Highly Parallel Seedless Random Number Generation from Arbitrary Thread Schedule Reconstruction

Eryn Aguilar[1], Jevis Dancel[1], Deysaree Mamaud[1], Dorothy Pirosch[1], Farin Tavacoli[1], Felix Zhan[1],
Robbie Pearce[2], Margaret Novack[2], Hokunani Keehu[2]
Benjamin Lowe[3], Justin Zhan[3], Laxmi Gewali[1], Paul Oh[2]
[1]*UNITE/University of Nevada, Las Vegas*
[2]RET/University of Nevada, Las Vegas
[3] University of Arkansas

***Abstract*—Security is a universal concern across a multitude of sectors involved in the transfer and storage of computerized data. In the realm of cryptography, random number generators (RNGs) are integral to the creation of encryption keys that protect private data, and the production of uniform probability outcomes is a revenue source for certain enterprises (most notably the casino industry). Arbitrary thread schedule reconstruction of compare-and-swap operations is used to generate input traces for the Blum-Elias algorithm as a method for constructing random sequences, provided the compare-and-swap operations avoid cache locality. Threads accessing shared memory at the memory controller is a true random source which can be polled indirectly through our algorithm with unlimited parallelism. A theoretical and experimental analysis of the observation and reconstruction algorithm are considered. The quality of the random number generator is experimentally analyzed using two standard test suites, DieHarder and ENT, on three data sets.**

***Index Terms*—random number generation, multiprocessor, threading, PRNG, TRNG**

I. Introduction

In the fields of cryptography and computer science, random number generation is an important method of securing, containing and protecting data. Generating random numbers is a key component in modern cryptography. Pseudo Random Number Generators (PRNG) for computing random numbers are inherently deterministic; the sequence can be completely determined by knowledge of the initialization vector or seed. For cryptographic purposes, the initialization vector is constructed by sampling a true random number generator and it is resampled frequently.

Several methods have been proposed for true random number generators, often sampling highly dynamic and chaotic natural processes. Some methods include obtaining random data through the chaos of lava lamps [1], using atmospheric noise in order to capture random data [2], and the device drive on Linux, /dev/random [3]; however, these methods are slow. Methods have been proposed to capture randomness occurring inside the computer such as measuring fluctuation in CPU jitter [4], and memory access times at the memory controller [5].

A solution was proposed by Antonaidis et al., to indirectly poll the memory accesses at the memory controller through software. [6] Antoniadis et al., show this solution is two to four orders of magnitude faster than other direct methods [6]. In their approach, called Co-RNG, Antoniadis et al., reconstruct the read and write history of two threads accessing two shared variables. They call this the schedule reconstruction problem [6]. The reconstruction of read-write history in [6] only supports two threads, making it a two thread schedule reconstruction problem. With only two interleaving threads, the chance of repeated read-write histories where the two threads execute in lock step is increased. The length of history is determined by the number of reads and writes these threads complete. Therefore, a longer history requires more read and write operations.

This research proposes CAS-RNG, a random number generator which makes use of the compare-and-swap instruction. The new approach simplifies the previous approach, generalizes to allow an aribitrary number of threads to execute in parallel, and maintains a high throughput and good quality randomness. In this research, the two-thread reconstruction is modified to a N thread schedule reconstruction problem, reconstructing the reads and writes of N concurrent threads. The reconstructed history is used to generate random numbers by using the history as an input to the Blum-Elias algorithm, similarly to Antoniadis, et al. [6]. The possibility of repeated histories is reduced by having N parallel threads with more opportunities for interleaving. This method constructs a longer history in the same number of rounds. The validity of the random number generator created will be tested using two test suites: ENT and Dieharder.

The rest of this paper is structured as follows: Section II discusses the related work and prerequisite material, Section III discusses our generalized algorithm for n schedule reconstruction, Section IV contains our experimental analysis of the randomness of our approach, and Section V summarizes the experiment and results with suggestions for future research.

978-1-7281-4607-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ICBK.2019.00009

II. Related Work and Preliminaries

A. Related Work

True random number generators generate random numbers from physical processes. This process is in direct contrast to pseudo-random number generators, which rely on algorithms and seeds. In some cases, the random source can exist within the computing hardware. In a study by Müller [4], frequencies in the processor and RAM are used to create random bits. The Müller experiment [4] uses memory access time as explained by Agafin [5] to create a more efficient random source. For this research, multi-core processors help to retrieve the true random source. By utilizing multiple cores, an increased number of threads are used to collect the random sequence. This use of parallelism increases speed and reduces the time to create the sequence from shared memory.

It has been shown by Agafin and Krasnopevtsev that memory accesses at the memory controller are a source of true randomness [5]. The reads and writes to memory form a truly random sequence, which can be modeled as a stochastic process X. However, the probability of the next element in the sequence given the proceeding sequence is unknown. To illustrate, $\Pr(X_i = j_i | X_{i-1} = j_{i-1}, X_{i-2} = j_{i-2}, \ldots, X_{i-k} = j_{i-k})$ is unknown. Fortunately, due to the work of Antonaidis, et al. in Co-RNG, the stochastic process X modeling the memory accesses at the memory controller has the first order Markov property [6]. Therefore, we capture the relevant information in X by a new stochastic process Y with the first order Markov assumption, that is $\Pr(X_i = j_i | X_{i-1} = j_{i-1}, X_{i-2} = j_{i-2}, \ldots, X_{i-k} = j_{i-k}) = \Pr(Y_i = w_i | Y_{i-1} = w_{i-1})$. The probability remains unknown, however, there exist algorithms to extract unbiased randomness from the Markov chain Y.

One such algorithm is the Blum-Elias algorithm. Consider a sequence $S = s_1, s_2, s_3, \ldots, s_n$ of memory accesses, which correspond to a path in the Markov chain Y. Elias proposed to partition the possible output sequences S into classes, for which each class has equal probability [7]. The output of the Elias function $l = f_E(S)$ is a label $l \in L$ which occurs with equal probability, $\forall l \in L, \Pr(f_E(S) = l) = \frac{1}{|L|}$.

