

StAccato: Reducing the Performance Penalty of Randomness

Ranganath Selagamsetty
Computer Science
University of Wisconsin - Madison
Madison, Wisconsin, USA
selagamsetty@wisc.edu

Joshua San Miguel
Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, Wisconsin, USA
jsanmiguel@wisc.edu

Mikko Lipasti
Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, Wisconsin, USA
mikko@engr.wisc.edu

Abstract

Pseudo-random numbers are integral to Monte Carlo methods, machine learning optimizations, and physics simulations, among other applications. However, reliance on a stream of stochastic values introduces significant performance overheads. Prior works identify one of these overheads as misprediction penalties when probabilistic values are used to determine the outcome of a branch. This work broadens the scope to address additional performance costs associated with using a stochastic value (SV) beyond branch mispredictions, including those from generation strategies, arithmetic transforms, and memory operations dependent on SVs.

We introduce StAccato, a stochastic accelerator that addresses randomness from generation to consumption. StAccato mitigates performance loss by leveraging a fast, high-quality hardware random number generator design, reducing latency and computational costs. We also present a compiler design to translate code to fully utilize StAccato, optionally leveraging a helper thread for further speedup. This achieves an average of 1.49x speedup—97.9% of the attainable ideal. Even without helper thread support, StAccato delivers a 1.34x speedup over the selected benchmarks. By combining hardware acceleration with compiler automation, StAccato optimizes stochastic workloads across diverse applications.

CCS Concepts

• **Computer systems organization** → **Special purpose systems; Architectures; Superscalar architectures**; • **Hardware** → **Arithmetic and datapath circuits**.

Keywords

Tightly coupled accelerators, stochastic workloads, pseudo-random number generators

ACM Reference Format:

Ranganath Selagamsetty, Joshua San Miguel, and Mikko Lipasti. 2026. StAccato: Reducing the Performance Penalty of Randomness. In *Proceedings of the 23rd ACM International Conference on Computing Frontiers (CF '26)*, May 19–21, 2026, Catania, Italy. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3801487.3801821>

1 Introduction

Stochastic programs play a central role across diverse domains, including cryptography [5, 22, 31, 34, 46, 65], machine learning

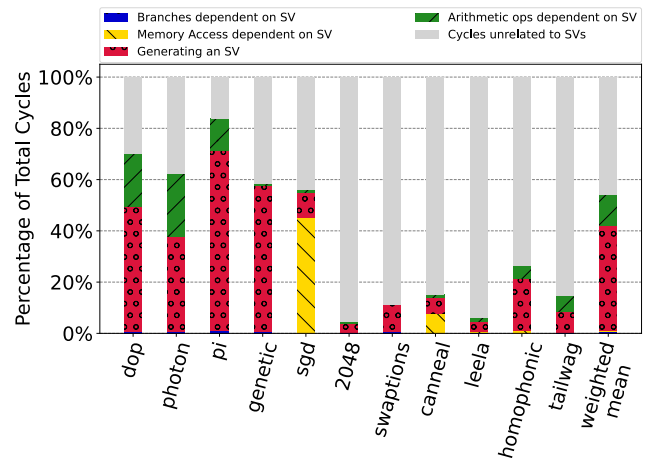


Figure 1: Breakdown of cycles related to generating random numbers (red circles), consuming them in a branch (blue dots), a memory access (yellow back slashes), or post-processing operation (green forward slashes), and cycles executed code segments that do not depend on stochastic values (gray). The mean is weighted by the number of Random Function Calls executed by the applications.

optimization [3, 51], modeling natural phenomena [1, 35, 53, 67], and financial forecasting [8, 24]. These programs rely on random values whose generation and use introduce significant challenges for modern computing systems. Stochastic values (SVs) may influence control flow or create data dependencies, but in all cases their inherent unpredictability conflicts with the predictable execution patterns favored by the Von Neumann model. As a result, traditional hardware optimizations struggle: branch predictors fail when SVs guide control flow [2, 37, 54], and data prefetchers lose effectiveness when SVs disrupt memory locality [13, 23]. These inefficiencies compound the already nontrivial cost of generating random values, leading to substantial performance overheads.

