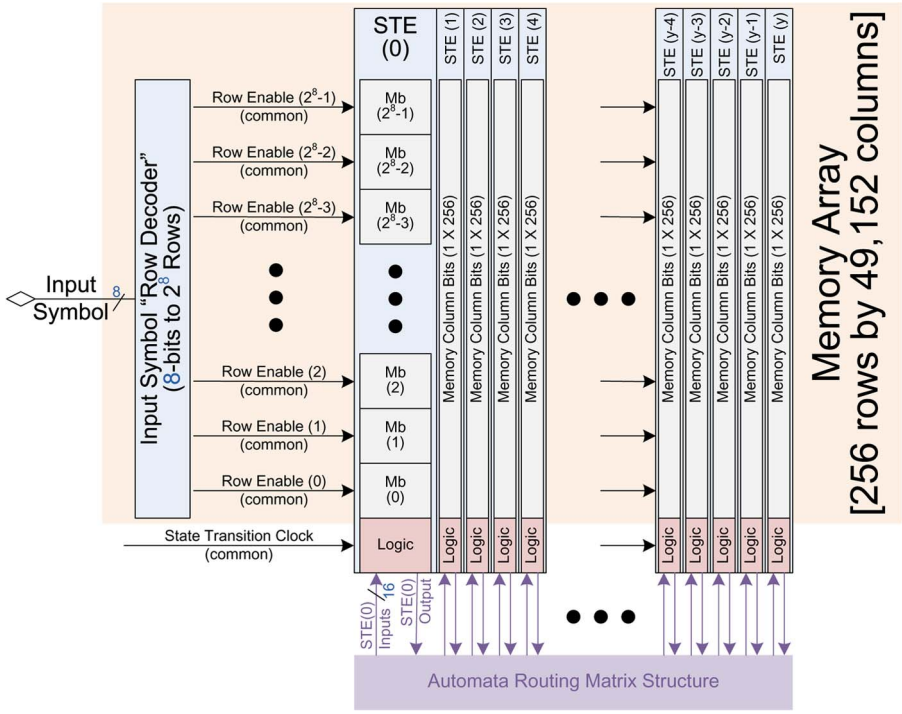

Fig. 1. Memory array.

One benefit of the lookup method above is that it inherently supports character classes. The bit-parallel execution model for an input string S is shown below.

1: $2^C = 2^I$
2: **if** 2^F & $0 \times 1 \neq 0$ **then**
3: match the empty string
4: **end if**
5: **for** each input character α in S **do**
6: $2^T = 2^C$ & $symbols[\alpha]$
7: **if** 2^T $2^F \neq 0$ **then**
8: we have a match
9: **end if**
10: Set $2^C = 0,\ \forall_q \in T,\ 2^C = 2^C | follow[q]$
11: **if** $2^C = 0$ **then**
12: stop processing S
13: **end if**
14: **end for**

Line 2 tests if the start state 2^0 is a member of the final states 2^F. We include this to provide a direct comparison with the traditional NFA; however, we do not do this in hardware since it has no practical value. Lines 6-9 can be executed in (m/w) time, where w is the machine word size implementing the execution model above. Navarro and Raffinot [22] describe a method, using k tables, to implement line 6 in (mk/w) time. The size of k, in Navarro's algorithm, depends on the space complexity of the equivalent DFA, since each table must store $O(2^{m+1/k})$ entries, in the worst case. We have developed, to our knowledge, the first practical method implementing the bit-parallel execution model described above for large $m = 48$ k. The practical method used to achieve this is described in the next section on the Architectural Design.

3 Architectural Design

3.1 A Memory-Derived Architecture

The automata processor is based on an adaptation of memory array architecture, exploiting the inherent bit-parallelism of traditional SDRAM. Conventional SDRAM, organized into a two-dimensional array of rows and columns, accesses a memory cell for any read or write operation using both a row address and a column address. The ''row address'', for the automata processor, is the input symbol. The 8-bit input symbol is decoded (8-to-256 decode) and then provided to the memory array. In place of memory's column address and decode operation, the automata processor invokes automata operations through its routing matrix structure. The memory array portion of the architecture is illustrated in Fig. 1.

The architecture provides the ability to program independent automata into a single silicon device. Each automaton and all automata routing matrix paths run in parallel, operating on the same input symbol simultaneously. Memory arrays are distributed throughout the silicon, providing O(1) lookup for a m = 48 K bit memory word. This first implementation, derived from Microns DDR3 SDRAM memory array technology, has an 8-bit DDR3 bus interface. It is capable of processing 8-bit input symbols at 1 Gbps, per chip.

3.2 The Routing Matrix

The routing matrix controls the distribution of signals to and from the automata elements, as programmed by the