Time efficiency and data accuracy become issues with multithreading in concurrent programming. Problems arise due to the readers and writers accessing the same shared memory at the same time [8]. In a study on concurrent programming [9], mutual exclusion is implemented to ensure the shared memory is not corrupted by being accessed multiple times. The shared memory, or shared resources, are known as critical sections; mutual exclusion ensures that only one process occurs at a time within the critical sections. Locking is essential to critical resource access. When a critical section is being used to complete a task, the critical section is locked so other processes cannot access it at the same time. When the task is completed, the critical section is unlocked, allowing another process to access the critical section. The flaw in mutual exclusion is that concurrency is limited, and processes become sequential. Processes must wait to use the critical sections until they are unlocked. Wait times can vary and may lead to processes never accessing the critical section at all. In some cases, processes may run for a long time or not run at all, leading to a deadlock. However, Peterson [10] indicates alternate solutions to the read-write problems that do not involve the concept of mutual exclusion are possible.

Wait free synchronization is a strong non-blocking mechanism providing guaranteed progress to all correct processes [11]. This synchronization allows the readers and writers to access the shared memory without mutual exclusion. For concurrent data structures, a wait-free approach guarantees any process can complete any operation in a finite number of steps. A wait-free implementation of an object can be built out of any object with the same or greater consensus number [12]. A universal constructor using an object of infinite consensus number, compare-and-swap, as a universal primitive, can be used to implement any wait-free object [12]. We verify correctness by showing that the implementation is linearizable.

An atomic snapshot can be constructed from atomic read and write registers using the clean double collect method [12]. This method takes two sets of atomic reads of the shared memory and compares them to construct the snapshot. The atomic snapshot reads the observations in multiple iterations and analyzes the pairs of read-write operations. The pairs are checked to see if they are identical; if so, the snapshot is complete. If the reads do not match in the first cycle, the process repeats until the collections match, which can lead to long delays and starvation. A helping mechanism can be implemented to create a wait free atomic snapshot using clean double collects [12]. While in theory, clean double collect allows all the observations to be read, in practice it is not optimal and can allow for many indistinguishable observations which complicates the reconstruction.

Verifying the quality of a random source is a difficult task because only the output sequences are tested, rather than the source itself. Therefore, the most common approach is to test very large output sequences consisting of millions of random bits. If random number generator is of poor quality, then the resulting sequences should have detectable patterns. The probability of a pattern emerging in a sequence increases proportionally to the length of the sequence. To this end, three test suites have been developed: the NIST Statistical Test Suite, the Dieharder suite, and the ENT suite. Each test suite is designed to look for patterns in input sequences by way of a variety of tests.

According to John Soto, an official at NIST, they have 3 viewpoints when evaluating sequences: threshold values, fixed ranges, and probability values [13]. For example, in monobits, a line of zeros and ones, they employ a frequency test, by counting and comparing the amount of ones and zeros to determine whether they have an equal amount [14].

Dieharder is a test constantly in development by Robert G. Brown. Similarly to NIST, it employs multiple tests in order to evaluate randomness. Dieharder tests use most of the processes that NIST uses [15]. The main difference being

the development of new tests by Robert G. Brown himself. It can simply be described as an upgrade from its predecessor, Diehard.

ENT is a program that tests sequences in bits in order to test randomness. ENT tests Entropy, uses the Chi-Square Test, finds the Arithmetic Mean, finds the Monte Carlo Value for Pi and the Serial Correlation Coefficient [16]. Both ENT and Dieharder are respected tests for measuring validity of a random number generator. Other research from the Data Science Lab includes [17]–[59].

B. Preliminaries

The Co-RNG approach due to Antoniadis et al. [6] splits the algorithm into two methods: Co-OBS and Co-REC. This method can be improved further. One improvement is to increase the number of threads. Instead of two observation threads in the Co-RNG [6], there can be a N number of threads. Having additional threads increases the speed of the random generator thereby reducing execution time. To keep track of the N observation threads, an atomic snapshot [61] must be used. Using an atomic snapshot records multiple shared memory locations in a single hardware step to prevent inaccurate recordings in an asynchronous system [12]. These enhancements to the Co-RNG algorithm will increase the speed and reduce the time to create the random sequence and improve the quality of random number generation. By implementing with multithreaded programming, the chances of concurrent memory interleavings are increased during the observation algorithm.

The Co-RNG [6] algorithm is used as a basis for the random number generator. The algorithm relies on measuring memory accesses at the memory controller as the random source to generate sets of random numbers. The random number generator constructed from the modified Co-RNG algorithm will measure concurrent memory accesses with an arbitrary number of threads. The previous Co-RNG algorithm consists of two algorithms, Co-OBS and Co-REC, which allow only two observation threads [6]. Having only two threads limits the utilization of modern multicore machines with many cores.

Linearizability is the property of a multi-threaded algorithm which allows for the construction of a unique serialized history for an asynchronous execution which respects the real time order [12]. The existing Co-RNG approach is not linearizable, despite both the observation threads performing atomic operations. Co-OBS is used to observe the sequence using two threads and records the shared memory asynchronously [6]. Co-REC [6], on the other hand, is used to reconstruct the sequence written by the two observation threads which are then sent to the Blum-Elias algorithm [7]. The Co-REC algorithm suffers from being complicated since the atomic reads of shared memory allow for indistinguishable observations. The use of compare-and-swap in our proposed algorithm simplifies the reconstruction algorithm because all observations are distinguishable, and the total order of operations can be maintained.

