Generating Efficient and High-Quality Pseudo-Random Behavior on Automata Processors

Jack Wadden, Nathan Brunelle, Ke Wang, Mohamed El-Hadedy, Gabriel Robins, Mircea Stan and Kevin Skadron
University of Virginia, Charlottesville, Virginia, USA
{wadden,njb2b,kw5na,mea4c,robins,mircea,skadron}@virginia.edu

***Abstract*—Micron's Automata Processor (AP) efficiently emulates non-deterministic finite automata and has been shown to provide large speedups over traditional von Neumann execution for massively parallel, rule-based, data-mining and pattern matching applications. We demonstrate the AP's ability to generate high-quality and energy efficient pseudo-random behavior for use in pseudo-random number generation or in chip simulation. By recognizing that transition rules become probabilistic when input characters are randomized, the AP is also capable of simulating Markov chains. Combining hundreds of parallel Markov chains creates high-quality, high-throughput pseudo-random number sequences with greater power efficiency than state-of-the-art CPU and GPU algorithms. This indicates that the AP could potentially accelerate other Markov Chain-based applications such as agent-based simulation. We explore how to achieve throughputs upwards of 40GB/s per AP chip, with power efficiency 6.8x greater than state-of-the-art pseudo-random number generation on GPUs.**

I. Introduction

As the breakdown in Dennard scaling makes it increasingly expensive to improve performance of traditional serial von Neumann architectures, heterogeneous computing, involving GPUs, DSPs, FPGAs, ASICs, and other processors, promises a possible path forward. The Micron Technology corporation has developed the Automata Processor (AP) [1], a reconfigurable, native-hardware accelerator for non-deterministic finite automata (NFA) based computation. Micron's unique, memory-derived architecture leverages the bit-level parallelism and address broadcast micro-architecture inherent in memory arrays in order to gain improvements in efficiency and state density over previous methods for emulating NFAs in hardware [1].

The AP implements an NFA using a reconfigurable network of *state transition elements* (STEs) – analogous to NFA states – that all consume a single input stream of 8-bit symbols. If an STE matches the input symbol, it activates, and causes transitions to other STEs via an on-chip routing matrix. STEs are capable of single-bit reports, analogous to "accepting states" in traditional NFAs. The AP is an extremely powerful and efficient pattern matcher, and has been shown to provide large speedups over von Neumann architectures such as CPUs and GPUs for massively parallel rule-based data-mining applications [2], [3], [4], but the exact capabilities of the AP and its advantage over these architectures remains an open research question.

This paper explores one novel application of the AP: a high-quality source of pseudo-random behavior for pseudo-random number generation (PRNG) and other potential simulations that can be defined with Markov chains (such as agent-based models). We achieve efficient, high-quality, massively parallel, MISD pseudo-random behavior by constructing and running many parallel Markov chains, simulated using AP-based NFAs.

Instead of driving automata transitions using conventional input data (e.g. a DNA sequence), we consider driving automata transitions using random or pseudo-random input. Because transitions between states in the AP are conditional on the input stream, a stochastic input stream immediately provides stochastic automata transitions, *even though the transition rules are deterministic*. Thus, probabilistic automata, including finite state Markov chains, can be emulated using the AP.

Using this intuition, we develop a novel method for creating high-quality pseudo-random behavior using Markov chains modeled by NFAs on the AP, which we call AP PRNG. We use parallel Markov chains to model rolls of fair dice, and then combine the results of each roll into a new pseudo-random output string. By combining the output of parallel rolls, driven by a single stream of random or pseudo-random input symbols provided by a host processor (e.g. the system CPU), we can construct new pseudo-random output hundreds of times larger than the input used to drive transitions on chip.

However, because we emulate Markov chains using NFAs with fixed transition functions, *any* non-trivial number of parallel Markov chains that consume the same input will produce output that is eventually correlated and patterned. Thus, it should be demonstrated that on-chip pseudo-random behavior can be of high enough quality before building applications based on this approach. This work is a strict prerequisite for evaluating the reliability of any simulation accelerated using shared-input parallel Markov chains, and such evaluations are left for future work.

Correlation introduces four important questions: (1) how does the number and size of parallel Markov chains affect correlation among Markov chains, and (2) how long can we run parallel Markov chains in a MISD fashion before we are able to reliably detect non-uniform output? Also, given results from these experiments, (3) how does the resulting performance of AP PRNGs compare to state-of-the-art parallel PRNGs? And (4) how can we modify the AP architecture to increase the performance and quality of PRNG? We simulate parallel Markov chains running on the AP in software and use the TestU01 statistical test battery [5] to evaluate the quality of the resulting pseudo-random output.