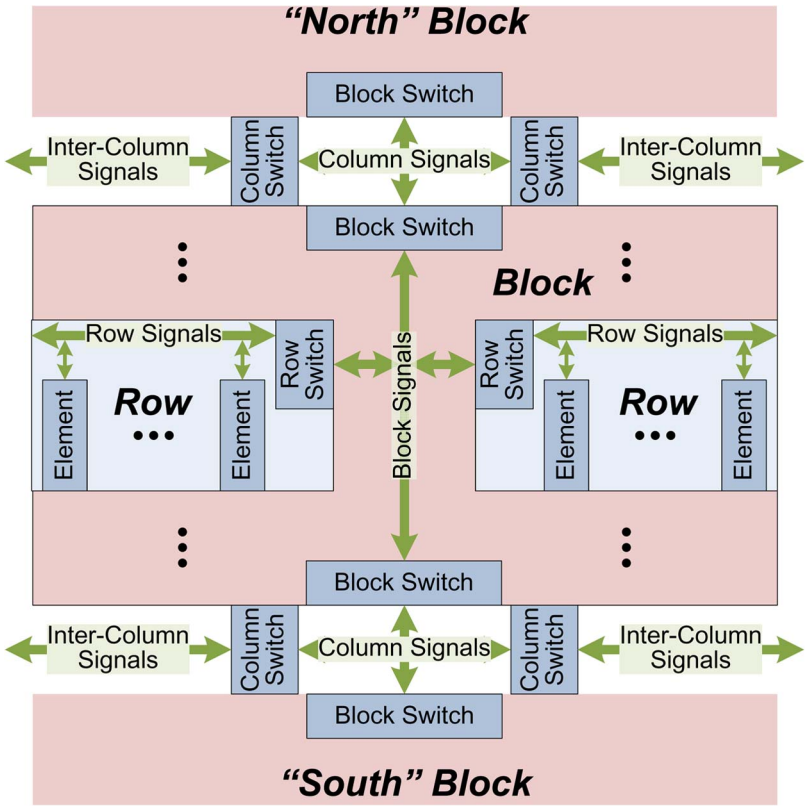

Fig. 2. Routing matrix.

application. The routing matrix is a complex lattice of programmable switches, buffers, routing lines, and cross-point connections. While in an ideal theoretical model of automata every element can potentially be connected to every other element, the actual physical implementation in silicon imposes routing capacity limits related to tradeoffs in clock rate, propagation delay, die size, and power consumption. From our initial design, deeply informed by our experience in memory architectures, we progressively refined our model of connectivity until we were satisfied that the target automata could be compiled to the fabric and that our performance objectives would be met.

The routing matrix is a hierarchical structure of groups of elements, with switches controlling the interconnection between the levels of the hierarchy. This is illustrated in Fig. 2.

A number of the various elements are grouped together into rows, rows are grouped into blocks, and blocks are laid out in a grid of block rows and block columns. The routing matrix provides the interconnections at the various levels of this hierarchy: within rows, within blocks, and within the grid of blocks. A summary of these routing matrix signals and their respective functions is given in Table 2.

The transitions between the different signal groups are controlled through the programming of different switch buffers. These switch buffers are bi-directional, tri-stateable, multiplexing buffers. A given signal can be selectively connected to several different inputs of adjacent levels of the hierarchy.

The maximum number of transitions, from a single state, at maximum clock frequency is 2304, corresponding to 256 states in each of 9 blocks arranged in an 8 point compass. A larger fan-out is achievable at a slower clock rate, albeit at the cost of reduced performance. While any state can have the maximum number of transitions, only 24 states total out of the 256 in a block can have that many transitions. The potential fan-out from any state is otherwise 16. Additional information regarding fan-out, fan-in, and other routing complexities may be found in the paper's supplementary material available online. The implemented selective and programmable connectivity has been sized and load-tested to have sufficient connectivity for representative classes of target automata and in practice has proven reasonably robust. The design does not preclude, of course, routing congestion issues and/or inability to route, especially with highly connected graph-type automata.

The routing matrix controls signal routing; the elements implement the various logical functions needed to

TABLE 2
Routing Matrix Signals

Signal Name	Function
row	Connections to all elements within a given row.
block	Connections to rows within a given block.
column	Interconnections of blocks in one column, in North and South directions
inter-column	Interconnection of block columns, in East and West directions

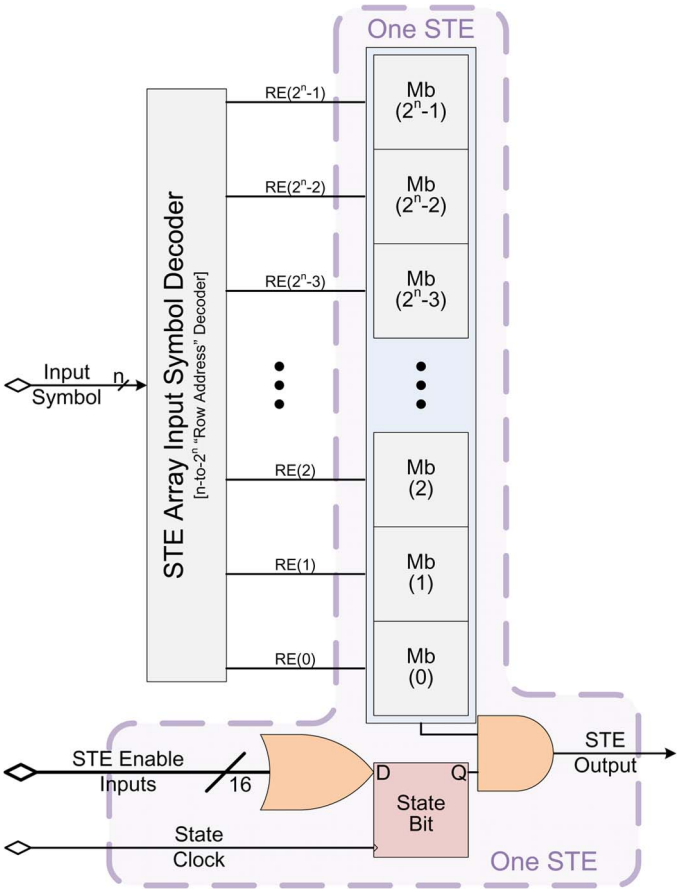

Fig. 3. State transition element memory array.

compute the automata. Programming the routing matrix and the elements actually implements the desired automata.

3.3 The Automata Processor Elements

The various functional components, called elements, are woven into the hierarchy of the routing matrix.

All elements share four features:

1. Each receives sixteen inputs from the routing matrix.
2. Each performs a function within an automaton.
3. Each is programmable.
4. Each originates (drives) one signal back into the routing matrix.