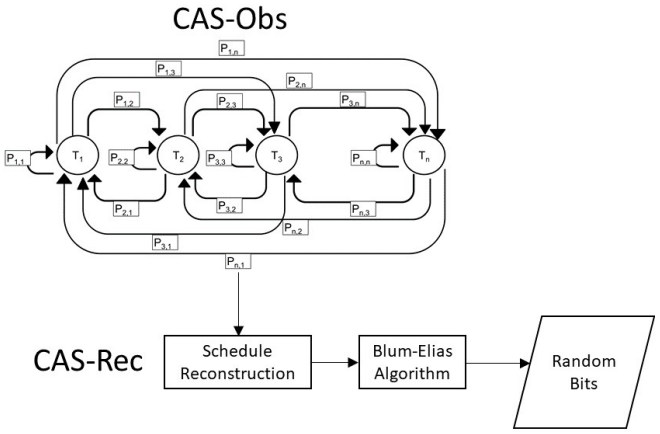

Fig. 1. The flow diagram of our approach.

To test the effectiveness of our Random Number Generator, two programs, ENT and Dieharder, are used. Each of these test suites include similar algorithms to test the authenticity of the generator. According to the National Institute of Standards and Technology, "Because there are so many tests for judging whether a sequence is random or not, no specific finite set of tests is deemed 'complete'" [62]. There is no single program set as the standard to test the probability of a random generator. Multiple tests will be used to verify the robustness of our algorithm's generated sequences.

III. CAS-RNG

To generate random numbers, memory accesses are measured indirectly at the memory controller. We measure the memory access times indirectly in software. These memory accesses can be converted into a sequence of unbiased random bits using our new algorithm. A previous solution using this technique was proposed with limited parallelism [6]. We take a new approach to measuring memory accesses using the compare-and-swap operation. We propose a new, generalized reconstruction algorithm to support n threads instead of two. In order to keep track of all the observations from each thread, an atomic snapshot of the shared memory will be taken. We break down our approach into three stages: observation, reconstruction, and extraction. The flow diagram of the algorithm is given in Figure 1.

A. Observation

There are two ways an atomic snapshot can be used in the Co-Obs algorithm: clean double collect [12] or the compare-and-swap method. The first method, clean double collect [12], is not time efficient. The algorithm is slower than the compare-and-swap method because at least $2n$ atomic reads are necessary for a single snapshot. The benefit of this method is it only uses atomic reads and writes. However, if other threads take steps during a thread's collection, then the thread must retry. These delays will cause the resulting reconstruction to be difficult as well as increase the time to execute the observation stage.

In order to avoid the drawbacks of the atomic snapshot using a clean double collect, we use a compare-and-swap operation. This method avoids the delays caused by the clean double collect, and also guarantees a total order on the operations. We use linearizability as the correctness condition for our algorithm. The CAS-OBS algorithm is given in Algorithm 1.

Algorithm 1 CAS-OBS

Input: SM, *the array of shared memory*, R, *the number of rounds*, ID, *the thread ID*
Output: Obs, *the array of observations recorded*

```
1:  for i ← 0 to R do
2:      success ← false
3:      while !success do
4:          old ← SM.get()
5:          x ← old
6:          x[ID].set(i)
7:          cacheFlush()
8:          success ← SM.compareAndSet(old, x)
9:          cacheFlush()
10:     Obs[i] ← x
11: return Obs
```

The CAS-OBS algorithm is run concurrently by n threads. Each thread completes R compare-and-swap operations. The atomic nature of the compare-and-swap operation onto the shared memory SM guarantees a total order. We can completely distinguish the sequence of operations by all threads. The shared memory location SM contains a reference to an array of length n and each thread is given a unique $ID \in 1, 2, \ldots, n$. Each thread only writes to the array location indexed by its unique ID. To complete a round, a thread must capture a local copy, old, of the shared memory in line 4, then make its change to old by writing its current round number to the ID^{th} location of the new array x in line 6. Finally, the thread must successfully complete a compare-and-swap operation on the shared memory location, SM, to publish the change to the other threads atomically in line 8. If the compare-and-swap fails, the thread must retry. However, we maintain progress in the observation stage by noting a thread can only fail the compare-and-swap operation if another thread succeeds. Once the thread succeeds, it can save the published array, x, as its observation for that round in line 10.

Each thread must make a new local copy of the array since; if not, multiple threads could change the same reference and the SM reference would not change. This condition would have the undesirable effect of threads being able to overwrite other thread's changes, since the compare-and-swap operation would succeed. Once a thread publishes the reference to the array in SM, the array is never written to, only read from. Furthermore, if a thread succeeds in publishing its change, then the array, x, that the thread publishes becomes a valid snapshot of the memory, linearized at the moment the compare-and-swap operation succeeds. Therefore, it is possible to reconstruct the total order of compare and swap operations onto the shared memory location, SM. Note, that if the old collection acquired in line 5 is used as that round's observation then it allows for indistinguishable outcomes where multiple threads observe the same state configuration.

A cache flush is required to implement the CAS-OBS algorithm for the random number generator to work properly. If the CAS-OBS algorithm runs without a cache flush, there is a high chance of data being retrieved from the cache instead of the RAM. Reads pulling memory from the cache is a case that should be avoided; cache is not an appropriate source of randomness [5]. Relying on the cache as a source of randomness can lead to a faulty input trace, where the sequence of reads and writes is not reliable as a random source. In lines 7 and 9 of the CAS-OBS algorithm, a cache flush is carried out after each read and write.

Following the observations, an $N \times R$ matrix, where each row is a thread's Obs array from Algorithm 1, completely describes the total order of NR observations. To construct the sequence, the CAS-REC algorithm is used which takes the $N \times R$ matrix as input.

B. Reconstruction

We avoid the problem of indistinguishable observations which complicate the reconstruction algorithm. The reconstruction algorithm proposed in Algorithm 2 is designed to accept the observations from Algorithm 1. Allowing for indistinguishable observations only serves to complicate the algorithm, and the use of compare-and-swap can maintain that at every step in the reconstruction algorithm, there exists only one thread whose write occurred. That is, it is always possible to tell which thread's write preceded the other threads'. This maintains a total order on the history and allows for a straight forward reconstruction.

