Uses for Random and Stochastic Input on Micron's Automata Processor

Jack Wadden Ke Wang Mircea Stan Kevin Skadron
University of Virginia
Charlottesville, Virginia, USA
{wadden,kw5na,mircea,skadron}@virginia.edu

ABSTRACT

Micron's Automata Processor (AP) is a configurable memory-based device, purpose-built to emulate a theoretical non-deterministic finite automata (NFA). While NFAs are not particularly suited for floating point computation, they are extremely powerful and efficient pattern matchers and have been shown to provide large speedups over traditional von Neumann execution for rule-based, data-mining applications. While these kernels like associative rule mining, Brill tagging, protein motif identification have seen impressive orders of magnitude speedups, the exact capabilities of the AP and its advantage over traditional von Neumann architectures like CPUs and GPUs remains an open research area.

As an example of the potential power of the AP, we demonstrate its ability to simulate probabilistic automaton and accomplish stochastic computation using a random or stochastic input stream. The combination of probabilistic automata and stochastic computation open up a wide range of application kernels, previously not known to be implementable on the AP, with the potential for impressive speedups over CPUs and GPUs. This paper first presents the construction of probabilistic automata, and shows how they can be used to simulate Markov chains, Brownian motion, and stochastic computation. We then suggest small, low-cost changes to the AP architecture that would enable more efficient construction of these previously unimplementable application kernels.

Keywords

Automata Processor, Markov chains, stochastic computing, approximate computing

1. INTRODUCTION

As the breakdown in Dennard scaling makes it increasingly expensive to improve performance of traditional von Neumann architectures, heterogeneous computing, involving GPUs, DSPs, FPGAs, ASICs and other processors has provided promise as a possible path forward. Micron, leveraging their experience and IP in memory and semiconductor technology, has developed the Automata Processor (AP) [2, 3], a configurable memory-based processor, purpose-built to emulate a theoretical non-deterministic finite automata (NFA). While NFAs are not particularly suited for traditional integer or floating point computation, they are extremely powerful and efficient pattern matchers, and have been shown to provide large speedups over von Neumann architectures like CPUs and GPUs for rule-based data-mining applications [5, 6].

The AP emulates an NFA using a connected network of State Transition Elements (STEs) consuming an input stream of 8-bit symbols. Each STE is activated when it matches the current input symbol and is also activated by a connected STE. STEs can be designated as "start" STEs, thus activating on the input stream alone. Similar to traditional theoretic NFAs, STEs can also accept (report), thus generating a single bit output on activation. Alongside STEs, which are enough to emulate NFAs, the AP is augmented with boolean logic and counters (elements that activate after a set threshold of activations), and are thus strictly more powerful than theoretical NFAs. Micron's efficient implementation of the AP using traditional memory technology allows for powerful non-von Neumann computation.

However, the AP's added boolean and counter elements make it strictly more powerful than theoretical NFAs, and while kernels that exploit rule-based pattern matching, like association rule mining, parts-of-speech tagging, and protein motif identification [6, 5], have already seen impressive speedups over CPU implementations, other strengths and capabilities of the architecture are largely unknown.

This paper attempts to explore and expand non-obvious abilities of the AP by examining the uses of random and stochastic symbol sequences as input. Random input is defined as simply an input stream of uniformly distributed random symbols. Stochastic input is defined assigning a particular probability distribution to such random input. Because activations of STEs in the AP are conditional on the input stream, a probabilistic input stream immediately provides probabilistic activations. Thus probabilistic automata (PA) are implementable using the AP.

Practically, PA (also known as stochastic automata) are incredibly useful as a modeling formalism for finance, manufacturing, communication, bioinformatics, and many other disciplines. For example, Markov chains (a specific case of PA) are used extensively in practically all fields concerned with simulation. We present a proof of concept implementation of a simple simulation showing the construction and application of Markov chains on the AP. We also explore another previously unknown capability of the AP enabled by random input and the addition of boolean logic and counters: stochastic computation.

The rest of the paper is organized as follows. Section 2 describes the AP programming model and architecture in

University of Virginia Department of Computer Science
Technical Report TR# CS-2015-06, September 2015

further detail. Section 3 discusses the theory of Markov Chains, and several basic implementations and their various properties. Section 4 shows how linear Markov chains can be used to implement a simple asset price motion simulation. Section 5 gives a brief overview of stochastic computation and a potential implementation and use case in AP hardware. And finally section 7 shows how a few small changes to AP hardware could unlock speedups on a powerful new set of applications.

2. MICRON'S AUTOMATA PROCESSOR

The AP aims to reproduce the power of a theoretical NFA's non-deterministic parallelism. For problems with large, combinatorially difficult search spaces, non-determinism can be a powerful tool, enabling a fast, parallel exploration of problem instances. Previous implementations of NFAs in hardware fall into two categories: specialized hardware for DFA and NFA execution, and FPGA implementations. Specialized hardware to execute DFAs and NFAs has been created to accelerate regular expression matching. Unfortunately, these architectures are application specific and can only solve problems framed as regular expression matching. FPGA implementations of NFAs and DFAs can be much more flexible in their capabilities, but suffer from density and throughput limitations and often do not expose automata level programmability to the application developer, preventing the creation of automata that cannot be expressed as regular expressions [3].

Micron's unique memory-derived architecture takes advantage of the bit-level parallelism inherent in SDRAM arrays to gain improvements in state density over such previous NFA and DFA implementations. Micron also allows configuration of the AP using both PCRE and Automata Network Markup Language (ANML) offering programmers fine-grained control over automata construction [3].