The key contributions and results in this paper include:

- A novel method for generating pseudo-random behavior on the AP for PRNG or for other applications that can be framed as probabilistic automata (such as agent-based models) by using MISD emulation of parallel Markov chains via non-deterministic finite automata on Micron's Automata Processor.
- A sensitivity analysis showing that the number of states in a Markov chain has a significant impact on the quality of pseudo-random behavior. An increase in the number of states greatly increases the total possible number of configurations of all considered Markov chains, making it harder for statistical tests to detect correlated behavior.
- A sensitivity analysis showing that the behavior of a large number of parallel Markov chains does not decrease quality of combined random output. However, the output quality is highly dependent on how integers are constructed from random output bits, motivating additional output permutation circuitry in the AP or support processor to reduce correlation among neighboring Markov chains. These evaluations will serve as a guideline for designing future architectures and applications that use this execution model.
- An experiment showing that 571, fair, 8-state Markov chains can consume approximately 1,000,000 symbol inputs before statistical tests can reliably identify non-uniform output. This variable threshold offers a straightforward performance/quality knob for PRNG.
- A performance model showing that with current technology, an AP chip is able to export $4.1GB/s$ of high-quality pseudo-random output, while passing all tests in the BigCrush test-suite. We identify bottlenecks to AP PRNG throughput on the AP architecture, and present an updated performance model considering an AP architecture implemented on a modern transistor technology node and memory specification. With reasonable assumptions, we show that the AP can create $40.5GB/s$ of high-quality random output per chip, using $6.8\times$ less energy than state-of-the-art GPU PRNGs.
- A demonstration that parallel, correlated Markov chains can behave similarly to independent Markov chains. We see this as a necessary first step in future research involving Markov chain simulation on parallel architectures. In particular, this shows promise for invention on the AP, since Markov chains are probabilistic automata.

II. Background

A. Micron's Automata Processor

AP Execution Model: The automata processor architecture consists of a directed graph of state transition elements (STEs), which 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 a STE connected to it via the routing matrix activated on the previous cycle. STEs can be configured to *report on activation*, producing a 1-bit output. This is analogous to accepting an input string in a NFA.

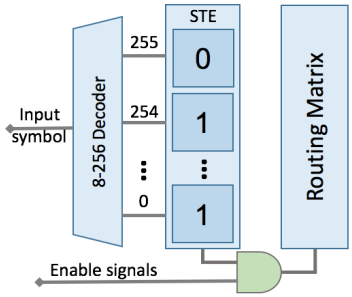

Fig. 1. A simplified STE memory column. 8-bit input symbols are decoded using row-address decoders. If a column reads 1 for an enabled STE, the STE activates and sends its output signal to the routing matrix.

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 recognize. A simplified diagram of an STE is shown in Figure 1.

An STE can recognize an arbitrary character set of possible input symbols on every cycle. If an STE column reads a 1, and the STE is enabled, the STE activates and sends its output signal to the routing matrix. The routing matrix allows STEs to connect to, and enable, any other STEs within the same AP core. Before automata can be run on the AP, they must be compiled (placed and routed) and loaded onto the AP fabric.

Columns of STEs are organized into *blocks* and a set of blocks makes up an AP *core*. In the proposed first generation AP chip architecture, a block contains 256 STEs, 32 of which can report. AP cores contain 96 blocks, offering a total of 24,576 STEs per core. AP chips contain two disjoint cores. The first generation AP chip operates at a constant frequency of $133MHz$, consuming one symbol every $7.5ns$, thus providing an input symbol stream throughput of $133MB/s$ per chip. First-generation AP chips connect to the system via a shared DDR3 interface and have a TDP of about $4W$.

When an STE on an AP chip reports, the AP generates a *report vector*. Each report vector is a bit-vector representation of all reporting STEs that activated at that particular cycle, and contains up to $1,024$ bits. Each first generation AP chip contains 6 reporting regions capable of exporting $1,024$ output vectors in $1.8ms$. Therefore a best-case upper-bound for the full AP output throughput is $436.9MB/s$ per chip.

The above metrics characterize the proposed first-generation AP architecture and implementation. Future AP architectures may allow much lower AP reconfiguration times and much higher output throughput. We explore these potential improvements in later sections.

B. Pseudo-random Number Generation

Pseudo-random number generation (PRNG) lies at the core of simulation and cryptographic applications. For example, Monte Carlo methods are pervasive simulation tools in physical and social sciences and rely on continuous random sampling to drive simulation of unpredictable processes. Because fast and high-quality pseudo-random number generation is on the critical path of these applications, developing fast and high-quality PRNGs is of the utmost importance to improving the quality and speed of any computational science.

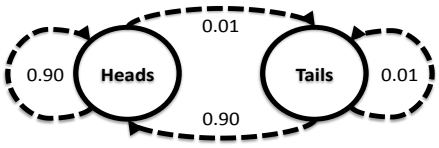

Fig. 2. A simple Markov chain that simulates an unfair coin toss with two states: *Heads*, and *Tails*. Transition probabilities between these states are *unfair*, i.e. the probability of transitioning to *Heads* is different than *Tails*.

Today, while there are many PRNG algorithms, not all are created equal. No matter the method, the harder it is to distinguish pseudo-random output from a truly random output, the better it represents a truly random number stream. The literature has adopted two avenues for evaluating the quality of PRNGs. Cryptographic applications require a PRNG to reduce to a hardness assumption, some problem widely believed intractable. This paper leaves such an analysis of AP PRNG to future work. Instead, because we aim to motivate the use of AP PRNG for Monte Carlo simulation, the literature suggests we evaluate empirically using statistical tests.

Statistical tests distinguish random from pseudo-random input by searching for over-prevalent or under-prevalent patterns. The most comprehensive and stringent collection tests are the BigCrush test battery from TestU01 suite [5], which includes the functionality of the Knuth tests [6], DIEHARD [7], and the NIST statistical test suite [8]. A test in the suite fails if it identifies a property of the pseudo-random sequence that should not exist in true randomness. If all tests pass, the pseudo-random numbers have been deemed indistinguishable from true randomness.