Once the observations are collected, they are stored in N arrays of length R, where N is the number of observation threads and R is the number of rounds. Each thread has an observation for each round which consists of a snapshot of the shared memory. Then, the schedule of memory accesses, or trace, is reconstructed using Algorithm 2. The output of this algorithm is the reconstructed trace, t, which is an array of length $N * R$. Each entry in the array is the ID of the thread whose successful compare-and-swap operation was linearized at that moment in time relative to all other threads. In the worst case, the algorithm runs in $O(R * N^3)$ time, where the for loop in lines 3 to 9 run for every thread in each observation, causing N^2 operations and this process is repeated $N * R$ times in the loop in line 2.

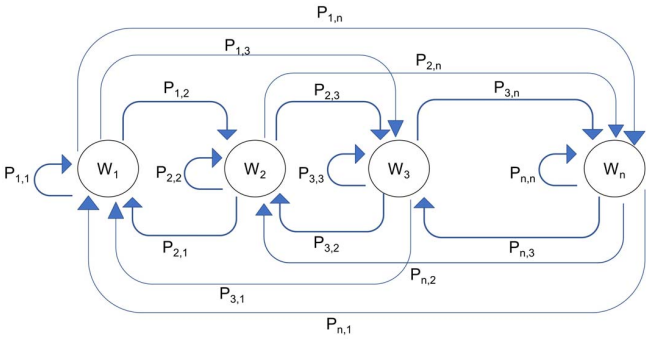

Fig. 2. N-State Markov Chain

Algorithm 2 CAS-REC

Input: *Obs, the observations*, R, *the number of rounds*, N, *the number of observers*
Output: t, *the reconstructed trace*

```
    count ← N
    while count > 1 do
3:    for i ← 0 to N do
        if n[i] < R then
          all ← true
6:        for j = 0 to N do
            if j ≠ i then
              if Obs[i][n[i]].get(j)≥ n[j] then
9:              all ← false
          if all = true then
            t[idx] = i
12:         idx + +
            n[i] + +
            break
15:   count ← 0
      for i ← 0 to N do
        if n[i] < R then
18:       count + +
    for i ← 0 to N do
      while n[i] < R do
21:     t[idx] ← i
        idx + +
        n[i] + +
24:
    return t
```

We model the shared memory accesses as a first order Markov chain [63] with n states, where n is the number of observation threads. It was shown in the work of Antoniadis, et al., that it is appropriate to model memory accesses as a first order Markov chain [6]. Each state represents a successful compare-and-swap operation to the memory. This process is simpler than previous approaches where the number of states scale exponentially due to only using atomic reads and writes. We attribute this improvement to the use of compare-and-swap. This refinement creates a simple and scalable Markov chain with N^{N*R} possible sequences of length $N * R$. Figure 2 shows the arbitrary state Markov chain and each edge of nonzero transition probability. The CAS-OBS procedure can be viewed as a random walk on the Markov chain in Figure 2 and a state k represents a successful compare-and-set operation in line 8 of Algorithm 1 by thread k in some round i. This is recorded as the i^{th} observation in line 10 of Algorithm 1. We use the path reconstructed by Algorithm 2 as input to the Blum-Elias algorithm as proposed by Zhou and Bruck [7] and implemented by Andoniadis, et al. [6].

IV. Experiment

To test the results of the experiment, two test suites: ENT and Dieharder, will be utilized. As stated in the Related Work, there is no single method for effectively testing randomness. Furthermore, there is no known way to prove that a random number generator is random or cryptographically secure. Rather, the best measurements of randomness are determined by a variety of tests on large samples generated. Each data set in the experiment contains several million random bits generated by CAS-RNG. Flaws in one test may be covered in another one. By testing in two different programs, the validity of results is assured.

NIST Statistical Test Suite (STS) has three principles concerning the testing of random number generators. These principles are probability values, threshold values, fixed ranges. The variable labeled as s stands for the binary sequence, which is present in every principle and most prominent in probability values. Probability values depend on the p-value statistic, described by NIST as "...the probability of obtaining a test statistic as large or larger than the one observed if the sequence is random" [13]. The closer the p-value is to 1, the more random the sequence. The p-value compares randomness of a true random number generator to the randomness of s. [64]. The next two principles are best understood in terms of monobits. Fixed ranges simply mean the ratio in which the data exists in a string of monobits. A truly random sequence will have a ratio of 1:1, with ones and zeroes occurring in equal amounts. The distribution of monobits cannot be skewed; otherwise, the data will be considered biased, and consequently, not random. Threshold values deal in terms of pattern recognition. If there are identifiable patterns, the data will not be considered random. These three principles confirm a single test for randomness would be incomplete.

The Dieharder test suite contains many of the same tests from the NIST STS. The key differences are the tests that Robert G. Brown develops himself as well as the large battery of tests compared to other test suites [15]. The RGB Bit Distribution test is unique to Dieharder. This test chunks the sequence and determines the Chi-square and p-value for the chunks, which are not overlapping. We test the CAS-RNG using the Dieharder and the ENT test suites on different data set sizes. The random number generator passes many of the Dieharder tests, depending on the sample. The output of the full analysis from a single trial is given in the Appendix. A plot of the p-values for each test is given in Figure 3.

TABLE I
RESULTS FROM ENT TESTING

Description	Test 1 Results	Test 2 Results	Test 3 Results
Sample Size(MB)	2.3	4.3	34.2
Entropy(bites/Byte)	7.99670	7.999267	7.999373
Chi Square Distribution	1098.95	4678.60	32220.20
Arithmetic mean value of data bytes	127.8565	128.1637	128.1489
Monte Carlo value of Pi	3.124803022(0.53%)	3.118101880(0.75%)	3.116531248(0.80%)
Serial correlation coefficient	0.000948	0.001586	0.001468

TABLE II
TIMING DATA

Bits	Real Time (s)	Throughput (bit/s)
18104080	60.502	299,231.10
17967832	64.358	279,185.68
17562904	71.966	244,044.47
17287136	64.121	269,601.78
16000312	67.106	238,433.40
18541432	71.029	261,040.31
18060368	71.085	254,067.22
17443920	69.214	252,028.78
16426072	67.714	242,580.15
18398040	64.538	285,072.98

Average Throughput: 262,528.59 bit/s

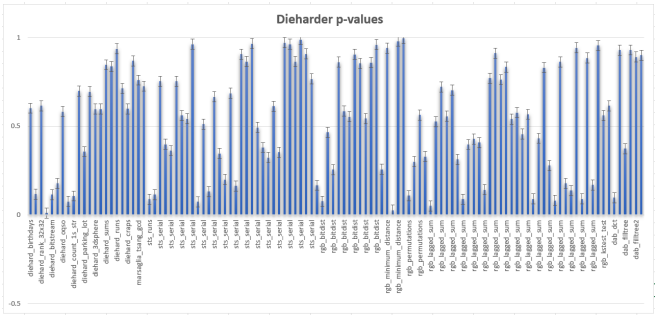

Fig. 3. The p-values from the Dieharder test.