The number of sixteen inputs arises as a consequence of the design of the routing matrix, described above, including consideration of physical feasibility and experimentation with target automata. The layout of automata may be modified if the number of inputs exceed 16 during compilation. We transfer the in-degree congestion to out-degree by state splitting. Any state can be split into two where the inputs are partitioned between the split pair.

The state transition element is at the heart of the design and is the element with the highest population density, having a one-to-one correspondence to the state bits of a bit-parallel automata model described in the previous section. Counters and boolean elements are used with state transition elements to increase the space efficiency of

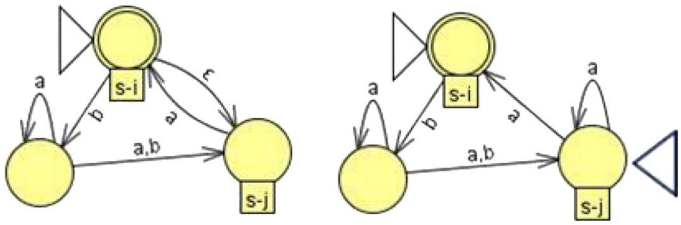

Fig. 4. Epsilon NFA.

automata implementations, as a design convenience, and to extend computational capabilities beyond NFA.

3.3.1 The State Transition Element

The state transition element consists of the current state memory and the next state decoder. In terms of classic hardware state machine design, it can be associated with the next state transition function. State transition elements are implemented as a memory array with control and computational logic. Each column of the memory array includes logic that contains a single state bit, enable inputs, and an output decoder/driver. The output is determined by the logical AND of the state bit and the output of the associated column of memory. Each state bit is either set (1) or reset (0), depending on whether that state transition element is in an active or inactive state. A conceptual model is shown in Fig. 3.

The height of this memory column is determined by the number of bits (n) in the input symbol. For example, for a byte-wide (8-bit) input symbol, each state transition element must have a column of $2^8 = 256$ bits of memory.

State bits are pre-loaded prior to processing any input symbols. This makes it possible for the initial state of every state transition element to be either active or inactive. Any state transition element and any number of state transition elements may therefore be a start state. This capability allows independent automata to each have their own start state and also permits individual automata to have multiple start states. This allows additional flexibility in the design compared to conventional automata, which are limited to a single start state. For example, ϵNFAs can be implemented directly with ease, using the automata processor. The ϵNFA on the left side in Fig. 4 with start state $s-i$ (indicated by the triangle) and an epsilon transition from $s-i$ to $s-j$ can be realized directly in the automata processor by also making $s-j$ a start-enabled state transition element, as shown on the right side of Fig. 4.

The 256 bits of symbol recognition memory can be thought of as a single column of 256 rows of memory. The input symbol is analogous to a memory row address, which is decoded (an 8-to-256 decode), to select one of the 256 memory cells of the symbol recognition memory. If the selected bit was programmed to recognize the input symbol, the symbol recognition memory outputs a 1. If it is not programmed to recognize the input symbol, the symbol recognition memory outputs a 0. An important consequence of this design is that it allows any subset of the all possible 8-bit symbols (2^{256} combinations) to be programmed to match. This provides the ability to handle full character classes in every state transition element.

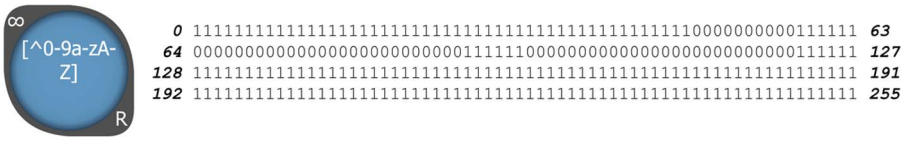

Fig. 5. Character class.

A simple example of a state transition element recognizing a character class is shown in Fig. 5. The character class in the example is any value which is not an ascii upper or lower case letter or a numeric character. The state transition element on the left is start-enabled, receiving every input byte, and reports on a match (the former indicated by the ∞ symbol in the upper left and the latter by the R in the lower right). This single state transition element could be a complete machine for reporting when unexpected values are found in an input stream. The encoding of the character class is shown on the right side of Fig. 5. Each of the 256 bits which is not 0-9a-zA-Z is set. Each input symbol is decoded from 8 to 256 bits and compared against all set character class bits to determine if there is a match.

3.3.2 The Counter Element

The counter element is a 12-bit binary counter. Every time one of the count enable signals are asserted the counter counts by one. When the counter's count equals the target value, an output event is triggered.

The counter has also implemented several features that provide greater flexibility in designing various automata. These features are:

1. the ability to cascade up to four counters, to achieve up to a 48-bit counter;
2. output and reload mode control (pulse, continuous, or pulse-and-reload);
3. an over-riding synchronous reset function;
4. the ability to choose different row signal inputs, for both the count and reset functions.

Counters can be an efficient way to count subexpressions, such as those that occur routinely in applications using regular expressions with quantifications. Fig. 6 shows the use of two counters to implement a range quantification. The automaton implements the regular expression /#[0-9]{500,999}#/. The state transition element on the left receives every input symbol (indicated by the ∞ symbol in the upper left). If it recognizes a # symbol, recognition of digits is begun by activation of the state transition element to its left. This self-looping state transition element remains asserted as long as digits continue to be recognized. This element also activates the two counters and the state transition element below it. The upper of the two counters counts the minimum range value of 500 and the lower counts the maximum range value of 999. If a non-digit is seen in the input stream, the lower state transition element will reset both counters (indicated by the connection to the R terminal at the lower left side of the counters) and recognition of the current input sequence will terminate. Each time a digit is received, the count advances in both counters (indicated by the connection to the C terminal on the upper left side of the counters). When the upper counter counts 500, it will continously activate (indicated by the clock edge symbol on the right side of the counter) the following state transition element. This element will report recognition of the sequence (indicated by the R in the lower right corner) if a terminating # is received. The lower counter enforces the maximum range value by resetting the upper counter, once the maximum range value is exceeded, causing activation of the final state transition element to cease and with it the ability for the element to report on receiving the terminating #.