2.1 AP Execution Model

AP automata are made up of a directed graph of STEs, boolean logic elements, and counter elements. Each STE can recognize an arbitrary character set of 8-bit symbols. An STE "activates" when it 1) recognizes the current input symbol and 2) it is "enabled." An STE is considered enabled when it is either configured to consume input from the input stream (a "start" STE), or an STE connected to it activated on the previous cycle. STEs can also latch activation, and then activate for the rest of execution. STEs can be configured to *report on activation*, producing a 1-bit output. This is analogous to accepting an input string in an NFA. An example of an AP design recognizing whether or not all input strings over 0,1 contain a 1 in the third position from the end is shown in Figure 1. Note the streaming nature of this design, i.e. a report will be generated for all satisfying substrings of the input, and not simply at the end of data input.

The AP also contains boolean logic elemens, which provide AND, OR, NOT, $NAND$, NOR, sum-of-products, and product-of-sums capability. The AP also provides special counter elements, which only activate after a pre-set threshhold of input activations is reached. Both of these elements (boolean and counter) are not available to theoretical NFAs and expand the set of languages recognized by the device past regular languages. For example, boolean AND elements can combine the results of two NFAs or NFA paths,

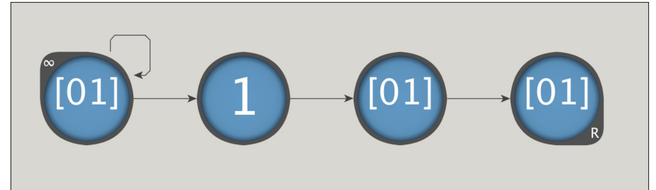

Figure 1: An graphical representation of four STEs representing the NFA that recognizes all strings over 0,1 containing a 1 in the third position from the end. The first STE is configured as a start STE and is implicitly activated by every input symbol (indicated by the infinity symbol). The last STE is configured to report when activated (indicated by the 'R' symbol.

providing the power of infinite, non-consuming look-ahead. Figure 2 shows how two latching STEs and an AND element can be used to detect whether a string contained both a lower-case letter and a number [2]. Note that unlike the previous example, this design only reports on the end of data input, symbolized by the E.

2.2 AP Architecture

The AP implements STEs using 256 bit memory columns ANDed with an enable signal. Each 256-bit column vector represents a character set of 256 possible 8-bit characters that this STE could be activated by. Any character, or character set, supplied as a row address will then force all STEs columns that recognize that character set to read out a 1. For example, the Kleene star operator would simply fill all bit rows in the STE column with 1s. Thus an STE is capable of recognizing an arbitrary character set of possible input symbols on every cycle. If a column reads a 1, and the STE is enabled, the STE activates and sends its output signal to the routing matrix.

The routing matrix enables STEs to enable any other STE within the same AP core, and is pre-configured (placed and routed) based on the compiled AP application design.

Columns of STEs are organized into *blocks* and a number of blocks makes up an AP *core*. Because the routing matrix only exists within cores, STEs are prevented from enabling other STEs across cores. In the current AP hardware, a block contains 256 STEs, 4 counters, and 12 boolean elements. AP cores contain 96 blocks, offering a total of 24,576 STEs per core.

As previously discussed, a boolean elements combine activations as logical signals. They are extremely useful in combining the results from different automata and coordinating computation.

Counter elements allow activation after a preset threshhold and are a simple, but powerful manifestation of storage on the AP. Every time a counter is activated via its *count* input, its internal storage increments from 0. A counter also has a *reset* input which sets the internal count to zero when activated. When the counter's internal threshhold is reached, the counter activates. Counters can be configured to either reset their count after activation, or continue to activate (i.e. latch activation) until they are explicitly reset.

3. MARKOV CHAINS

In informal terms, Markov Chains are automata with probabilistic transitions between states. To be formally consid-

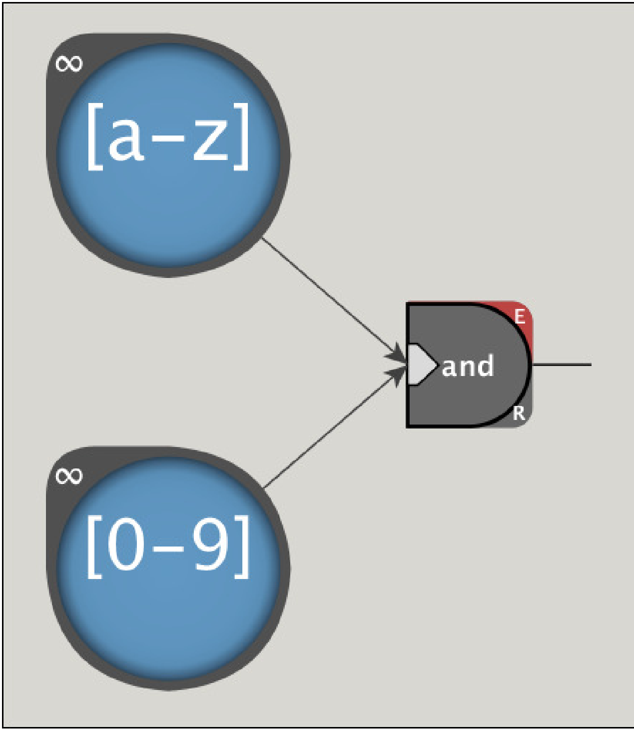

Figure 2: An AP design recognizing all strings with both a lower-case letter and a number. The *AND* element is configured to only report on the end of the data stream.

ered Markov chains, transitions in the automata must 1) be stochastic processes (i.e. they occur with some set probability), and 2) respect the *Markov property*, which states that every probabilistic transition depends only on the current state, and is not influenced by memory of prior states. An example Markov Chain implementing an unfair coin is illustrated in Figure 3.

Markov chains are defined by stochastic *transition matrices* which hold all transition probabilities from a start state (row) to a transition state (column). Each row of the transition matrix must be *stochastic*, i.e. each column must add up to 1. Logically, this makes sense because we must always make some transition in a time step, even if it is to the current node. The transition matrix for the unfair coin example in Figure 3 is shown in Table 1.

The follwing section describes how we can use probabilistic input and STEs to create Markov Chains on the AP hardware.