ENT is different from the Dieharder test suite in that it only employs 5 tests in order to validate randomness: Entropy, Chi-Square, Arithmetic Mean, Monte Carlo Value for Pi and Serial Correlation Coefficient. The Chi-Square Test is frequently used as a way to test random data. The results given from three trials of the ENT test suite on different random data sets are given in Table I. The bit entropy is at least 7.99 in every trial.

Lastly, we give the throughput calculated from our implementation of the random number generator. We implemented the CAS-OBS and CAS-REC in Java and created a C library to implement the cache flush, since Java lacks instructions to flush the cache directly. Unfortunately, this removes the portability of our implementation which Java normally provides. Instead, an architecture dependent cache flush should be provided for porting to different platforms. After the trace is generated, we use the existing implementation of the Blum-Elias algorithm given by Antoniadis [6]. We implemented a modified driver program for the Blum-Elias algorithm to interface with our Java implementation. Finally, we create a driver program in C to provide a command line interface to easily generate test data.

We ran our tests on a Virtual Machine running 64-bit Ubuntu 18.04 LTS on a Windows 10.0.17134 host operating system using VirtualBox 6.0.4r128413 (Qt5.6.2). The virtual machine was allocated 2 hardware cores (4 logical), 8,196 MB of RAM, and 128MB of video memory with VT-x/AMD-V, nested paging, and KVM paravirtualization. The hardware on the host machine has a Intel Xeon E5-1630 v4 CPU clocked at 3.70 GHz, 64 GB of RAM, and an Nvidia GeForce 1080 graphics card. We achieve an average throughput of 262,528.59 bits per second with our implementation shown in Table II. The throughput could easily be improved by running the algorithm on faster hardware, allocating more physical cores to the implementation, writing the implementation in C, and using process-based parallelism like OpenMP to scale the algorithms to run on multiple nodes. A comparison with other approaches is given in Table III. It is difficult to draw a fair comparison between methods as there are differences in hardware and implementation and each method has pros and cons besides its throughput.

TABLE III
COMPARISON WITH EXISTING METHODS

Test	Throughput (Mbits/sec)
CPU Jitter [4]	0.008
FPGA-based TRNG [65]	6.050
EEPROM RNG [66]	166
STRNG [67]	200
CAS-RNG (This Work)	**262.529**

V. CONCLUSION

In this paper we propose a new random number generator which measures memory accesses at the memory controller indirectly through software. We generalize on a previous approach and support an arbitrary number of threads. The advantages of this method are unlimited parallelism which translates to faster trace generation and increased chances for memory interleavings. We also solve the problem of indistinguishable observations faced in a previous approach by using compare and swap operations which simplifies the schedule reconstruction. We test our random number generator with two test suites, Dieharder and ENT, on three data sets and achieve satisfactory results. We achieve an average throughput of 262,528 bits per second. The throughput could easily be

improved by running the algorithm on faster hardware, writing the implementation in C, and using process-based parallelism like OpenMP to scale the algorithm to run on multiple nodes. Benefits of this approach over Co-RNG are a simplified, linearizable protocol that can easily be scaled to take advantage of multicore architectures, which increases the robustness of the created traces.

Some ideas for future work include implementing the proposed algorithms as a kernel module, testing the random number generator in cryptographic protocols, and designing a linear time algorithm for N thread schedule reconstruction.

ACKNOWLEDGMENT

This research was supported in part by the Department of Defense under the Army Educational Outreach Program (AEOP) and the National Science Foundation under the Research Experiences for Teachers (RET) program.

APPENDIX

Table IV contains the results of a trial using the Dieharder test suite.