C. State-of-the-Art Parallel PRNG Algorithms

Mersenne Twister [9] is an extremely popular PRNG algorithm used pervasively in Monte Carlo simulations. However, it is famously difficult to port onto smaller core architectures (e.g. GPU, XeonPhi) because it involves a large amount of state per thread [10]. Not only is Mersenne Twister hard to parallelize, it also fails many tests in TestU01's BigCrush test battery [10], and is thus a dubious choice as the gold standard PRNG for future high-performance scientific simulation.

Philox [11] is a newer parallel PRNG algorithm that drastically reduces the state requirements per parallel thread, and relies on integer computation giving it impressive performance on massively parallel SIMD GPU hardware–$145GB/s$ onboard PRNG on an NVidia GTX580 [11]. Philox is trivially parallelizable, and the fastest PRNG algorithm known (both serially and in parallel) to pass all tests in the BigCrush test suites. Thus, we consider this algorithm the current state-of-the-art in both performance and quality.

III. Implementing Markov Chains on the AP

In informal terms, Markov chains are automata with probabilistic transitions between states. An example Markov Chain describing an unfair coin is illustrated in Figure 2.

Markov chains are defined by stochastic *transition matrices*, which hold all transition probabilities from a start state (row) to a end state (column). The transition matrix for the unfair coin example in Figure 2 is shown in Table I.

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

TABLE I
Stochastic transition matrix of a Markov chain representing an unfair coin.

A. Mapping Markov Chains to the AP

To easily communicate the concept of probabilistic transitions and our implementation of Markov Chains on the AP, we first begin with a simple-to-understand construction that clearly illustrates the mapping and algorithm.

Construction: Consider the unfair coin example described in the previous section and shown in Figure 2. To produce the probabilistic transitions of the Markov chain in the automaton we first assume the input symbol stream is a source of uniformly distributed random symbols. Each Markov chain can be constructed using the following algorithm 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

```
INITIALIZATION;
foreach FromState do
    Create a reporting STE "state node" representing
     FromState that recognizes all input symbols;
Select an arbitrary FromState to be start state, activating
 on start of data;
CONSTRUCTION;
foreach FromState do
    foreach ToState do
        Create "edge node" STE EdgeNode;
        TransProb ← StochMat[FromState][ToState];
        Without replacement, randomly select
         TransProb * |Σ| symbols from Σ as the character
         class recognized by EdgeNode;
        Add edge from FromState to EdgeNode;
        Add edge from EdgeNode to ToState;
```

An example of this construction for the unfair coin example is shown in Figure 3. For illustrative purposes, we restrict the input symbols to be within the character class $[0-9]$. The proportion of symbols in each EdgeNode corresponds to the transition probability.

Note that this construction takes two cycles to generate an output, one to move from a "from state" node to an edge node, and another to move from an edge node to a "to state" node. Algorithm 1 can easily be modified to generate an output on every cycle by also setting a randomly selected edge node, along with an arbitrary state node, to act as a start state. We omit this construction for clarity but assume one random output per cycle when modeling performance of Markov chains on AP hardware.

We leave research and evaluation of other constructions as future work. Markov chains can also be used to construct probabilistic automata that model more complex systems. We

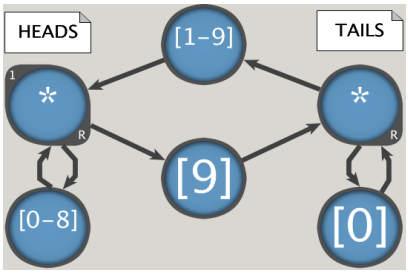

Fig. 3. A Markov chain implemented on the AP corresponding to the theoretical Markov chain in Figure 2, with two reporting "state nodes" representing *Heads* and *Tails*. Reporting states are indicated by an "R" subscript. The start state is indicated by a "1" superscript. Transition probabilities between these states are *unfair* and are modeled by dividing the possible input symbols $[0-9]$ into random groups, proportional to the transition probabilities.

leave evaluation of these more complex systems, such as agent agent-based simulations, to future work.

IV. Efficient and High-Quality PRNG on the AP

Using Parallel Markov chains to increase throughput: To construct a PRNG from a single Markov chain, we first build a fair Markov chain of a certain number of states. The number of states in the Markov chain should be any power of two, ensuring a uniform distribution of 0's and 1's in the output bit stream. On every cycle, a single chain will report which state it randomly transitioned to, thus emitting $\log_2(states)$ bits of random output per Markov chain per cycle.

By adding more Markov chains, and interleaving their output, we can increase the total amount of pseudo-random output relative to the input symbols used to drive random transitions. A single 2-state Markov chain only emits a single random bit per random input byte. Eight 2-state chains therefore create as much pseudo-random output as input.

Because only 32 STEs out of 256 in an AP block are capable of reporting in the first generation AP architecture [1], each Markov chain may be limited by either reporting elements or total STEs per block. An N-state fair chain requires N reporting STEs, thus we can instantiate a maximum of 16, 8, and 4 chains per block for 2, 4, and 8-state chains respectively. A N-state fair chain requires N^2+N STEs, thus, based on STE capacity, we can instantiate a maximum of 42, 12, and 3 chains per block for 2, 4, and 8-state chains respectively.