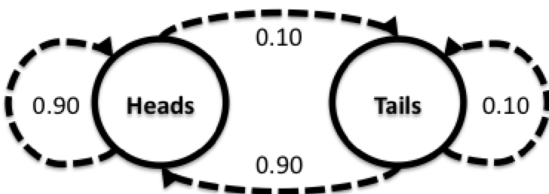

Figure 3: A simple Markov chain, which simulates an unfair coin toss, with two states *Heads* and *Tails*. Transition probabilities between these states are *unfair* meaning that the probability of transitioning to/flipping *Heads* is different than *Tails*.

Table 1: Stochastic transition matrix of a Markov chain representing an unfair coin. Note the probability of transitioning "to heads" or "to tails" from any state is always .9, and .1 respectively.

	To Heads	To Tails
From Heads	0.90	0.10
From Tails	0.90	0.10

3.1 Mapping Markov Chains to the AP Hardware

To easily communicate the concept of probabilistic transitions and Markov Chains on the AP, we first begin with an simple-to-understand construction that is not necessarily STE efficient but clearly illustrates the mapping.

Star-State Construction:

Consider the unfair coin example described in Section 3 and shown in Figure 3. To produce probabilistic transitions we first restrict the input symbols to be random and uniformly distributed over some range. The method to create a random input symbol stream is not discussed here as it is orthogonal to the Markov chain implementation, however, random number generation is extremely important in this context and deserves the utmost attention in a full implementation.

Each Markov chain can be constructed in the following manner provided a stochastic transition matrix:

Algorithm 1: Construct AP Markov Chain Simulation

Data: Square Stochastic Matrix $StochMat$; Set of possible input symbols Σ
Result: AP Markov Chain Simulator

1 INITIALIZATION;
2 **foreach** $FromState$ **do**
3   Create a reporting STE which recognizes Kleene star representing $FromState$;
4   Randomly select a single $FromState$ to be start state, activating on start of data;
5 CONSTRUCTION;
6 **foreach** $FromState$ **do**
7   **foreach** $ToState$ **do**
8   Create transition STE $TransNode$;
9         $TransProb \leftarrow StochMat[FromState][ToState]$;
10   Without replacement, randomly select $TransProb * |\Sigma|$ symbols from Σ as the character class recognized by $TransNode$;
11   Add edge from $FromState$ to $TransNode$;
12   Add edge from $TransNode$ to $ToNode$;

An example of this construction for the unfair coin example is shown in Figure 4. For illustrative purposes, we restrict the input symbols to be within the character class $[0-9]$. For this construction, we randomly choose a single state out of all possible states to act as the start state, although choosing multiple or all states as start states implicitly implements an interesting feature called *Markov chain*

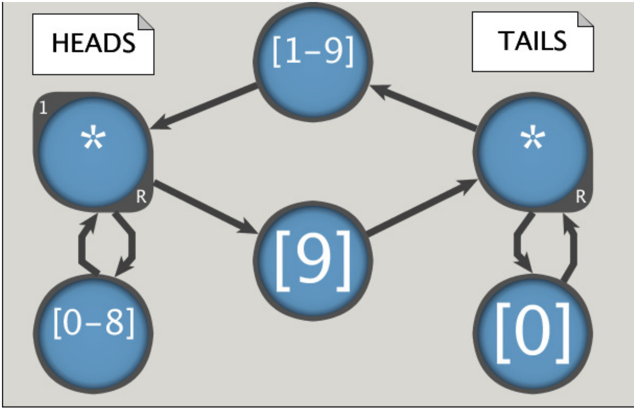

Figure 4: A simple Markov chain, which simulates an unfair coin toss, with two states *Heads* and *Tails*. Transition probabilities between these states are *unfair* meaning that the probability of transitioning to/flipping *Heads* is different than *Tails*.

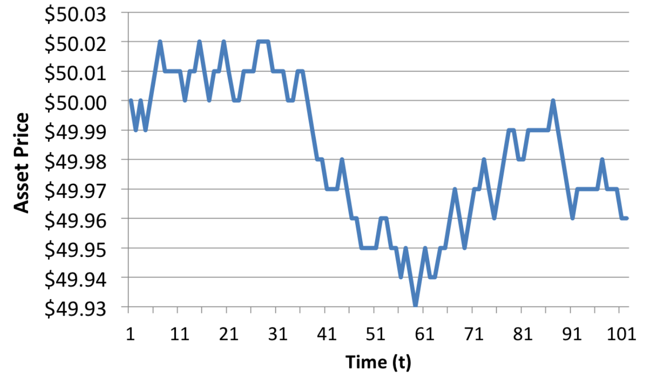

Figure 5: Asset price simulation modeling a random walk with transitions of +/-$0.01 or no change.

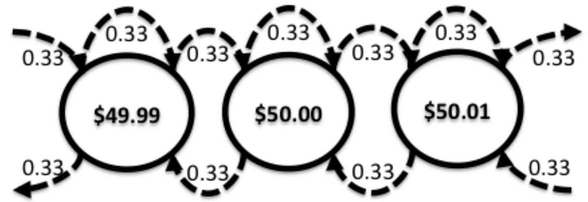

Figure 6: Linear Markov chain modeling a random walk with transitions of +/-$0.01 or no change.

coupling, which reduces the time it takes the Markov chain to reach a stationary distribution.

Proof of construction: To prove that this construction indeed simulates a Markov Chain, we must show that 1) the transitions are stochastic according to the transition matrix and 2) that the Markov property is respected.

(1) The fact that transitions between states are stochastic transitions should be obvious when provided with random input. Given a state X, we have constructed all transition nodes to contain random buckets of symbols according to the probability presented in the given stochastic matrix. When presented with a uniformly distributed random symbol, we will therefore transition to the transition node with the same probability as the ratio of symbols in that transition node to the total number of possible input symbols. We can thus always construct these sets of buckets to be identical to the transition matrix. Because we can easily direct transition nodes to activate the desired "to state" star-node, it is trivial to see that this construction simulates stochastic transitions between states according to the desired stochastic matrix.