TABLE IV
RESULTS FROM DIE HARDER TESTING

Test Name	nt	t sam.	p sam.	p-value	Assessment
dh birthdays	0	100	100	0.60211193	PASSED
dh operm5	0	1000000	100	0.11761350	PASSED
dh rank 32x32	0	40000	100	0.61627575	PASSED
dh rank 6x8	0	100000	100	0.01053413	PASSED
dh bitstream	0	2097152	100	0.11493929	PASSED
dh opso	0	2097152	100	0.17779064	PASSED
dh oqso	0	2097152	100	0.58282417	PASSED
dh dna	0	2097152	100	0.07571953	PASSED
dh count 1s str	0	256000	100	0.10675637	PASSED
dh count 1s byt	0	256000	100	0.69887467	PASSED
dh parking lot	0	12000	100	0.35741380	PASSED
dh 2dsphere	2	8000	100	0.69467665	PASSED
dh 3dsphere	3	4000	100	0.59753284	PASSED
dh squeeze	0	100000	100	0.59698262	PASSED
dh sums	0	100	100	0.84714513	PASSED
dh runs	0	100000	100	0.83717795	PASSED
dh runs	0	100000	100	0.93718614	PASSED
dh craps	0	200000	100	0.71422711	PASSED
dh craps	0	200000	100	0.59813653	PASSED
tsang gcd	0	10000000	100	0.86871313	PASSED
tsang gcd	0	10000000	100	0.76056949	PASSED
sts monobit	1	100000	100	0.72613955	PASSED
sts runs	2	100000	100	0.08819269	PASSED
sts serial	1	100000	100	0.11455184	PASSED
sts serial	2	100000	100	0.75445713	PASSED
sts serial	3	100000	100	0.39886555	PASSED
sts serial	3	100000	100	0.36319430	PASSED
sts serial	4	100000	100	0.75433968	PASSED
sts serial	4	100000	100	0.56159855	PASSED
sts serial	5	100000	100	0.54198244	PASSED
sts serial	5	100000	100	0.96334311	PASSED
sts serial	6	100000	100	0.07487319	PASSED
sts serial	6	100000	100	0.51223479	PASSED
sts serial	7	100000	100	0.13144794	PASSED
sts serial	7	100000	100	0.66598230	PASSED
sts serial	8	100000	100	0.34618789	PASSED
sts serial	8	100000	100	0.19970824	PASSED
sts serial	9	100000	100	0.68683720	PASSED
sts serial	9	100000	100	0.16390930	PASSED
sts serial	10	100000	100	0.90677933	PASSED
sts serial	10	100000	100	0.86450898	PASSED
sts serial	11	100000	100	0.96559554	PASSED
sts serial	11	100000	100	0.49351225	PASSED
sts serial	12	100000	100	0.38054616	PASSED
sts serial	12	100000	100	0.32306979	PASSED
sts serial	13	100000	100	0.61301409	PASSED
sts serial	13	100000	100	0.35404988	PASSED
sts serial	14	100000	100	0.97201966	PASSED
sts serial	14	100000	100	0.96328185	PASSED
sts serial	15	100000	100	0.86571502	PASSED
sts serial	15	100000	100	0.98836476	PASSED
sts serial	16	100000	100	0.90823582	PASSED
sts serial	16	100000	100	0.76676980	PASSED
rgb bitdist	1	100000	100	0.16720244	PASSED
rgb bitdist	2	100000	100	0.07757307	PASSED
rgb bitdist	3	100000	100	0.46617512	PASSED
rgb bitdist	4	100000	100	0.25515346	PASSED
rgb bitdist	5	100000	100	0.86238900	PASSED
rgb bitdist	6	100000	100	0.58571105	PASSED
rgb bitdist	7	100000	100	0.55347182	PASSED
rgb bitdist	8	100000	100	0.90526925	PASSED
rgb bitdist	9	100000	100	0.85459888	PASSED
rgb bitdist	10	100000	100	0.54343586	PASSED
rgb bitdist	11	100000	100	0.85955345	PASSED
rgb bitdist	12	100000	100	0.96073582	PASSED
rgb min dist	2	10000	1000	0.25687550	PASSED
rgb min dist	3	10000	1000	0.94105725	PASSED
rgb min dist	4	10000	1000	0.02655776	PASSED
rgb min dist	5	10000	1000	0.97958993	PASSED
rgb perm.	2	100000	100	0.99548110	WEAK
rgb perm.	3	100000	100	0.10918976	PASSED
rgb perm.	4	100000	100	0.30067013	PASSED
rgb perm.	5	100000	100	0.56355072	PASSED
rgb lagged sum	0	1000000	100	0.32977275	PASSED
rgb lagged sum	1	1000000	100	0.05176343	PASSED
rgb lagged sum	2	1000000	100	0.52586403	PASSED
rgb lagged sum	3	1000000	100	0.72181352	PASSED
rgb lagged sum	4	1000000	100	0.55689168	PASSED
rgb lagged sum	5	1000000	100	0.70416891	PASSED
rgb lagged sum	6	1000000	100	0.31310402	PASSED
rgb lagged sum	7	1000000	100	0.08921029	PASSED
rgb lagged sum	8	1000000	100	0.39772159	PASSED
rgb lagged sum	9	1000000	100	0.42865011	PASSED
rgb lagged sum	10	1000000	100	0.40793002	PASSED
rgb lagged sum	11	1000000	100	0.14196060	PASSED
rgb lagged sum	12	1000000	100	0.77073119	PASSED
rgb lagged sum	13	1000000	100	0.91291123	PASSED
rgb lagged sum	14	1000000	100	0.76361644	PASSED
rgb lagged sum	15	1000000	100	0.83418338	PASSED
rgb lagged sum	16	1000000	100	0.54128493	PASSED
rgb lagged sum	17	1000000	100	0.57651017	PASSED
rgb lagged sum	18	1000000	100	0.45630399	PASSED
rgb lagged sum	19	1000000	100	0.56757579	PASSED
rgb lagged sum	20	1000000	100	0.08996408	PASSED
rgb lagged sum	21	1000000	100	0.43218363	PASSED
rgb lagged sum	22	1000000	100	0.83088358	PASSED
rgb lagged sum	23	1000000	100	0.27826794	PASSED
rgb lagged sum	24	1000000	100	0.08133363	PASSED
rgb lagged sum	25	1000000	100	0.86286318	PASSED
rgb lagged sum	26	1000000	100	0.17775301	PASSED
rgb lagged sum	27	1000000	100	0.13708069	PASSED
rgb lagged sum	28	1000000	100	0.94299568	PASSED
rgb lagged sum	29	1000000	100	0.09042752	PASSED
rgb lagged sum	30	1000000	100	0.88517483	PASSED
rgb lagged sum	31	1000000	100	0.16946859	PASSED
rgb lagged sum	32	1000000	100	0.95616908	PASSED
rgb kstest test	0	10000	1000	0.56155741	PASSED
dab bytedistrib	0	51200000	1	0.61587529	PASSED
dab dct	256	50000	1	0.09733919	PASSED
dab filltree	32	15000000	1	0.93009017	PASSED
dab filltree	32	15000000	1	0.37324042	PASSED
dab filltree2	0	5000000	1	0.93042991	PASSED
dab filltree2	1	5000000	1	0.89077655	PASSED
dab monobit2	12	65000000	1	0.90143897	PASSED

REFERENCES

[1] L. C. Noll, R. G. Mende, and S. Sisodiya, "Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system," mar " 24" 1998, uS Patent 5,732,138.