Assuming that each reporting element can be used to create random output, and assuming no chip output reporting bottlenecks, the AP can create a staggering $51GB/s$ of pseudo-random output. However, correlation among Markov chains and practical output throughput bottlenecks prevents AP PRNG from reaching this theoretical upper-bound. Below we discuss the source of inter-chain correlation, and potential mitigation techniques. Later sections propose solutions to relax the AP's output throughput bottlenecks.

Reducing Correlation Among Markov Chains: Parallel Markov chains may produce output that initially looks random, but over time, certain output configurations *must* appear more often than others, and some configurations will provably never appear. Thus, after a certain number of input symbols, the patterns will emerge in the output. We must periodically ask the host processor to randomize the transition tables of the Markov chains, reconfiguring all transition STEs on the device.

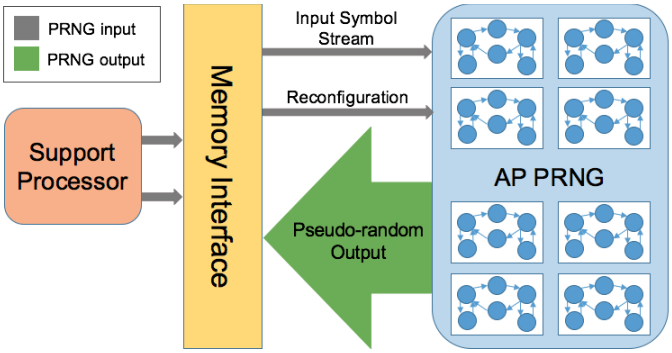

Fig. 4. AP PRNG system-level diagram. A support processor provides a small amount of pseudo-random input to drive transitions and reconfiguration.

In order to evaluate whether or not the AP and the above algorithm can be used as a practical PRNG accelerator, we must answer the following questions:

1) How does the number of states in each Markov chain effect the quality of random output? Because AP reconfiguration may be expensive, we favor chains that produce higher quality output as they will postpone mandatory reconfiguration.
2) How does the number of parallel Markov chains effect the quality of random output? If increasing the number of chains reduces random quality, and increases reconfiguration frequency, using a large amount of on-chip resources may hurt performance.
3) What is the shortest reconfiguration threshold, i.e. the number of input symbols that can be consumed by the best performing configuration before we can identify non-random behavior, based on the best performing parameters identified by the previous questions? A shorter reconfiguration threshold, will cause the algorithm to spend more time building new transition tables and reconfiguring the AP.
4) Can we increase random quality with small changes to the AP architecture? By how much will these changes allow us to increase the reconfiguration threshold while still passing all tests in BigCrush?
5) What is the performance of AP PRNG on the proposed first generation AP hardware compared to current state-of-the-art PRNGs? What is the performance of AP PRNG on a plausible next generation AP implementation, assuming adjustments to the AP architecture, and projected implementation on a modern transistor technology node and memory specification?

V. Experimental Framework

To test AP PRNG performance and quality, we implemented Algorithm 1 in C++ and used the test batteries in the TestU01 statistical test suite [5] to assess quality of random output. Because first-generation AP hardware is not yet available, this is a simulation of AP PRNG behavior based on Markov chain automata developed and verified in Micron's AP SDK. A high-level diagram of the modeled system is shown in Figure 4.

TestU01 is made of three main test batteries: SmallCrush, Crush, and BigCrush. SmallCrush consists of 10 statistical tests, and is meant to quickly check obvious statistical flaws in PRNG output. Crush applies 96 distinct statistical tests (144

total test statistics), while BigCrush applies 106 distinct tests (160 total test statistics). Because production AP hardware is not currently available, the BigCrush test battery can take between 3-7 days to complete on a single CPU core, depending on how often simulated AP reconfiguration is required. We therefore do initial sensitivity analyses using SmallCrush, and Crush, in order to quickly identify trends in relative quality of random output between different AP PRNG configurations. Once we identify a suspected proxy configuration for the best-performing AP PRNG, we use the full BigCrush test battery, the most comprehensive, and exhaustive test-suite available [5], [10], to identify reconfiguration thresholds that provide good random quality. We then use the derived AP PRNG parameters and input thresholds as input to a performance model and compare to state-of-the-art PRNGs.

Because AP PRNG requires a host processor to supply random input to the hypothetical AP, and to randomly configure Markov chain transition tables, we must provide a source of pseudo-random numbers that simulates the role of an AP PRNG host processor. For this evaluation we use the *Philox*32*x*4_10 generator [11] to provide all random input. As one should expect for any PRNG, we found that using lower quality input sources, such as the C standard library's rand() or Mersenne Twister, translated to significantly lower quality AP PRNG output. We therefore use the Philox algorithm, as it is the most performant and highest quality generator available, and available as an open source C++ library [12]. We use this library implementation to drive all random configuration and streaming input to the AP PRNG. We performed all evaluations of random quality on a cluster of Intel i7-4820k CPUs operating at 3.7GHz with 64 logical cores.

VI. Sensitivity Analyses

A. Effect of Markov Chain Size on Random Quality

To show how randomness quality is affected by Markov chains of different sizes, we run multiple trials of SmallCrush using the AP PRNG simulator. Although SmallCrush may not be the most stringent test suite, and thus less useful when comparing to state-of-the-art PRNGs, it is suitable to quickly identify relative patterns in failures among different AP PRNGs, and motivate appropriate parameters for more stringent tests.