(2) It is also not hard to see that the Markov property is always respected. Because transitions only involve the current state, and a random input symbol, there is no way for previous states, or iterations, to effect the probability of the next state transition.

4. SIMULATING ASSET PRICE MOTION USING RANDOM WALKS ON LINEAR MARKOV CHAINS

Asset price motion is concerned with simulating and predicting the price of a financial asset over some set amount of time. Based on randomly generated input, an asset price is adjusted over time according to a motion function. This function is often chosen to be *Brownian motion*. Brownian motion was originally developed as a part of particle theory and defined to model the random motion of particles in a system, but has been re-purposed as a proposed model for behavior of the price of assets and financial markets.

Because Brownian motion is a stochastic process, it can be simulated using *random walks* on Markov chains []. Random walks are driven by random transitions (defined by a motion function) between states of a *walker* over some dimensional space of states. In our asset price example, the state space is simply all discrete prices that the asset could occupy, thus a $1D$ number line. At any given time, the price could go up or down depending on a random input, and the added or subtracted amount would be determined by the motion function. At the end of the simulation, the final state of the walker is the predicted asset price of the simulation.

An example of a $1D$ random walk simulation of an asset price is illustrated in Figure 5. $1D$ random walks can be simulated using linear Markov chains, or Markov chains with diagonal or banded stochastic matrices. As long as the transitions to neighboring (or nearly neighboring) states correctly encode the proposed transition function. An example of the linear Markov chain that produced the prior simulation is shown in Figure 6. Note that by definition, our linear Markov chain is only capable of modeling discrete, non-continuous values. However, as long as we can create enough states, we can model to arbitrary precision with arbitrary price bounds. Our random walk construction described in the next section shows how we can implement this arbitrary precision on the AP without this 1-1 correspondence between states and discrete values.

For our explanitory example, we will simply set the cost function to be +/-$0.01 with equal probability, and then explain how to construct simulated arithmetic Brownian motion cost functions.

4.1 Mapping an Asset Price Simulation to the AP Hardware

We now show how Markov chains on the AP can be used to simulate asset price motion via random walks on a linear Markov chain. We first present the construction of a +/-1

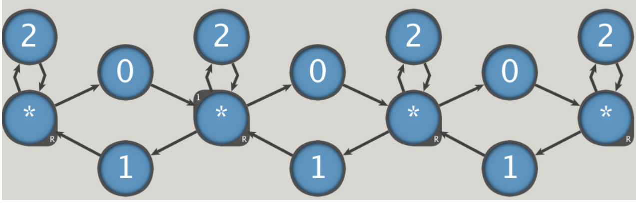

Figure 7: How a linear Markov chain can be implemented on the AP. This chain corresponds to the transition matrix in section 4.1.1.

linear Markov chain "walker" simulation, and then present a method to avoid the 1-1 correspondence between states and discrete values in the simulation.

4.1.1 *Linear Markov Chain Random Walker:*

To construct a linear Markov chain "walker," we first need to define the appropriate diagonal stochastic transition matrix according to our transition function. Because we define our current transition function to increment or decrement by \$0.01, or remain at the current price with equal probability, we know that the transitions from each state to its neighbor will have a probability of 0.33. Thus the transition matrix looks like the following:

$$\begin{vmatrix} 0.33 & .33 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ .33 & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & .33 & .33 & 0 & 0 & 0 & 0 & 0 \\ 0 & \cdots & .33 & .33 & .33 & 0 & 0 & 0 & 0 \\ 0 & \cdots & 0 & .33 & .33 & .33 & 0 & 0 & 0 \\ 0 & \cdots & 0 & 0 & .33 & .33 & .33 & 0 & 0 \\ 0 & \cdots & 0 & 0 & 0 & .33 & .33 & .33 & 0 \\ \vdots & \cdots & 0 & 0 & 0 & 0 & .33 & .33 & .33 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & .33 & .33 \end{vmatrix}$$

Each state then represents a certain value of the asset with cent precision, and can transition to its +\$0.01 or -\$0.01 neighbor or remain at the current price with equal probability. Later we will discuss how to modify the transition matrix to simulate other, more complicated transition functions.

Because we only have three transitions, *up*, *down*, or *stay* with equal probability we can simplify the construction of the Markov chain to consider only random input in the range $[0-2]$, where a 1 will force a transition to the neighboring up state, and 0 to the neighboring down state, and a 2 will cause the walker to remain in the current state. An illustration of a section of such a Markov chain walker as it would be implemented on the AP is shown in Figure 7.

4.1.2 *Bounding Walkers with Counters:*

With the above construction, we only have enough STEs per AP core to represent 10,000 discrete prices, or values from \$0.00 to \$100.00. This is not a realistic implementation as prices may be out of this range, and also may have much larger variation, or need more precision. As well, because this construction utilizes almost all STEs on a core, it would preclude parallel simulations or heterogeneous functionality. Therefore, we propose bounding small walkers with counters as a way to reduce the STE costs of such simulations.

Figure 8: A linear Markov chain walker bounded by counters. This construction can represent discrete values with arbitrary precision without a linear increase in states.

The intuition behind counter bounding is that if we can keep track of how many times we "fall off" the edge of a small walker in both directions, we can simulate an infinite sized walker. We accomplish this by using two counters: one to count up falls, and one to count down falls. If the simulation ever falls off either end of the walker, it activates both the corresponding counter, as well as the STE on the other end of the walker, thus "wrapping around" in a ring. At the end of the simulation, the end price will be the current price plus the difference between the counters multiplied by the size of the counter. An example of such a construction is shown in Figure 8.

We can also use this technique to bound higher dimensional random walk Markov chains. For every dimension, we need two counters to keep track of whenever we "fall off" the top or bottom of each. The difference in the counters times the size of the walker plus the value in the walker, will always give the coordinate of the random walk in that dimension.