Counters are also a type of restricted memory device, enabling creation of non-deterministic counter automata. Counter automata can implement proper subsets of pushdown automata and, therefore, some context-free languages [23] and have been theoretically demonstrated to even be capable of implementing Turing machines [24]. We have been able to construct many practical automata using counters as a scratchpad within NFA. A simple example is the use of the counter to prune paths and prevent looping through cycles as shown in Fig. 7. The three connected state transition elements in the example form a cycle. While the example is designed to be self-contained, the principle can be extended to any graph with multiple cycles. The left-most state transition element is the start of the cycle, receiving the first input symbol (indicated by the 1 symbol in the upper left). The counter prevents the

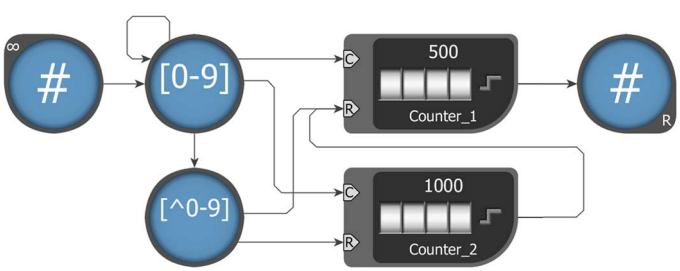

Fig. 6. Counter element example.

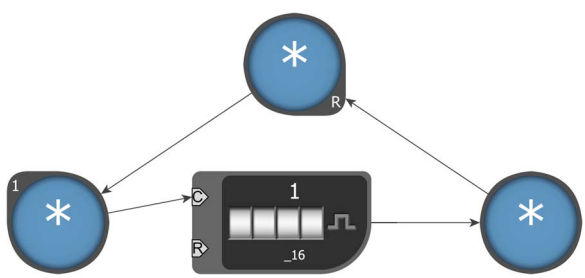

Fig. 7. Pruning with counters example.

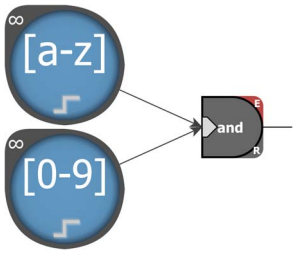

Fig. 8. Boolean element example.

cycle from being traversed more than once. It is configured as a one-shot, that is, it is set to count to 1. The first time it counts it activates the next state transition element, and, without a reset, stays dormant (this is the behavior of pulse mode, the setting indicated by the clock edge symbol on right side of the counter). Subsequent activations will not result in an output activation to the following state transition element. This behavior allows the cycle to be traversed only once.

3.3.3 The Boolean Element

The boolean element is a function-programmable combinatorial element, with an optional synchronized enable. The boolean element can be programmed to perform the following logical functions: OR, AND, NAND, NOR, sum of products, and products of sums. Booleans have no state (and do not use memory), unlike the state transition and counter elements.

Boolean elements are routinely used to simplify designs, where a combination of the results of sub-expressions is required.

The synchronized enable is triggered by a special signal propagated simultaneously to all boolean elements. This signal is controlled by instructions to the automata processor and can occur at any desired position(s) in the symbol input data set, including at the end of the symbol set. When the synchronized enable is used, booleans introduce positional dependence to the operation of the automata, since the boolean only computes its combinatorial function when the special signal is asserted. One use of this feature is to implement a construct commonly used in regular expressions—evaluation only on end-of-data (right-anchored expressions). More general usage with automata includes gating data into chunks sized for specific automata and reduction or aggregation of results. This capability of the boolean element introduces a feature beyond the formal definition of NFA, adding a dynamic aspect to automata processing.

An example of the boolean element combining sub-expressions with use of the synchronized enable is illustrated in Fig. 8. The automata reports, at assertion of the synchronized enable, if the input stream up to that point contained both a lower case ascii letter and an ascii digit. The state transition elements on the left each receive all input symbols (indicated by the ∞ symbol in the upper left corner), the top one checking for the letter, the bottom for the digit. Each state transition element is latched (indicated by the clock edge symbol at the bottom of the state transition element symbol), meaning that if it matches it will continue to assert until reset. The two state transition elements are combined in an AND, effective on assertion of the synchronization enable (indicated by the E symbol in the upper right corner). If the AND generates a high value it will report that at least one letter and one digit were seen in the input stream (indicated by the R in the lower right corner).

3.4 Reconfigurability

The automata processor is a programmable and reconfigurable device. The element arrays are fixed but the operations of individual elements and the connections between them are programmable. The automata processor is also partially dynamically reconfigurable, as the operation of elements can be reprogrammed during runtime. The connections between elements are optimized for resource utilization and require placement and routing analysis, done in a more time-consuming compilation phase. However, once place-and-route has been done for the automata, that structure may be incrementally loaded dynamically, alongside existing automata, to a chip with unused capacity.

3.5 Intra-Rank Bus: scaling Performance and Capacity

The automata processor architecture includes an intra-rank bus. It enables symbols to be distributed to a connected set (rank) of automata processor chips, allowing expansion of both the capacity and the performance of automata processor systems. The intra-rank bus allows a range of configurations. For example, in a 64-bit system that has eight automata processor chips in a rank, the intra-rank bus allows configurations with up to eight times the automata capacity or eight times the processing throughput of a single chip.