Figure 1 quantifies the performance impact of randomness across the applications studied in this work. Cycles dedicated to generating and consuming SVs are shown in colored and patterned segments, while non-SV-related cycles appear in gray. SV-related activity accounts for up to 83.8% of total execution cycles (e.g., pi) and averages 54.3% across benchmarks. Decomposing this overhead reveals four categories: SV generation, probabilistic branches, SV-dependent memory accesses, and arithmetic post-processing. Probabilistic branches contribute relatively little overhead (0.6% on average), whereas SV generation dominates (41.2% on average), post-processing can be significant (up to 6.4% in tailwag), and SV-dependent memory accesses are critical for some workloads (up to



This work is licensed under a Creative Commons Attribution 4.0 International License. <https://doi.org/10.1145/3801487.3801821>

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2568-5/26/05
<https://doi.org/10.1145/3801487.3801821>

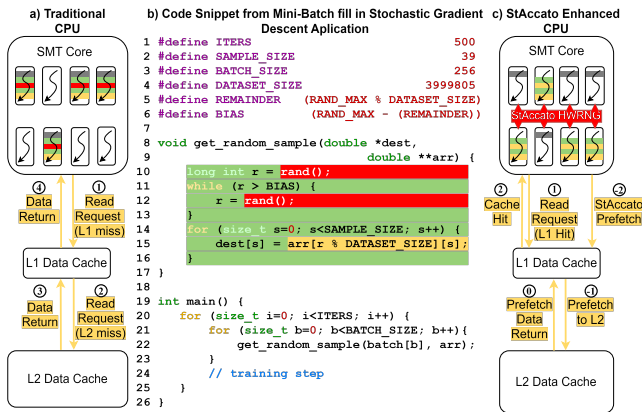


Figure 2: Overview of how StAccato accelerates stochastic workloads. Across all subfigures, parts shown in red relate to code segments or structures associated with generating a stochastic value, gray shows non-stochastic segments of the workload, green represents arithmetic operations with stochastic data dependencies, and yellow highlights memory accesses whose addresses depend on stochastic values. Subfigure b) shows a code snippet of the sgd workload. Subfigure a) depicts the execution of this code snippet on a traditional CPU. Finally, subfigure c) shows the accelerated execution of the code snippet due to StAccato.

45.1% in sgd). These observations motivate StAccato, which targets all major sources of SV-related overhead through hardware acceleration and compiler coordination. Unlike prior work that focused primarily on probabilistic control flow, StAccato addresses the full spectrum of random value-dependent computation, as reflected in our diverse benchmark suite.

With a growing interest in large-scale programs (whether data or compute-centric), many programmers take advantage of stochastic methods to improve overall runtime. One such class of programs that employ these techniques is search space reduction applications. A classic example of this is Stochastic Gradient Descent (SGD) [51], an optimization that significantly reduces the time to train a machine learning model by reducing the effective number of data samples needed for the error rate of a machine learning model to converge. This use of stochastic methods introduces both SV-dependent post-processing operations as well as memory dependencies on SVs. Data sampling optimizations come with the risk of unrepresentative draws from the complete dataset, which in the case of SGD, may yield a globally non-optimal solution. This concern is assuaged under the assumption that the sampling done as part of the "stochastic" aspect is sufficiently random. In other words, the risk of the dataset subset being biased is inversely related to the quality of the random number stream.

We propose StAccato, a novel system inspired by a hardware-software co-design process, illustrated in Figure 2. The code example in subfigure 2b), simplified from the sgd application, highlights random number generation overhead (red), post-processing of random numbers (green), and problematic (miss-inducing) memory accesses (yellow). Memory accesses based on random numbers lack both spatial and temporal locality and are much more likely to

generate cache misses than typical memory references. In modern processing cores, Simultaneous Multithreading (SMT) is supported via the duplication of hardware thread resources within a core. The processor in subfigure 2a) shows a processor with 8 available SMT contexts, four executing the code shown (those with colored-in thread blocks), and four idle threads. In this processor, the miss-inducing accesses incur heavy performance penalties, as requests must traverse down the cache hierarchy, ① and ②, before finding the data in main memory, and propagating the data back up through the hierarchy, ③ and ④. An SMT core enhanced with the StAccato modifications, shown in subfigure 2c), provides hardware acceleration for random number generation (HWRNG), as well as helper threads that provide low-overhead, concurrent execution of post-processing code and issue prefetches for problematic memory references. Figure 2c) illustrates how these miss-inducing accesses can be prefetched by a helper thread ahead of demand, ②, ③, and ④. When the demand access arrives ①, the data has been prefetched into the closest cache, realizing only a cache hit latency on the access ②. We show that StAccato’s HWRNG is fast and provides high-quality random numbers. Further, we show that the StAccato compiler can coordinate a helper thread to leverage StAccato’s lightweight HWRNG in a runahead feature to generate timely prefetches based on its precomputed addresses.