[2] M. Haahr, "Random. org: True random number service," *School of Computer Science and Statistics, Trinity College, Dublin, Ireland. Website (http://www. random. org). Accessed*, vol. 10, 2010.

[3] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the linux random number generator," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 15–pp.

[4] S. Múller, "Cpu time jitter based non-physical true random number generator," in *Ottawa Linux Symposium*, 2014.

[5] S. Agafin and A. Krasnopevtsev, "Memory access time as entropy source for rng," in *Proceedings of the 7th International Conference on Security of Information and Networks*. ACM, 2014, p. 176.

[6] P. F. Blanchard, R. Guerraoui, J. M. Stainer, and K. Antoniadis, "Concurrency as a random number generator-technical report," École Polytechnique Fédérale de Lausanne, Tech. Rep., 2016.

[7] H. Zhou and J. Bruck, "Generalizing the blum-elias method for generating random bits from markov chains," in *2010 IEEE International Symposium on Information Theory*. IEEE, 2010, pp. 1248–1252.

[8] L. Lamport, "Concurrent reading and writing," *Communications of the ACM*, vol. 20, no. 11, pp. 806–811, 1977.

[9] P. B. Hansen, "Concurrent programming concepts," *ACM Computing Surveys (CSUR)*, vol. 5, no. 4, pp. 223–245, 1973.

[10] G. L. Peterson, "Concurrent reading while writing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 1, pp. 46–55, 1983.

[11] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[12] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[13] J. Soto, "Statistical testing of random number generators," in *Proceedings of the 22nd National Information Systems Security Conference*, vol. 10, no. 99. NIST Gaithersburg, MD, 1999, p. 12.

[14] L. Obrátil, "The automated testing of randomness with multiple statistical batteries," Ph.D. dissertation, Masarykova univerzita, Fakulta informatiky, 2017.

[15] R. G. Brown, D. Eddelbuettel, and D. Bauer, "Dieharder: A random number test suite," *Open Source software library, under development, URL http://www.phy.duke.edu0/řgb/General/dieharder.php*, 2013.

[16] J. Walker, "Ent: a pseudorandom number sequence test program," *Software and documentation available at/www. fourmilab. ch/random/S*, 2008.

[17] N. R. R. M. M. S. M. B. J. Z. L. G. P. O. Felix Zhan, Anthony Martinez, "Beyond cumulative sum charting in non-stationarity detection and estimation," *IEEE Access*, 2019.

[18] Z. J. Schwob, M. and D. A., "Modeling cell communication with time-dependent signaling hypergraphs," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, p. doi: 10.1109/TCBB.2019.2937033, 2019.

[19] C. Chiu and J. Zhan, "Deep learning for link prediction in dynamic networks using weak estimators," *IEEE Access*, vol. 6, no. 1, pp. 35 937 – 35 945, 2018.

[20] M. Bhaduri and J. Zhan, "Using empirical recurrences rates ratio for time series data similarity," *IEEE Access*, vol. 6, no. 1, pp. 30 855–30 864, 2018.

[21] J. Wu, J. Zhan, and S. Chobe, "Mining association rules for low frequency itemsets," *PLOS ONE*, vol. 13, no. 7, 2018.

[22] P. Ezatpoor, J. Zhan, J. Wu, and C. Chiu, "Finding top-k dominance on incomplete big data using mapreduce framework," *IEEE Access*, vol. 6, no. 1, pp. 7872–7887, 2018.

[23] P. Chopade and J. Zhan, "Towards a framework for community detection in large networks using game-theoretic modeling," *IEEE Transactions on Big Data*, vol. 3, no. 3, pp. 276–288, 2017.

[24] M. Bhaduri, J. Zhan, and C. Chiu, "A weak estimator for dynamic systems," *IEEE Transactions on Big Data*, vol. 5, no. 1, pp. 27 354–27 365, 2017.

[25] M. Pirouz and J. Zhan, "Toward efficient hub-less real time personalized pagerank," *IEEE Transactions on Big Data*, vol. 5, no. 1, pp. 26 364–26 375, 2017.

[26] M. Bhaduri, J. Zhan, C. Chiu, and F. Zhan, "A novel online and non-parametric approach for drift detection in big data," *IEEE Access*, vol. 5, no. 1, pp. 15 883–15 892, 2017.

[27] C. Chiu, J. Zhan, and F. Zhan, "Uncovering suspicious activity from partially paired and incomplete multimodal data," *IEEE Access*, vol. 5, no. 1, pp. 13 689 – 13 698, 2017.

[28] R. Ahn and J. Zhan, "Using proxies for node immunization identification on large graphs," *IEEE Access*, vol. 5, no. 1, pp. 13 046–13 053, 2017.

[29] M. Wu, J. Zhan, and J. Lin, "Ant colony system sanitization approach to hiding sensitive itemsets," *IEEE Access*, vol. 5, no. 1, pp. 10 024–10 039, 2017.

[30] J. Zhan and B. Dahal, "Using deep learning for short text understanding," *Journal of Big Data*, vol. 4, no. 34, pp. 1–15, 2017.

[31] J. Zhan, S. Gurung, and S. P. K. Parsa, "Identification of top-k nodes in large networks using katz centrality," *Journal of Big Data*, vol. 4, no. 16, 2017.

[32] J. Zhan, T. Rafalski, G. Stashkevich, and E. Verenich, "Vaccination allocation in large dynamic networks," *Journal of Big Data*, vol. 4, no. 2, pp. 161–172, 2017.

[33] J. M.-T. Wu, J. Zhan, and J. C.-W. Lin, "An aco-based approach to mine high-utility itemsets," *Knowledge-Based Systems*, vol. 116, pp. 102–113, 2017.

[34] M. Pirouz, J. Zhan, and S. Tayeb, "An optimized approach for community detection and ranking," *Journal of Big Data*, vol. 3, no. 22, pp. 102–113, 2017.

[35] J. Zhan, V. Gudibande, and S. P. K. Parsa, "Idenfication of top-k influential communities in large networks," *Journal of Big Data*, vol. 3, no. 16, 2016.