4 Evaluation

At the time of this writing, a chip design for the architecture has been completed in DRAM process technology and is currently in fabrication. An SDK has been developed which supports configuration of the chip with automata designs and runtime control of sets of automata processors. Evaluation of the architecture is limited to the results from compilation of automata and simulation.

The automata processor may be configured with automata using either a list of regular expressions in PCRE syntax or a direct description of the automata in an XML-based high-level language that we have created called the Automata Network Markup Language (ANML). Evaluation of the architecture is discussed for each type of input, PCRE and ANML.

4.1 Configuration by PCRE

The automata processor has been designed to have a high degree of compatibility with PCRE. Features of PCRE which exceed the computational power of regular languages, such as lookahead and lookbehind assertions and backreferences, cannot be implemented using NFA alone

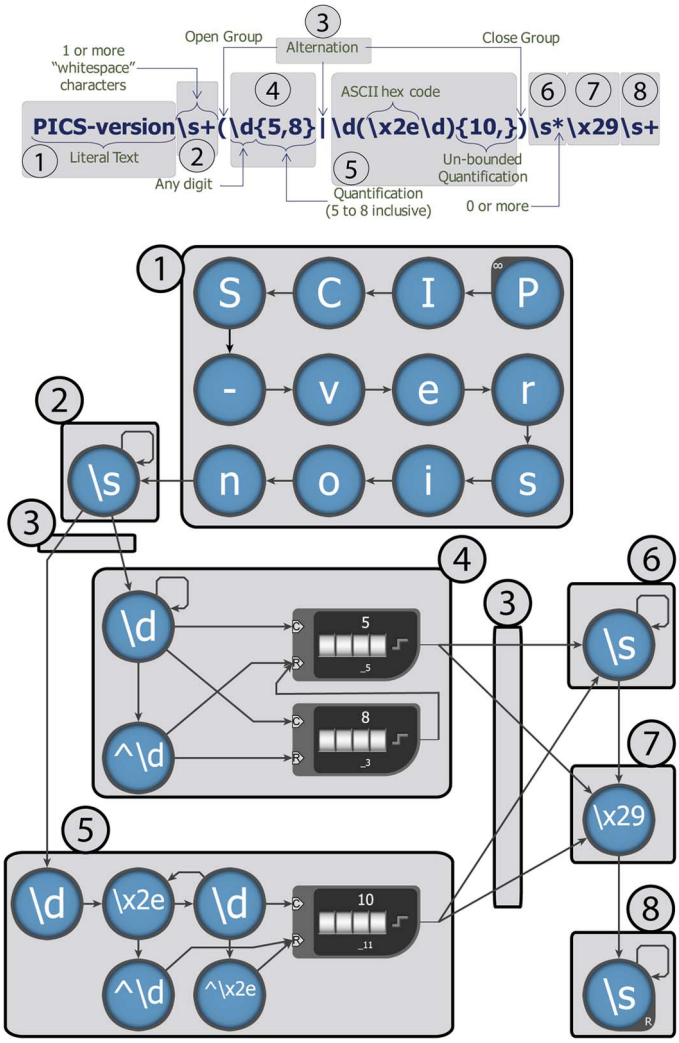

Fig. 9. Snort PCRE Rule as automata.

(see [14]). While counters raise the computational power of the automata processor beyond pure NFA, PCRE expressions using the aforementioned constructs must still be postprocessed in software. The software imposes a small number of restrictions on these constructs in order to ensure that effective use is made of the hardware. This design allows the automata processor to represent PCREs of any complexity as compact and efficient automata. Fig. 9 provides an example of a complex rule, taken from Snort [25], and shows how this is converted to an automaton that runs directly on the automata processor. This particular rule is designed to capture a buffer overflow attack on an Apache web server. A video showing operation of the automaton in a visual simulation tool has been included in the paper's supplementary material, available online.

We have run compilation and simulation tests from [26], in which Becchi and Yu assembled large and complex rule sets designed to test the capabilities of wide range of regular expression engines. The tests were extracted from Snort and modified to increase the number of patterns including character classes and to increase the percentage of patterns using unbounded repetitions of wildcards. We report the number of regular expressions in the dataset, the

TABLE 3
PCRE Compilation Results

Ruleset	Description	Num. of Regex	NFA States	STEs Used	% chip used
Backdoor	real	226	4.3k	4.5k	13
Spyware	real	462	7.7k	7.8k	23
EM	only exact match patterns	1k	28.7k	29.7k	78
Range.5	50% of patterns have char-classes	1k	28.5k	29.3k	78
Range1	100% of patterns have char-classes	1k	29.6k	30.4k	80
Dotstar.0.5	5% of patterns have *	1k	29.1k	30.0k	77
Dotstar.1	10% *	1k	29.2k	30.0k	77
Dotstar.2	20% *	1k	28.7k	29.6k	77
Dotstar.3	30% *	1k	unavail	29.3k	76

number of NFA states needed to implement the datasets evaluated by Becchi and Yu [26], the number of state transition elements used after configuration of the chip by our SDK's compiler, and the percentage of chip capacity that represents (Table 3).

The results show that the usage of state transition elements corresponds nearly 1-to-1 with the number of NFA states and that resource utilization does not grow with expression complexity. All rule sets will fit in a single automata processor chip and will compute results at exactly 1 Gbps per chip. A rank of 8 chips configured as 8 groups would run at 8 Gbps and further scaling can be obtained by multiplying ranks.

4.2 Configuration by ANML

Our architecture, when configured through ANML, provides, as far as we know, the first hardware implementation to allow direct programming of automata structures. While an extended discussion of ANML applications is beyond the scope of this paper, we provide here two abbreviated examples which show how ANML can be used for problems that cannot be readily formulated using regular expressions. Videos showing operation of both automata in a visual simulation tool has been included in the paper's supplementary material available online.