We configured AP PRNG with a reconfiguration threshold of 50,000 input symbols, 384 parallel Markov chains, each with 2, 4, or 8 states. For each state configuration, we ran 16 trials of SmallCrush and collected data on all test failures. The results are shown in Table II.

Number of Markov Chain States	2	4	8
Average Number of Failures	5.5	2	0
Distinct Number of Failures	6	2	0

TABLE II
IT IS STATISTICALLY HARDER TO IDENTIFY CORRELATION BETWEEN CHAINS WITH MORE STATES.

We can see that random quality is highly sensitive to the number of states used to build each Markov chain. 2-state Markov chains fail an average of 5.5 tests over the 16 trials, failing 6 unique tests in the test suite. 4-state Markov chains consistently fail the same two tests over the 16 trials. 8-state Markov chains are completely resistant to failure over the 16 trials, passing all tests. A larger number of states allows for a larger number of possible configurations of a single Markov chain, and therefore a single Markov chain with 8 states will take much longer to repeat states and exhibit statistically identifiable non-random behavior than a chain with 2 states. When considering configurations of states of a set of Markov chains, this effect is compounded, as it takes much longer to identify which configurations occur too frequently or infrequently as compared to random.

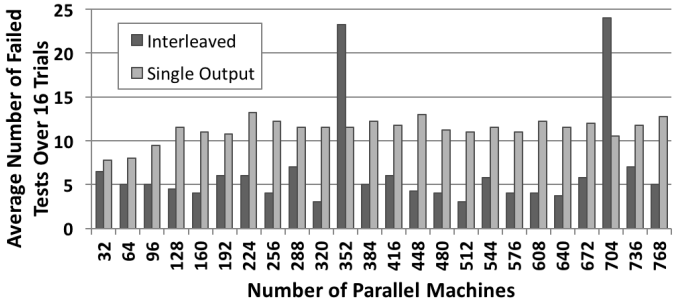

Fig. 5. Average number of Crush failures over four trials for parallel 8-state Markov chains with a reconfiguration threshold of 200,000. The darker bars represent failure rates when interleaving output bits. Spikes in failure rates occur when the same Markov chains always contribute to the same bits in output integers. The lighter bars represent failure rates when successive output from a Markov chain contributes to a single output integer. This eliminates the spike in failures, but reduced overall performance.

B. Effect of Parallel Markov Chains on Random Quality

To show how the number of parallel, 8-state Markov chains affects the random quality produced, we explore performance of AP PRNG with from 32 to 768 Markov chains operating in parallel. For these experiments, we use Crush to evaluate random quality. Figure 5 summarizes the experimental results. Apart from two obvious outliers, when using 352 and 704 parallel Markov chains, as the number of parallel Markov chains increases, quality of randomness is stable, suggesting that practically, larger numbers of parallel Markov chains do not significantly decrease quality of random output.

We hypothesized that the marked increases in the failure rate for 352 (32×11) and 704 (64×11) parallel Markov chains was due to an undesirable property of our algorithm for converting output bits from AP PRNG to 32-bit integers. Because the value of an integer is influenced more by high-significance bits, non-random behavior in high-significance bits will be noticed quicker than non-random behavior in low-significance bits on the numeric-based tests.

Because algorithm 1 constructs 32-bit integers by concatenating three output bits from every 8-state chain in a round-robin fashion, certain numbers of chains will align their output, such that a single chain will always contribute to the same digits. This puts undue reliance on chains that contribute highly significant bits of integers, and may amplify patterns in the psuedo-random output. To test this hypothesis, we modified Algorithm 1 to have a Markov chain provide all the bits for a single 32-bit value before the value is consumed by the statistical tests. Thus, all Markov chains contribute uniformly to the digits in a single integer, rather than a particular set of digits. The results of this experiment are shown in Figure 5.

The spikes in failure rates are eliminated, although failure rates are higher on average, supporting our hypothesis. We

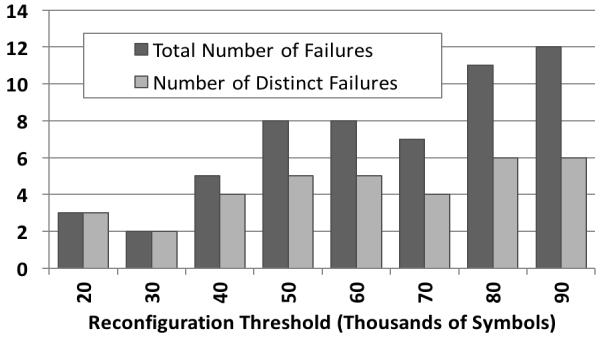

Fig. 6. As the reconfiguration threshold increases, it is becomes easier for statistical tests to identify non-random behavior.

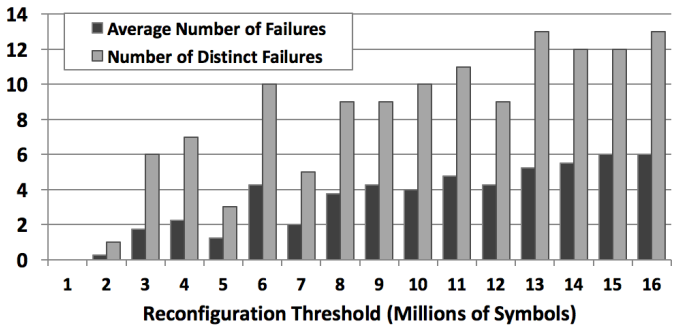

Fig. 7. Output quality of AP PRNG with output permutation hardware greatly increases quality of random output. AP PRNG passes all tests in BigCrush with a reconfiguration threshold of at least $1,000,000$

therefore adjust Algorithm 1 to use the largest prime number of Markov chains able to fit onto an AP chip. A prime number of machines will prevent any one machine from contributing to the same location in an supplied 32-bit integer at least until we provide 32 times that prime number of input symbols. The maximum number of 8-state Markov chains that will fit onto an AP chip is 576; we therefore use 571 (the largest prime less than 576) 8-state chains per AP chip as the highest performing configuration for final quality and performance evaluations.