4.1.3 *Extracting Values from Counters:*

As the value stored within counters is not readily available, we need some way to extract it. We accomplish this be reserving a single symbol for "counter pumping." Consider a counter with target 16 and internal value 13. To know 13, we activate the counter using the pumping symbol until it activates (3 cycles). In post-processing, we now know that the value stored in the counter was the target (16) minus the number of cycles pumped before activation (3), thus 13 is known.

Because counters can have large targets, we may need to pump a large number of times, wasting time and power. Therefore, in practice, two counters may need to be used to keep track of digits of the count, bounding the number of pumps to be the radix of the chosen counting scheme. For example, to construct a two digit counter, have the first digit activate the second digit when it reaches its target and then reset its count to 0. Pumping these counters will therefore efficiently extract each digit of the number.

4.2 Final Construction

We construct a prototype asset price simulator based on the example described in the previous section. The full prototype, including walker, bounding counter structures, and pumping system, are shown in Figure 9.

We purposefully use four counters in this construction because in that case, we can tune the size of the walker such that the entire simulation fits inside a single AP block.

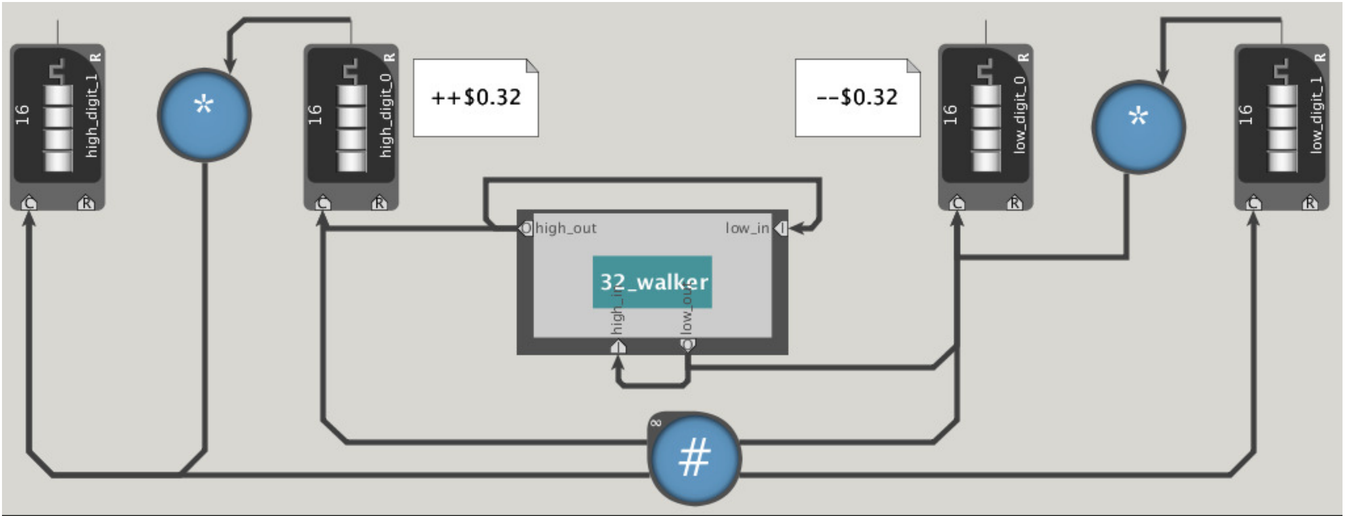

Figure 9: A linear Markov chain walker bounded by counters. This construction can represent discrete values with arbitrary precision without a linear increase in states.

Table 2: Stochastic bit-stream representations of a few different probabilities. Note that there can be multiple different representations of different ratios, reflecting the random, stochastic nature of this number system.

Prob.	Stochastic Stream
0 (0/10)	0000000000
1 (10/10)	1111111111
.6 (6/10)	0110100111
.6 (6/10)	1110001110
.6 (6/10)	1010101011

While simulation of Brownian motion is just one example of the uses of Markov chains, other practical uses for simulations of Markov chains exist. For example, Markov Chain Monte Carlo simulation, which was voted one of the ten most important algorithms of all time, is heavily used in theoretical and applied science and mathematics, and although not feasible for large, continuous state spaces, could be practically implemented on the AP when states are sparsely connected (linear arithmetic Brownian motion), or when the application calls for a relatively small, known transition matrix (credit default modeling).

5. STOCHASTIC COMPUTATION

Stochastic computation (SC) was first described in the early 60's [4] as a method of computing on probabilities represented by continuous binary input streams. Each input stream represents a probability using the proportion of 1's to 0's in a randomly generated sequence. For example, out of 10 bits, Table 2 shows stochastic bit-stream representations of a few different probabilities. Note that there can be multiple different representations of different ratios, reflecting the random, stochastic nature of this number system.

The main advantage of computation on stochastic bit streams is that its remarkably simple and utilizes extremely low-cost digital elements. For example, multiplication of two stochastic input streams, shown in Figure **??**, requires a single AND

Figure 10: Approximate multiplication of two stochastic input streams [1]. This example illustrates exact computation of $4/8 \times 6/8$.

Figure 11: Approximate multiplication of two stochastic input streams [1]. This example illustrates inexact computation of $4/8 \times 6/8$.

gate, whereas a digital multiplier would require thousands of individual logic gates!