The major contributions of this work can be summarized as follows:

- A characterization of the negative performance impacts of relying on a pseudo-random number stream.
- A thorough investigation into various random number generation schemes, evaluating generator quality, storage costs, and latency.
- StAccato’s Hardware Random Number Generator (HWRNG): an RNG decoupled from a processor’s core that prevents programs with fundamental dependencies on random values from suffering the performance costs they incur.
- StAccato: a light-weight, software strategy that exploits the advantages of StAccato’s HWRNG to successfully remove the majority of RNG-dependent work off the critical path.
- A design for compiler support for StAccato: identifying minimal code changes needed to allow a stochastic workload to take full advantage of StAccato hardware when present.

The rest of this paper is organized as follows: Section 2 provides information on technologies and algorithms leveraged by this design, Section 3 describes the implementation details of StAccato and its compiler and helper thread support, Section 4 describes our methodologies for evaluation, Section 5 summarizes our results, and Section 6 concludes with the key findings of this work.

2 Background

2.1 Usage of Random Numbers

Monte Carlo methods are a prominent class of applications that rely on streams of random numbers [24, 45, 52, 53], providing a means to estimate quantities that are otherwise difficult to compute analytically. Prior work has shown that using random numbers to steer control flow introduces performance overheads [2]. Beyond

Table 1: Estimated Generator Characteristics

Generator	SW Latency (CPU cycles)	HW Latency (CPU cycles)	Memory Footprint (B)
Intel rdrand	65 [19]	65 [19]	Unknown
Intel rdseed	200 [19]	200 [19]	Unknown
LCG	4 [20, 42]	9-20 [6]	4 [20, 42]
mt19937	2 [33]	1 [6]	2496 [33, 41]
taus88	7 [36]	1 [6]	38-51 [6]

control flow, however, we observe that Monte Carlo applications incur additional costs from how stochastic values are used throughout execution.

One significant source of overhead arises from **Stochastic Memory Accesses (SMAs)**, where a memory address is computed from a random value. Because these addresses are data-dependent on stochastic inputs, they exhibit little correlation with prior accesses and are difficult to predict, leading to poor locality and increased memory latency. This pattern commonly appears in data sampling workloads such as `sgd` and `canneal`. In addition, applications frequently perform arithmetic transformations on random values, such as range scaling, to produce usable inputs for computation and memory indexing. These transformations further contribute to execution overhead, motivating architectural support that targets both stochastic memory accesses and post-processing operations.

2.2 Existing Random Number Generators

Pseudo-random number generators (PRNGs) are widely used in software due to their simplicity and reproducibility. Linear Congruential Generators (LCGs) generate values using a simple recurrence, $X_n = (A \cdot X_{n-1} + c) \bmod m$, and were historically adopted in C and C++ standard libraries because their behavior and period could be analytically characterized [20, 42]. However, LCGs exhibit a crystalline structure that makes their output predictable from few samples, rendering them unsuitable for high-quality randomness applications such as cryptography [39, 57, 64]. More sophisticated PRNGs, such as the Mersenne Twister (mt19937), improve equidistribution and offer a long period of $2^{19937} - 1$ [41], while hardware-oriented generators like Linear Feedback Shift Registers (LFSRs), also known as Tausworthe generators [59], provide space-efficient designs. In this work, we examine the Tausworthe88 (Taus88) generator [36, 59], which is used by the `leela` application.

Although LCGs, mt19937, and Taus88 are commonly used because they support deterministic seeding and reproducibility, this same property makes them highly predictable, as the seed is their sole source of entropy. To address this limitation, modern processors provide hardware-based randomness through dedicated instructions. Arm exposes `rndr` and `rndrrs`, while Intel and AMD provide `rdrand` for random number generation and `rdseed` for seeding custom generators. These instructions enable access to high-quality entropy directly from hardware. Table 1 summarizes the characteristics of the generators discussed and those used in our benchmarks.

2.3 RNG Quality Comparison

Measuring the quality of a random number generator is not straightforward, as there is no framework that can guarantee a generator

exhibits sufficient randomness. Dieharder 3.31.1 [11] is a testing suite for random number generators that provides insights into their quality. Within this framework, as the authors explicitly cautioned, positive results do not imply a perfect random number generator. It is the presence of negative results that assuredly indicate a poor generator. We use the default p-score threshold for Dieharder to determine if a test has passed, weakly passed, or failed. We focus on generators that either exist in modern processors, are used directly in the benchmarks in this study, or in the StAccato accelerator.

We examined the quality of values from `rdrand` and `rdseed` instructions in the Dieharder framework. The outcome of each of the tests is a grade that is evaluated based on the p-score determined by the test. If the test’s p-score satisfies a given threshold, an outcome of PASSED is reported for the test. When a test earns a WEAK assessment, this may indicate a flaw in the generator, if repeatable. A test report of FAILED is highly indicative of a poor generator. Each instruction was tested over 100 different trials, and both instructions were able to provide high-quality streams of random numbers. `rdrand` achieved PASSED on 97.6%, WEAK on 2.4%, and FAILED on 0.0% of tests across 100 trials on all 114 tests. Due to the high quality exhibited by the `rdrand` instruction, we use `rdrand` as a target in our RNG quality evaluation in Section 5.1.