C. Effect of Input Size on Random Quality

To show how the quality of random output is affected by the number of input symbols, we ran four trials of 761 8-state Markov chains through the Crush test suite, varying the number of input symbols from 20,000 to 90,000. We hypothesized that the more symbols we allow each parallel machine to consume without reconfiguration, the more likely the output is to look non-random. Therefore, *shorter* reconfiguration thresholds will likely increase quality of random output. However, shorter reconfiguration thresholds force the AP to reconfigure more often, incurring a reconfiguration penalty. The results of the experiment are shown in Figure 6.

Figure 6 shows that, as we increase the number of input symbols between reconfigurations, the quality of random output decreases. However, even for 20,000 symbols, 761, 8-state Markov chains do not consistently pass all Crush tests. In initial exploratory tests, some BigCrush tests even failed with reconfiguration thresholds as low as 10,000, meaning that even shorter reconfiguration thresholds are required in order to match the quality of the Philox algorithm. As the AP only takes $7.5ns$ to consume a symbol, but approximately $45ms$ to reconfigure [2], a reconfiguration threshold of 10,000 would cause the AP to spend 99.83% of its time reconfiguring, translating to $48MB/s$ of output.

We observe that performance of AP PRNG could be drastically increased if we could reduce the impact of *neighbor dependence*. Because our algorithm always forms integers by interleaving bits of neighboring Markov chains in a deterministic manner, we put undue pressure on close groups of machines to produce uncorrelated output. Previously, we saw that neighbor dependence induced catastrophic failure rates when the same configuration of machines was always used to contribute the same digits of output integers. By using a prime number of machines, we could force this configuration to at least rotate. However, rotation still preserves the relative order of machines contributing to any individual output integer. To eliminate neighbor dependence without decreasing throughput, we investigate potential impact of output permutation via pipelined support hardware or software to reorder output bits of a group of Markov chains before we combine the bits into integer values for analysis.

We configured AP PRNG to use 571 8-state Markov chains with 32-wide permutation hardware that changes the output ordering of every 32 Markov chains every $1,000$ symbol cycles. We then ran four trials of BigCrush to assess improvements in quality of random output. Results are shown in Figure 7. Compared to the results from Figure 6, random output quality greatly improved. Previously, AP PRNG failed all tests in Crush with a reconfiguration threshold of $20,000$ input symbols. However, AP PRNG with permutation capability passes all BigCrush tests with a reconfiguration threshold of at least $1,000,000$ symbols. This indicates that neighbor dependence was a major contributor to poor output quality. AP PRNG with permutation capability can generate at least $50\times$ more random output before full reconfiguration is required.

While this functionality is simulated in software, we hypothesize its implementation as an addition to the reporting architecture, support ASIC, or via the host or support processor. The most efficient implementation will depend on the particular deployment scenario of the AP. Output permutation is only one way of increasing quality of output, and this result motivates further studies to identify other methods (especially to replace the simple round-robin scheme) to cheaply mitigate the effects of neighbor dependence in hardware, or software.

VII. AP PRNG Performance Model

The above sensitivity analyses motivate using 571, 8-state Markov chains, with a reconfiguration threshold of at least $1,000,000$ input symbols, and a permutation threshold of $1,000$ as a high-quality PRNG. To analyze the practical performance of this AP PRNG configuration, we constructed a performance model based on these parameters and the reported configuration of the first-generation AP architecture [1]. Because the AP operates at $133MHz$, consuming one 8-bit symbol and executing all transition rules every $7.5ns$ cycle, an AP performance model does not require simulation. Below we describe the model inputs, output metrics, and sensitivity to certain architectural parameters.

First Generation AP Architectural Parameters	
Frequency	133MHz
Cycle Time (T_c)	7.5ns
STE Size	256 bits
Random State per Chip ($ChipState$)	1.17MB
Est. AP Reconfiguration Time (T_r)	45ms

AP PRNG Parameters	
States per Markov chain (N_s)	8
Markov chains per AP Chip (N_{mc})	571
Input Reconfiguration Threshold (I_R)	1,000,000
Permutation Width (P_W)	32
Permutation Reconfiguration Threshold (P_R)	1,000

AP PRNG Performance Model	
Chip Output per Input Symbol (O)	$\log_2(N_s) * N_{mc}$
Random Generation Time (T_R)	$I_R * T_c$
Runs per second ($Runs$)	$1/(T_{Run} + T_r)$
AP PRNG Throughput (P)	$Runs * O$
Random Input Stream Rate (In_s)	$Runs * I_R$
Random Input Required for Reconfiguration (In_r)	$Runs * ChipState$
Random Input Required per Permutation (In_p)	$P_W log_2(P_W)$