While SC is simple to implement, the technique suffers from two main drawbacks that have prevented its mainstream adoption. The first and most significant drawback is the inherent inaccuracy involved in SC. Because a probability cannot be measured exactly, and therefore must be estimated, random variance in SC is inevitable. And although higher order terms of computations are computed first (available early, before lower order terms), high precision of lower order terms cannot be guaranteed. The second drawback, related to the first, is that in order to reduce variance, such that results from stochastic computations have an acceptable precision, a large amount of time and bandwidth is required. For example, if we compute using a generating process

$$\hat{p} = k/N \quad (1)$$

where k is the proportion of 1s out of N bits, then the stan-

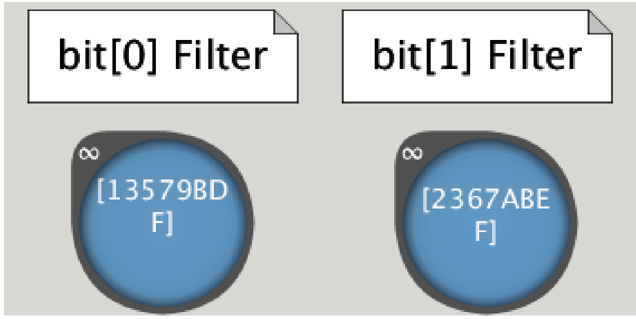

Figure 12: Stochastic filter showing how to extract bit pulses from symbols in the range [\x00-\x0F].

dard deviation of the sampled probability is

$$\sigma(\hat{p}) = [p(1-p)/N]^{1/2} \tag{2}$$

Therefore the accuracy of a stochastic measurement or computation increases on the order of the square root of the time or length of the input sequence [4].

Using examples from the fundamental introduction to SC "Stochastic Computing Systems" by B.R. Gaines [4], we first show how arithmetic, traditionally thought of as a weakness by automata, can be implemented using automata coupled with boolean logic elements and counters. We then show how the output of multiple, uncorrelated functions across different automata could be used to reduce the critical path of precise computation. Finally, we end with a proposal for small adjustments to the AP hardware (such as decrementing counters and distributed random number generators), which would enable more stochastic computing capabilities.

6. MAPPING STOCHASTIC COMPUTATION TO THE AP HARDWARE

Stochastic streams can be naturally implemented on the AP using activations to represent bits flowing through the device. By using only STEs with unconditional activation (i.e. Kleene star), an activation acts as a binary pulse, just as in a digital computer, flowing through all star states it connects to. The following sections explain how to create and control such input, compute on stochastic streams, and then convert stochastic output to binary, all on the AP.

6.1 Stochastic Input

6.1.1 *Direct Modulation:*

While input on the AP is normally thought of in terms of discrete 8-bit symbols, we consider treating individual bit of the input stream as a separate stochastic stream. This means that naively, we have control over 8 inputs into any stochastic function we construct. We call this input method *direct modulation*. However, we still need some way to convert the symbol representation of the input streams into activation pulses. This can be done by creating *filter nodes* that only activate on the character set of symbols with that bit set. This essentially filters the input symbol on a particular digit, creating a corresponding activation pulse every time that bit is sent into the AP. For example, the character set to filter bit 0 and 1 of a hypothetical 4-bit input symbol is shown in Figure 12.

Table 3: Stochastic transition matrix of a Markov chain that generates a stochastic stream with probability p.

	1 State	0 State
1 State	p	$1-p$
0 State	p	$1-p$

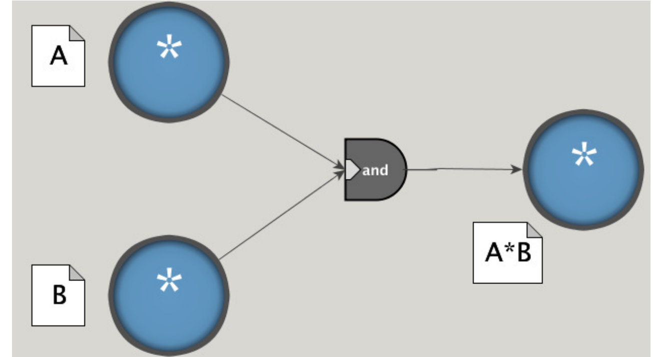

Figure 13: Stochastic multiplication on the AP using an AND element.

6.1.2 *Markov Chain Input Generation:*

Because 8 inputs may not be enough input for the desired function and precludes the use of the input stream for other application input while in simultaneous operation, we introduce another method of generating input using the Markov chains introduced in Section 3.

Markov chains can be set up to generate arbitrary Bernoulli processes by the following construction: Create a two state Markov chain simulating a coin toss according to Algorithm 1 using the stochastic matrix in Table 3 and pick p to be the generating probability of the desired stochastic stream. On a random symbol input, the chain will activate the 1 state with probability p. We consider this state a *stochastic generator*, as it now represents the stochastic number p. Therefore, *any* stochastic number stream can be created on-chip by constructing a Markov chain with a stochastic generator node.

6.2 Stochastic Computation with Boolean and Counter Elements

This section presents various binary arithmetic functions, their implementations according to Gaines [4], and our construction using AP elements. We always consider operations on two stochastic number streams A and B. For further discussion on these constructions, we refer the reader to the Gaines paper [4].

A stochastic multiplier is constructed by simply ANDing A and B together. We can obviously use the AND capability of the boolean elements to implement this functionality with no cycle penalty. An example of stochastic multiplication was already shown in Figure **??**. Stochastic multiplication as implemented on the AP is shown in Figure 13 below.

Stochastic addition is conceptually more complicated than multiplication. Because stochastic numbers are probabilities, and are defined to exist within the [0,1] range, adding such numbers may not make sense. Stochastic addition is therefore *scaled* where $P(A)+P(B) = (P(A)+P(B))/2$. This is analogous to randomly picking bits from either stream

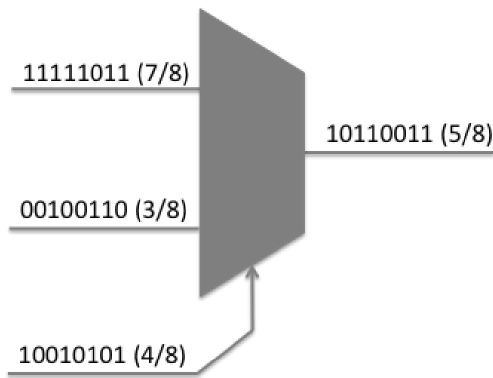

Figure 14: Stochastic adder.

to compose a new stream. Gaines constructs stochastic adders using a MUX of two streams, selected by a fair coin stream C. An example of a scaled stochastic adder is shown in Figure 14. We can construct a stochastic MUX on the AP by ANDing A with a .5 probability stream C and B with the compliment of C, thus implementing a scaled stochastic adder. Our general construction of this stochastic adder is shown in Figure 15.