Our first abbreviated example reports when a sequence contains one or more a symbols, followed by one or more b symbols, followed by one or more c symbols, and the total number of symbols in the sequence is equal to 17. A regex solution requires that every possible combination of symbols a, b and c be enumerated. This problem can be solved directly with the following simple ANML machine (Fig. 10).

The starting element (with the all-input ∞ symbol) will begin counting once an a is received. The upper three state transition elements recognize one or more of symbol a followed by one or more of symbol b followed by one more of symbol c and each time an a, b or c is recognized the count advances. If something other than an a, b, or c is received, the state transition elements implementing the

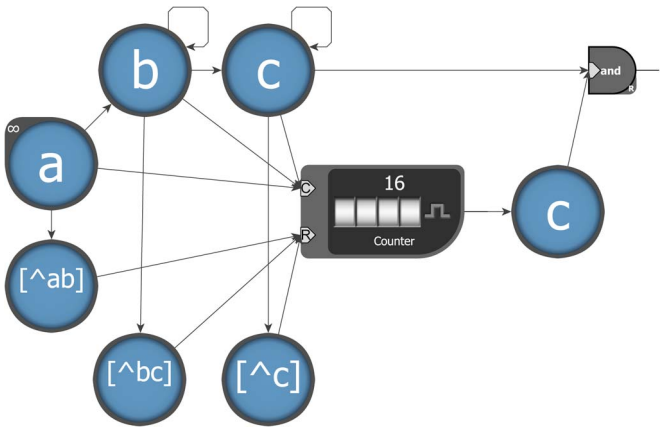

Fig. 10. ANML pattern match example.

negated character classes $[^{\wedge}ab]$, $[^{\wedge}bc]$ and $[^{\wedge}c]$ will reset the counter, restarting the sequence. When 16 of a, b or c have been counted the state transition element connected to the counter output is activated. If the seventeenth symbol is a c, both the lower and upper state transitions elements will input into the AND element causing the AND to report that the input sequence conforms to the pattern.

The second example illustrates how graph data structures can be implemented directly in ANML using the parallel evaluation and activation of automata processor elements to naturally perform a breadth-first descent of the tree. The graphic representation of a small ANML tree is shown in Fig. 11, where each square represents a tree node consisting of several state transition elements which have the task of reporting when a 4-byte search key matches the 4-byte value stored in the node. The search key is broadcast n times, where n is the depth of the tree. Each broadcast takes 4 symbol cycles. In the worst case, all nodes matching the search key will be identified in $4n$ symbol cycles.

The contents of an internal tree node are shown in Fig. 12. The small pentagonal shapes labeled I and O are macro ports, i.e., connection points into and out of the macro structure used to encapsulate a node. The state transition elements on the right side form a chain which will report in the last element, if the search key matches the node's value. If one or more byte values do not match, activation will follow the chain in the middle or the chain on the left and the right-most chain will not report.

5 Related Work

The first effort to create a direct hardware implementation of automata (NFA) goes back to Floyd and Ullman in 1982 [27], and has been an active area of research since then. All recent work that we are aware of has involved the use of regular expressions as the means for expressing automata and implementation as NFA in high-capacity FPGAs. Becchi [19] implemented a multi-stride (simultaneous input bytes) NFA compiled from Snort [25] pattern sets on Xilinx Virtex-4 and Virtex-5 FPGAs, obtaining between 2 and 7 Gbps of throughput. She estimated that about 1000 complex regular expressions implemented as multi-stride NFA could be deployed on a Virtex-5 XC5VLX50 device. Nakahara *et al.* [18] implemented a modular NFA (MNFA) on a Virtex-6, reporting system throughput of 3.2 Gbps on a compiled subset of Snort regular expressions. Yang and Prasanna [15] reported obtaining up to 11 Gbps on a Virtex-5 using various regex sets taken from Snort, implementing a compiled RE-NFA structure with multi-stride capability and enhanced character class processing. They reported being unable to fit the entire regex portion of Snort, consisting of 2630 expressions, into a single FPGA and broke their test runs into six rule sets. Kaneta *et al.* [16] developed a dynamically reconfigurable NFA on a Virtex-5 running regexes in different complexity classes at 2.9 and 1.6 Gbps for simple regexes and 0.5 Gbps for complex regexes. The major potential advantage of the design is that universal control logic is compiled into the FPGA and specific regexes are run by modifying memory, eliminating the cost of regex-specific compilation. Kaneta's design, however, is restricted in the complexity of regexes that can be implemented with this method. Wang *et al.* [17] created a counter-based NFA design

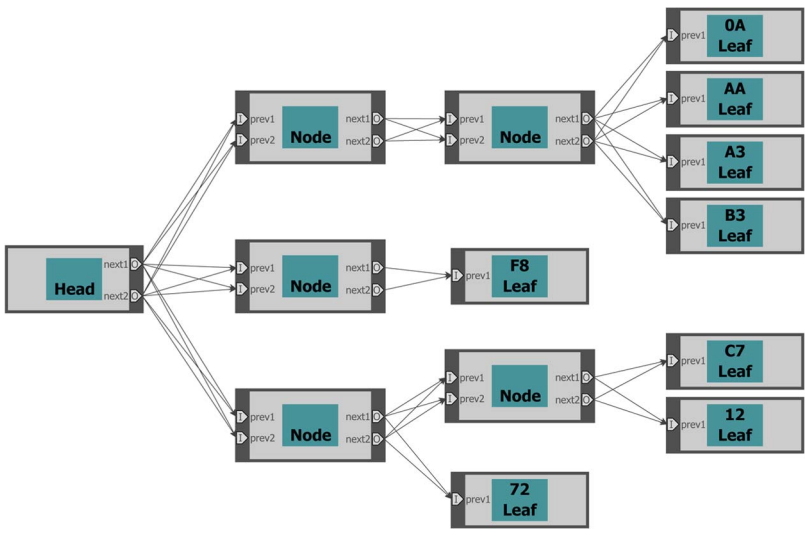

Fig. 11. ANML graph example.