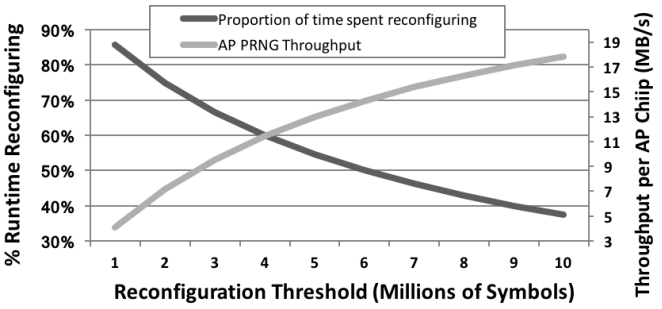

Fig. 8. Percentage of runtime spent reconfiguring vs. AP PRNG throughput with different reconfiguration thresholds. Performance increases dramatically if AP PRNG is able to reconfigure less frequently.

A. Performance Sensitivity to Reconfiguration Threshold

Figure 8 shows the throughput of AP PRNG predicted by the performance model for various reconfiguration thresholds (I_R). When $I_R = 1,000,000$, AP PRNG produces $4.1GB/s$ of random output per proposed first-gen AP chip.

One unique feature of AP PRNG is that it allows the user to easily trade random quality for higher random throughput. This is desirable if an application or simulation does not require extremely high-quality randomness and is constrained by power, or performance. If strict random quality is not required, and the user allows for a longer I_R, the model predicts that performance can be increased dramatically. Figure 8 shows that for an $I_R = 10,000,000$, a single first-gen AP chip is capable of producing $17.8GB/s$ of random output. End users will therefore be able to reduce the number of AP chips, and power consumption, or increase performance, if statistically perfect randomness is not required for a particular application.

AP PRNG also requires a source of random input to drive automata transitions, reconfiguration, and permutation. While we assume the system host processor is able to implement the $Philox32 \times 4\_10$ algorithm and provide this random input, the resulting need for random input throughput can be significant. For $I_R = 1,000,000$, the model predicts we need $200.7MB/s$ of random input per chip. For $I_R = 10,000,000$, the model predicts we need $176.3MB/s$ of random input per chip.

Memory Technology	**DDR3**	**DDR4**	**HMC 2.0**
Peak Throughput (GB/s)	12.8	17.0	320
T_r (μs)	91.4	68.8	7.3
AP Chip Output (GB/s)	28.2	28.3	28.5
Throughput Limited Output (GB/s)	12.8	17.0	28.5

TABLE III
AP PRNG PERFORMANCE MODELED ON DIFFERENT MEMORY TECHNOLOGIES. AP PRNG THROUGHPUT IS LIMITED BY PEAK MEMORY THROUGHPUT FOR DDR3 AND DDR4 TECHNOLOGIES.

B. Performance on Future AP Hardware

On the first generation AP chip hardware, maximum output throughput is projected to be $436.9MB/s$ [13], thus limiting the practical amount of random output that can be exported off chip. STE reconfiguration time, which we have shown to be the next most significant performance bottleneck, is projected to be $45ms$ on first-generation AP hardware. While these parameters represent the projected performance of the first-generation AP chip architecture and implementation, they do not reflect fundamental bottlenecks to AP PRNG. For example, output report vectors and STE columns are implemented as DRAM memory; therefore, it is not unreasonable to assume that both STE reconfiguration and output reporting could happen at or near native memory I/O speeds, drastically decreasing reconfiguration times and increasing practical AP PRNG output throughput.

We consider the performance of AP PRNG where writes could occur at native DDR3, DDR4 and Hybrid Memory Cube (HMC) throughputs. HMC technology accomplishes massive I/O throughput by stacking DRAM layers directly on top of logic, and inserting vertical communication links with through-silicon vias [14]. One could integrate the AP into HMC as one or many layers of a stacked HMC design as a part of a heterogeneous memory module. Table III shows the performance of DDR3, DDR4, and HMC2.0 technologies with derived reconfiguration times (T_R), and AP PRNG throughput.

For 571, 8-state Markov chains we only need to reconfigure all transition STEs (64 for 8-state chains). For a single chip, this translates to $1.17MB$ of state. Table III shows that even if we reconfigure the AP using native DDR3 bandwidth, the AP PRNG performance model predicts each AP chip can produce $28.2GB/s$ of high-quality pseudo-random output. However, peak DDR3 throughput is only $12.8GB/s$, and thus limits the practical amount of output we can export off chip.

If we consider future high-throughput memory technologies, such as Hybrid Memory Cube (HMC) [14], the AP PRNG performance model predicts that each AP chip will produce a comparable $28.33GB/s$ of pseudo-random output. However, HMC's much larger output bandwidth allows us to easily export this output off chip.

As we increase the number of reconfigurations per second, AP PRNG also requires more random input throughput. The HMC configuration, with $T_R = 1,000,000$, requires $289MB/s$ of random input per chip, about $7\times$ larger than AP PRNG on the first-generation AP chip. While this is not an insignificant amount of random input, the host processor can easily supply it. The significant amount of state needed for STE reconfiguration motivates research into techniques to increase T_R (such as output permutation) without sacrificing random quality.