[36] M. Pirouz and J. Zhan, "Node reduction in personalized page rank estimation for large graphs," *Journal of Big Data*, vol. 3, no. 12, 2016.

[37] H. Selim and J. Zhan, "Towards shortest path identification on large networks," *Journal of Big Data*, vol. 3, no. 10, 2016.

[38] X. Fang and J. Zhan, "Sentiment analysis using product review data," *Journal of Big Data*, vol. 2, no. 5, pp. 1–14, 2015.

[39] P. Chopade and J. Zhan, "Structural and functional analytics for community detection in large-scale complex networks," *Journal of Big Data*, vol. 2, no. 1, pp. 1–28, 2015.

[40] J. Zhan and X. Fang, "A computational framework for detecting malicious actors in communities," *International Journal of Privacy, Security, and Integrity*, vol. 2, no. 1, pp. 1–20, 2014.

[41] A. Rajasekar, H. Kum, M. Cross, J. Crabtree, S. Sankaran, H. Lander, T. Carsey, G. King, and J. Zhan, "The databridge," *Science Journal*, vol. 2, no. 1, pp. 1–14, 2013.

[42] J. Zhan, X. Fang, and N. Koceja, "A novel framework on data reduction," *Science Journal*, vol. 2, no. 1, pp. 15–23, 2013.

[43] A. Doyal and J. Zhan, "Towards ddos defense and traceback," *International Journal of Privacy, Security, and Integrity*, vol. 1, no. 4, pp. 299–311, 2013.

[44] J. Zhan and X. Fang, "Towards social network evolution," *Human Journal*, vol. 1, no. 1, pp. 218–233, 2012.

[45] J. Zhan, J. Oommen, and J. Crisostomo, "Anomaly detection in dynamic systems using weak estimator," *ACM Transaction on Internet Technology*, vol. 11, no. 1, pp. 53–69, 2011.

[46] J. Zhan and X. Fang, "Social computing: The state of the art," *International Journal of Social Computing and Cyber-Physical Systems*, vol. 1, no. 1, pp. 1–12, 2011.

[47] N. Mead, M. S., and J. Zhan, "Integrating privacy requirements considerations into a security requirements engineering method and tool," *International Journal of Information Privacy, Security and Integrity*, vol. 1, no. 1, pp. 106–126, 2011.

[48] J. Zhan, "Granular computing in privacy-preserving data mining," *International Journal of Granular Computing, Rough Sets and Intelligent Systems*, vol. 1, no. 3, pp. 272–288, 2010.

[49] J. Wang, J. Zhang, and J. Zhan, "Towards real-time performance of data privacy protection," *International Journal of Granular Computing, Rough Sets and Intelligent Systems*, vol. 1, no. 4, pp. 329–342, 2010.

[50] J. Zhan, "Secure collaborative social networks," *IEEE Transaction on Systems, Man, and Cybernetics, Part C*, vol. 40, no. 6, pp. 682–689, 2010.

[51] J. Zhan, H. C., I. Wang, T. Hsu, C. Liau, and W. D., "Privacy-preserving collaborative recommender systems," *IEEE Transaction on Systems, Man, and Cybernetics, Part C*, vol. 40, no. 4, pp. 472–476, 2010.

[52] H. Park, J. Hong, J. Park, J. Zhan, and D. Lee, "Attribute-based access control using combined authentication technologies," *IEEE Transaction on Mobile Computing*, vol. 9, no. 6, pp. 824–837, 2010.

[53] I. Wang, C. Shen, J. Zhan, T. Hsu, C. Liau, and D. Wang, "Empirical evaluations of secure scalar product," *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, vol. 39, no. 4, pp. 440–447, 2009.

[54] A. Inoue, T. Wong, and J. Zhan, "Applications of machine learning to information security and privacy," *Journal of Japanese Society for Fuzzy Theory and Intelligent Informatics*, vol. 19, no. 3, pp. 222–232, 2009.

[55] J. Zhan, "Privacy-preserving collaborative data mining," *IEEE Computational Intelligence Magazine*, vol. 3, no. 2, pp. 31–41, 2008.

[56] A. Bashir and J. Zhan, "Not always a blunt tool – legislation in the context of privacy externalities," *Communications of the Chinese Cryptology and Information Security Association*, vol. 2, no. 1, pp. 36–48, 2008.

[57] J. Zhan and V. Rajamani, "The economic aspects of privacy," *International Journal of Security and Its Applications*, vol. 2, no. 3, pp. 101–108, 2008.

[58] J. Zhan, "The economic aspects of privacy," *International Journal of Security and Its Applications*, vol. 2, no. 3, pp. 101–108, 2008.

[59] W. Zhang, P. Wang, K. Peace, J. Zhan, and Y. Zhang, "On truth, uncertainty, and bipolar logic," *Journal of New Mathematics and Natural Computing*, vol. 4, no. 2, pp. 55–65, 2008.

[60] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *Journal of the ACM (JACM)*, vol. 40, no. 4, pp. 873–890, 1993.

[61] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," Booz-Allen and Hamilton Inc McLean VA, Tech. Rep., 2001.

[62] C. M. Grinstead and J. L. Snell, *Introduction to probability*. American Mathematical Soc., 2012.

[63] K. Marton and A. Suciu, "On the interpretation of results from the nist statistical test suite," *Science and Technology*, vol. 18, no. 1, pp. 18–32, 2015.

[64] S. H. Kwok and E. Y. Lam, "Fpga-based high-speed true random number generator for cryptographic applications," in *TENCON 2006-2006 IEEE Region 10 Conference*. IEEE, 2006, pp. 1–4.

[65] J. Genoff, "An extremely massive high-quality true-random binary data stream generator," in *2018 IEEE XXVII International Scientific Conference Electronics-ET*. IEEE, 2018, pp. 1–4.

[66] A. Cherkaoui, V. Fischer, L. Fesquet, and A. Aubert, "A very high speed true random number generator with entropy assessment," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 179–196.