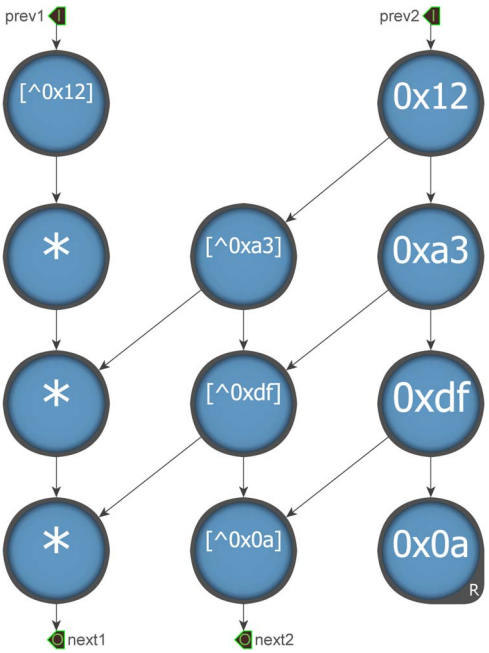

Fig. 12. ANML graph internal node.

that solves complex processing of character classes and developed a methodology for recognizing when overlapping character classes are inherently collision free. The design is capable of updating regexes through memory writes. A maximum system throughput of 2.57 Gbps was reported, but this was dependent on the complexity of the regex and the capacity of the FPGA only allowed for testing of partial rule sets derived from Snort.

A detailed comparison of the automata processor to the related works cited above is presented in the paper's supplementary material available online. The primary relevance of this prior work is that it has established NFA as the most efficient automata implementation for hardware. Our device is the only direct semiconductor architecture purpose-built for automata processing, as far as we know, and targets a different type of domain than high-capacity FPGAs, relative to cost, footprint, power consumption, and system integration. Much of the existing research has focused on adaptation of the building blocks of FPGAs, primarily LUTs and block memory, to regular expression-specific automata processing. Our architecture, designed to provide native support for PCRE, has addressed the challenges described in the prior work (for example, in character class handling as extensively treated in [17]). It is not, however, regular expression specific. The architecture allows for the concise implementation of NFA difficult or impossible to formulate with regular expressions and also allows creation of pushdown automata and dynamic control of automata at runtime. Partial dynamic modification of automata and incremental addition of automata at runtime are also supported. Other major differentiators with prior work include the ability to interrupt and resume input streams, with a mechanism for saving and restoring machine state, and the ability to distribute input data within a rank of automata processors to increase system scale.

6 Conclusion

The automata processor is, to our knowledge, the first semiconductor architecture of its kind; a non-von Neumann architecture designed for efficient and scalable automata processing. We have shown that the chip matches or exceeds the performance of high-capacity FPGAs programmed with comparable functionality, while providing greater logic capacity and offering superior capabilities in incremental and dynamic updates, the ability to interrupt and resume processing input streams, performance scaling by grouping processors into ranks, and expressiveness and computational power beyond that of regular expressions. Routine silicon process shrinks, by moving to a new process node, brings significant increases in capacity and higher throughput, even without improvements in the architecture. We expect, however, in addition to process shrinks, many improvements to the architecture which will improve speed, routing capacity, output capacity, and increases in computational power from the automata elements. We believe that our architecture also implies a new parallel programming paradigm, which we have begun to see evolving from ANML, and tools created for automata processor development.

References

[1] A.V. Aho and M.J. Corasick, ''Efficient String Matching: An Aid to Bibliographic Search,'' *Commun. ACM*, vol. 18, no. 6, pp. 333-340, June 1975.

[2] M. Becchi, C. Wiseman, and P. Crowley, ''Evaluating Regular Expression Matching Engines on Network and General Purpose Processors,'' in *Proc. 5th ACM/IEEE Symp. Architectures Netw. Commun. Syst.*, 2009, pp. 30-39.

[3] M. Góngora-Blandón and M. Vargas-Lombardo, ''State of the Art for String Analysis and Pattern Search Using CPU and GPU Based Programming,'' *J. Inf. Security*, vol. 3, no. 4, pp. 314-318, Oct. 2012.

[4] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, ''GPU-Based NFA Implementation for Memory Efficient High Speed Regular Expression Matching,'' *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 129-140, Aug. 2012.

[5] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, ''Parallelization and Characterization of Pattern Matching Using GPUs,'' in *Proc. IEEE IISWC*, 2011, pp. 216-225.

[6] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, ''Infant: NFA Pattern Matching on GPGPU Devices,'' *ACM SIGCOMM Comput. Communication Review*, vol. 40, no. 5, pp. 20-26, Oct. 2010.

[7] S. Pu, C.-C. Tan, and J.-C. Liu, ''Sa2px: A Tool to Translate Spamassassin Regular Expression Rules to Posix,'' in *Proceedings of Sixth Conferences on Email and Anti-Spam*, 2009, pp. 1-10.

[8] R. Baeza-Yates and G. Navarro, ''Faster Approximate String Matching,'' *Algorithmica*, vol. 23, no. 2, pp. 127-158, Feb. 1999.

[9] Y.-H. Yang, V.K. Prasanna, and IEEE, ''Space-Time Tradeoff in Regular Expression Matching with Semi-Deterministic Finite Automata,'' in *Proc. IEEE INFOCOM*, 2011, pp. 1853-1861.