Future AP architectures implemented on cutting-edge transistor process nodes will also most likely have larger STE ca-

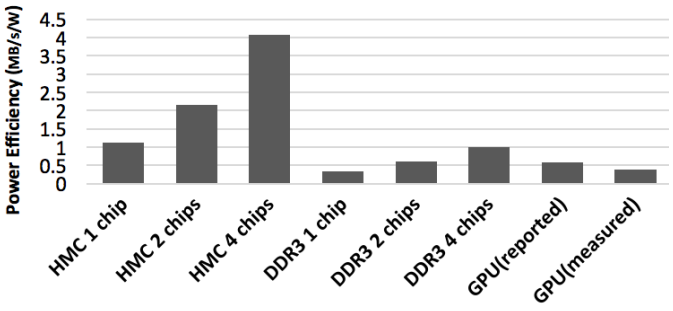

Fig. 9. AP PRNG is up to 6.8× more power efficient than the highest-throughput reported GPU PRNG depending on the deployment scenario.

pacities. Adjusting the model for a reasonable 1.41× increase in STE capacity (corresponding to 1.41× more Markov chains) per AP core, AP PRNG can produce $40.5GB/s$ of random output per chip, while only requiring $355MB/s$ of random input from the host processor to configure transition tables, drive automata transitions, and drive permutation reconfiguration.

C. Estimating AP PRNG Power Advantage

Ultimately, performance and power advantages over current PRNG implementations will greatly depend on the implementation and deployment scenario of AP PRNG or other applications of Markov chains such as discrete event simulations, however, we project that AP PRNG will be much more power efficient than GPU PRNG and Markov-chain based discrete-event simulations. For example, the GTX 580 GPU used in Salmon et.al [11] has a TDP of 244W, while each DDR3-based AP chip has a projected TDP of 4W, and stacked HMC-based memories are projected to use 70% less energy than DDR3. Figure 9 shows the PRNG efficiency of a few different realistic AP PRNG deployment scenarios.

All AP deployment scenarios require a support processor to generate random input and configure the AP. We assume that a typical single CPU support processor core consumes 35W. The configuration with 4 AP chips implemented in an HMC technology produces 4MB/s/W, 6.8× more power efficient than the best performing GPU PRNG reported in the literature [11], and 10.8× more power efficient than our measured experiments using the curand library implementation of *Philox*32*x*4_10 on an NVidia K20C GPU. Disregarding support processor power consumption, and conservatively assuming a 4W TDP per AP chip, AP chips are 16.8× more power efficient than the reported GPU implementation.

VIII. Conclusions and Future Work

This work explored using Markov chains emulated on Micron's Automata Processor (AP) as an efficient, high-quality, and high-performance source of pseudo-random behavior. We first showed that the AP can emulate Markov chains, which can then be used for pseudo-random number generation, or other on-chip probabilistic computation such as agent-based simulation. However, many parallel NFA-based Markov chains driven by a single source of random input are inherently correlated, and may reduce the quality of psuedo-randomness or simulation results.

To test the quality of random behavior from correlated Markov chains we used the most stringent statistical test suite available (TestU01) and showed that the proposed first-generation AP chip is capable of producing $4.1GB/s$ of high quality (indistinguishable from random) pseudo-random output with implementations on current memory standards and an older transistor process node. If using imminent high-bandwidth memory specifications, such as Micron's hybrid memory cube (HMC), and a state-of-the-art process technology node, the AP architecture is capable of producing upwards of $40.5GB/s$ of high-quality pseudo-random output per chip.

Our experiments show that MISD, non-von Neumann, memory-based architectures, such as the Automata Processor, are promising as fast and efficient accelerators for pseudo-random number generation and Markov chain based simulation, and will offer exciting performance and power advantages over purely parallel von Neumann PRNG for applications in future heterogeneous systems. Motivated by the high-quality of output from Markov chains over many thousands of input symbols, future work will focus evaluating performance and power benefits of on-chip agent-based simulations such as SI/SIS agent-based epidemiology models.

IX. Acknowledgments

This work was partly funded by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, Achievement Rewards for College Scientists (ARCS), Micron Technologies, and NSF grant no. EF-1124931.

References

[1] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, p. 1, 2014.

[2] K. Wang, Y. Qi, J. J. Fox, M. R. Stan, and K. Skadron, "Association rule mining with the micron automata processor," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 689–699, IEEE, 2015.

[3] K. Zhou, J. Fox, K. Wang, D. Brown, and K. Skadron, "Brill tagging on the micron automata processor," in *Semantic Computing (ICSC), 2015 IEEE International Conference on*, pp. 236–239, Feb 2015.

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

[5] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, Aug. 2007.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[7] G. Marsaglia, "Diehard: a battery of tests of randomness," *See http://stat. fsu. edu/ geo/diehard. html*, 1996.

[8] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, S. Leigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, A. Rukhin, J. Soto, M. Smid, S. Leigh, M. Vangel, A. Heckert, J. Dray, and L. E. B. III, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," 2001.

[9] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

[10] M. Manssen, M. Weigel, and A. K. Hartmann, "Random number generators for massively parallel simulations on GPU," *The European Physical Journal-Special Topics*, vol. 210, no. 1, pp. 53–71, 2012.

[11] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: as easy as 1, 2, 3," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12, IEEE, 2011.

[12] "Random123." http://www.deshawresearch.com/resources_random123.html.

[13] "Designing for the Micron D480 Automata Processor." http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html.

[14] "Hybrid memory cube specification 2.0." http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.0_Public.pdf.