A scalar division of stochastic streams can be accomplished by counting the number of activations of the numerator A, and activating when this count reaches a scalar C. An example of a scalar divider is shown in Figure 16. This is the exact functionality of the AP's counter element, and so scalar division is trivial to implement on the AP. Our general construction of a scalar divider on the AP is shown in Figure 17 using 7 as an example scalar target.

Stochastic division, i.e. dividing A by B, can be accomplished using counters with decrement capability. Because the counter elements on the current generation of the AP do not have this functionality, this efficient construction cannot be implemented. However, this functionality could be easily added to the architecture, and is discussed in Section 7.

Subtraction requires a negative representation in binary. In his work, Gaines discusses three different stochastic number systems that support addition of negative numbers and their corresponding computation. We refer the reader to Gaines [4] for a discussion of these models, and leave their corresponding AP constructions for future work.

6.3 Stochastic Output

Eventually we may need to read out a stochastic value from the AP if it is not consumed internally. This can be accomplished by using the AP's reporting capability [2] and computing the ratio of reports to a desired input length off-chip. However, depending on the report frequency, and the number of STEs reporting at any one time, this may cause a reporting bottleneck. To stalls resulting from this bottleneck, we can also use counters to implement *stochastic integration*. Simply count the number of activations of a stochastic result, and then extract the value when necessary after a certain number of counted cycles. We can then delay reporting until after a desired number of cycles N, pump the counters to extract their value A (as described in Section 4.1.3), and compare the counter value with the number

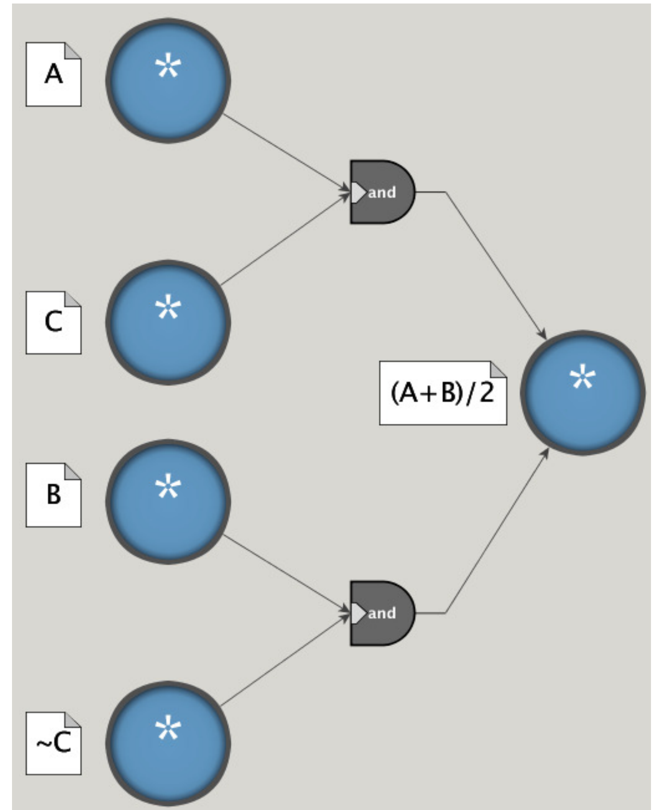

Figure 15: Stochastic addition on the AP using two AND elements and a selector signal C.

Figure 16: Stochastic division on the AP using a counter with target 7.

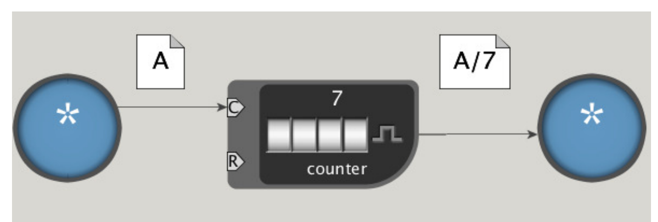

Figure 17: Stochastic division on the AP using a counter element.

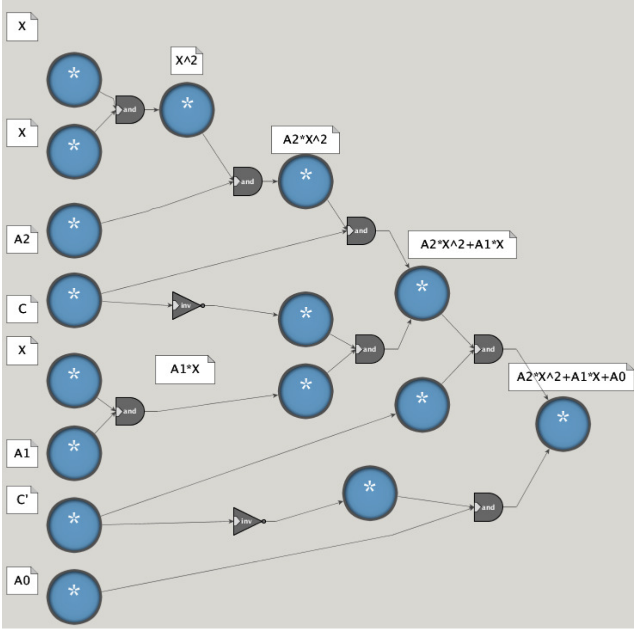

Figure 18: Example of how a two degree polynomial can be evaluated on the AP. This automata evaluates $A2 * X^2 + A1 * X + A0$. Note that this implementation takes only 9 boolean elements.

of cycles passed as an estimate of the stochastic result A/N.