[10] M. Becchi, M. Franklin, and P. Crowley, ''A Workload for Evaluating Deep Packet Inspection Architectures,'' in *Proc. IEEE IISWC*, 2008, pp. 79-89.

[11] M. Becchi and P. Crowley, ''A Hybrid Finite Automaton for Practical Deep Packet Inspection,'' in *Proc. ACM CoNEXT Conf.*, New York, NY, USA, 2007, pp. 1:1-1:12. [Online]. Available: http://doi.acm.org/10.1145/1364654.1364656

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, ''Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection,'' *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 339-350, Oct. 2006.

[13] M. Becchi and P. Crowley, ''An Improved Algorithm to Accelerate Regular Expression Evaluation,'' in *Proc. 3rd ACM/IEEE Symp. Architecture Netw. Commun. Syst.*, 2007, pp. 145-154.

[14] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* 2nd ed. Reading, MA, USA: Addison-Wesley, 2001.
[15] Y.-H.E. Yang and V.K. Prasanna, ''High-Performance and Compact Architecture for Regular Expression Matching on FPGA,'' *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 1013-1025, July 2012.
[16] Y. Kaneta, S. Yoshizawa, S.-I. Minato, and H. Arimura, ''High-Speed String and Regular Expression Matching on FPGA,'' presented at the Asia-Pacific Signal Information Processing Association Annu. Summit Conf., Xi'an, China, 2011.
[17] H. Wang, S. Pu, G. Knezek, and J. Liu, ''Min-Max: A Counter-Based Algorithm for Regular Expression Matching,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, Jan. 2013.
[18] H. Nakahara, T. Šasao, and M. Matsuura, ''A Regular Expression Matching Circuit Based on a Modular Non-Deterministic Finite Automaton with Multi-Character Transition,'' in *Proc. 16th Workshop Synthesis Syst. Integr. Mixed Inf. Technol., Ballroom*, 2010, pp. 359-364.
[19] M. Becchi, ''Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation,'' Ph.D. dissertation, Dept. Comput. Sci. Eng., Washington Univ., St. Louis, MO, USA, 2009.
[20] V.M. Glushkov, ''The Abstract Theory of Automata,'' *Russ. Math. Surveys*, vol. 16, no. 5, pp. 1-53, 1961.
[21] R.A. Baeza-Yates, ''Efficient Text Searching,'' Ph.D. dissertation, Univ. Waterloo, Waterloo, ON, Canada, 1989.
[22] G. Navarro and M. Raffinot, ''New Techniques for Regular Expression Searching,'' *Algorithmica*, vol. 41, no. 2, pp. 89-116, Feb. 2005.
[23] P.C. Fischer, ''Turing Machines with Restricted Memory Access,'' *Inf. Control*, vol. 9, no. 4, pp. 364-379, Aug. 1966.
[24] M.L. Minsky, ''Recursive Unsolvability of Post's Problem of 'tag' and Other Topics in Theory of Turing Machines,'' *Ann. Math.*, vol. 74, no. 3, pp. 437-455, Nov. 1961.
[25] Snort. [Online]. Available: http://www.snort.org
[26] X. Yu and M. Becchi, ''GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space,'' in *Proc. ACM Int. Conf. Comput. Frontiers*, 2013, p. 18.
[27] R.W. Floyd and J.D. Ullman, ''The Compilation of Regular Expressions into Integrated Circuits,'' *J. ACM (JACM)*, vol. 29, no. 3, pp. 603-622, July 1982.

Paul Dlugosch received a BS degree in electrical and electronics engineering from North Dakota State University, Fargo, ND, USA in 1989. He currently works at Micron Technology as Director of Automata Processor Technology Development. He has held previous positions in ASIC development and product engineering at Rosemount Industrial and Hewlett Packard. He has been awarded several patents in the field of system and semiconductor architectures. His current interests include automata-based computing applications, hybrid logic memory architectures and cognitive processing techniques using artificial neural networks and systolic array based architectures.

Dave Brown received BS and MS degrees in Electrical Engineering from Clarkson University, USA in 1988 and 1990, respectively. He currently works at Micron Technology in Allen, TX, USA as the lead chip designer for the automata processor. He has been lead designer on many DRAM devices including: SDRAM, mobile DDR, RDRAM and RLDRAM.

Paul Glendenning received a BE degree in engineering from James Cook University (Australia) in 1986. He joined Micron Technology in 2011 and is currently working in the DRAM Solutions Group as the Software Development Manager and Chief Scientist, Automata Processor Technology Development. He is responsible for developing the software tool chain to support the automata processor and for modeling next generation automata processor architectures.

Michael Leventhal received his BS-EECS degree from U.C. Berkeley, USA in 1994. As a member of the Automata Processor Technology Development team at Micron he has been responsible for the creation of the ANML language used to program the automata processor, developer's tools and investigation of automata processor applications. Prior to joining Micron he worked in the area of FPGA-based accelerators and was deeply involved in the creation of XML and the development of many internet-based software applications built around the use of XML. His current interests include automata theory and automata-based computing applications and bioinformatics.

Harold Noyes received his BS degree in Electrical Engineering from Utah State University, USA in 1981. He is the Senior Architect (Hardware), Automata Processor Technology Development, at Micron Technology, USA. Prior to joining Micron, he worked in various research and development roles for Hewlett-Packard Company, USA and contract engineering for Freescale Semiconductor, USA. His experience encompasses engineering and project management, including printed circuit assembly design, electromagnetic and regulatory compliance engineering, modem design, full-custom silicon SRAM design, ASIC design, and technical writing, for portable computers and LaserJet printers. He is a member of the IEEE and Tau Beta Pi.