6.4 Chaining Stochastic Arithmetic

To evaluate arbitrary functions, we must be able to pipe computed stochastic streams to new computing elements. We can accomplish this by using Kleene star STEs. Figure 18 shows how an arbitrary two degree polynomial can be evaluated on the AP by chaining together successive multiplications and additions. Note that re-using a stochastic input stream for multiple different computations may introduce correlation among streams. Correlation of input can have disastrous results, and ruin the precision of the computation. For example, consider the computation of A^2. If we use the same input A to supply both inputs of our multiplying AND gate, the result is trivially always A, not A^2. Thus, correlation should be avoided at all costs. Input streams can be de-correlated by involving separate input streams to represent the same probability, or separate Markov chain generators for the same probability. Other de-correlation techniques may be able to cheaply allow a single stochastic stream to be consumed by many different functions and is left for future work.

6.5 Reducing Variance with Parallel Computation

While variance in stochastic computation is normally reduced by increasing the number of input bits (i.e. N from equation 2), we can accomplish variance reduction by simply running multiple computations in parallel. Practically, this can be done by either averaging the scalar output of the resulting computation or by doing scaled stochastic additions on-chip. Each parallel computation will add N bits to the computation, but require the same amount of time providing a clear resource/time trade-off. Quantifying exactly how

Figure 19: Counter implementing stochastic division of arbitrary input streams.

variance is reduced by these two techniques is left for future work.

7. PROPOSED ARCHITECTURAL ENHANCEMENTS

Random input, Markov chains, and stochastic computing are an exciting initial exploration of the additional capabilities (aside from REGEX processing) provided by boolean logic and counter elements. However, small changes to the AP hardware would make Markov chains and stochastic computation much more attractive and powerful abilities of the AP. Some of these proposed architectural enhancements are discussed below.

Counter Decrement: Arbitrary stochastic division requires the ability to decrement, as well as increment, the value in a counter. Figure 19 shows how stochastic division could be implemented if such a capability existed [4]. Counter decrement would also make techniques like counter bounding more efficient. Because counters cannot be decremented, we currently have to use two counters to keep track of up and down counts when a walker falls up or down the walker simulation. A decrement port in the counter element would enable a single counter to keep track of the dimensional coordinate, incrementing when the walker falls up and decrementing when the walker falls down.

Per Core Pseudo-Random Number Generation: If too many Markov chains share the same random input, they may become correlated and produce output unsuitable for random simulations or probabilistic generation. Because each AP chip is limited to a single 8-bit input, this may be a significant bottleneck, preventing massively parallel random simulation. We therefore propose adding small hardware pseudo-random number generators (PRNG) within each AP half-core. STEs could then be directed to consume either from the symbol stream, which would still exist for normal input, or from their half-core's PRNG. This would greatly increase the possible number of parallel simulations, and also eliminate any input stream bandwidth dedicated to random input.

Adaptive Counter Elements: Some artificial intelligence techniques such as Neural Networks and Hidden Markov Models require *adaptive threshold logic elements* (ATLEs) to be implementable. ATLEs fire which a certain input threshold has been reached. Counters may seem like a natural implementation target for an ATLE but counters are unable to record parallel activation. A hardware ATLE, capable of firing when a parallel target threshold is reached, would make these computation models much easier to implement on the AP.

Classed Reporting: Many small Markov chains can be used for massively parallel simulation on the AP, however, recording reports on every cycle from each simulation would bottleneck the reporting system, and cause stalls in the ar-

chitecture. Counters could be used to reduce these stalls, but counters are a scarce resource on the AP when compared to STEs and would be required for every Markov chain state in every simulation to generate the same output. We therefore propose a hardware structure to facilitate *classed reporting*. Classed reporting aggregates reports from classes of STEs into a single count. The functionality here is very similar to a histogram. The AP currently lacks a way to link the reports of STEs with similar, parallel function. A hardware structure build into the reporting vector could snoop for reports of a certain class, increment a counter, and discard the report until the end of data. When the data input stream ceases, the structure could issue a single histogram report, noting how many STEs of each class reported. This would remove the reporting bottleneck for parallel simulations, and may also help the efficiency of other classification automata.

8. CONCLUSIONS AND FUTURE WORK

This paper is the first to explore the uses of random and stochastic input on Micron's Automata Processor. We first show how random input enables the AP to act as a probabilistic automata. We then explore three different uses for random transitions or probabilistic input streams 1) Markov Chains, 2) random walks, and 3) stochastic computation. While the presented examples are admittedly small, toy examples, they are a useful initial exploration of non-obvious capabilities of the AP enabled by boolean logic elements and counters. We also propose new small hardware structures, which will further enhance the power and efficiency of the AP.

Future work investigating the usefulness might include, but is not limited to:

- Quantifying how many Markov chains are able to consume a single byte of random input, without introducing unacceptable correlation among simulations
- Quantifying how parallel execution increase the precision in time of stochastic computation as discussed in section 6.5.
- Exploration of the efficiency of other useful manifestations of PA, for example Stochastic Automata Networks, Stochastic Petri Nets, and Performance Evaluation Process Algebras.
- Conducting a cost/benefit analysis of the additional hardware structures proposed in section 7.

9. ACKNOWLEDGMENTS

The authors would like to thank the members of the Center for Automata Processing at the University of Virginia for their insightful input.

10. REFERENCES

[1] A. Alaghi and J. P. Hayes. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):92:1–92:19, May 2013.

[2] 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*, 99(PrePrints):1, 2014.

[3] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. Supplementary material for an efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2014.

[4] B. R. Gaines. Stochastic computing. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 149–156, New York, NY, USA, 1967. ACM.

[5] I. Roy and S. Aluru. Finding motifs in biological sequences using the micron automata processor. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 415–424, May 2014.

[6] C. Sabotta. *Advantages and challenges of programming the Micron Automata Processor*. PhD thesis, Iowa State University, 2013.