2.4 Software Prefetching

Modern processor designs recognize that program access patterns may be unpredictable from the view of a data prefetcher. To permit the extraction of more performance from memory-intensive applications, ISAs often support a prefetch instruction. Such instructions also offer variants that configure what level of temporal locality the reference stream may exhibit and have been used to implement various forms of software prefetching [13, 23, 43]. For applications where probabilistic memory accesses may cause severe performance degradation, the use of a software-directed prefetch instruction may seem advantageous in improving performance. However, incorporating these prefetches does not yield large performance improvements and may not be portable across machines.

Performance measurements of N-ahead software prefetcher on the `sgd` application show that when the prefetch distance, N, is small ($N < 4$), we see performance improvements. However, as N increased, the prefetched lines were evicted before use, and the costs of issuing the prefetch requests outweighed any cache benefits they provided. With the optimal choice of N, the performance improvement was minimal (<2% reduction in execution time). The optimal prefetch distance changed on a different machine, confirming that N is a hardware-dependent, unpredictable variable.

2.5 Helper Threads

Chip-level multiprocessors (CMPs) exploit parallelism by duplicating hardware resources at the core level [7, 32], but their scalability is limited when synchronization costs dominate. Simultaneous multithreading (SMT) addresses this challenge by enabling parallel execution within a core through hardware thread duplication, leveraging wide issue windows and shared core structures to exploit instruction- and thread-level parallelism [62]. As a result, SMT has been widely adopted in commercial processors [28, 56, 61]. Beyond throughput gains, SMT enables concurrency strategies such as run-ahead or helper threads, where one thread executes ahead

to precompute or prefetch data and reduce stalls in the main thread [10, 15, 17, 18, 27, 30, 38, 49, 50, 63]. While effective, helper threads contend for shared hardware resources in fully utilized systems, limiting scalability. Dedicated helper engines or mini-cores have been proposed to avoid this contention [25, 40, 58, 66], but these approaches incur area overheads proportional to code complexity [16] and often require invasive programmer effort [66]. Our analysis of stochastic memory access and post-processing regions reveals diverse floating-point and integer instruction mixes, making specialized accelerators costly. Consequently, we find that leveraging helper threads on SMT cores provides an effective and low-overhead means of exploiting concurrency, and our evaluation in Section 5.2 shows that, when paired with StAccato's lightweight hardware, their benefits consistently outweigh resource contention overheads.

3 StAccato

The fundamental goals of the StAccato architecture aim to address all latencies associated with using a stream of stochastic values. In programs that rely on stochastic input, the usability of a random number generator largely depends on the quality of the random number stream and the speed at which the numbers can be produced. We first present the design of StAccato's Hardware Random Number Generator, which addresses these goals. We then present StAccato, a design that recruits a helper thread to leverage StAccato's lightweight, decoupled HWRNG to mitigate performance losses from post-processing operations and SMAs.

3.1 StAccato HWRNG

Figure 3 shows the hardware components of StAccato's Hardware Random Number Generator (HWRNG) architecture. StAccato's per-core HWRNG adapts the Taus88 generator to a more hardware-friendly format, similar to [29]. As such, the HWRNG consists of three 32-bit state registers (labeled "S1," "S2," and "S3" in Figure 3). We further enhance the Taus88 baseline design to allow for flexible reseeding. The Seed Queue buffers new seeds to replace the value in the S1 register. The Seed Queue is populated with seeds generated from a random seed generator, if one is present in default hardware. If not, the queue is filled directly by the programmer, which is advantageous when replaying a stochastic value stream is necessary. The SV Queue buffers the output of the stochastic number generator. These values are consumed when stochastic value is demanded from the core. The decoupling of the HWRNG with the core via queues creates exploitable concurrency, as the stochastic values can be precomputed before they are needed. The SV Queue provides a lookahead feature to read newly generated random numbers and retain them for later use, which is fundamental when StAccato orchestrates a helper thread to prefetch SMAs.

The logic for updating the state registers in StAccato's HWRNG is governed by equations 1 - 3. Equation 1 defines the recurrence for updating state register S1. Included is a datapath to allow for dynamic reseeding into S1. In the common case, where no reseeding is necessary, S1 is simply updated by a bit rearrangement and partial self-xor operation (xor operations are denoted by \oplus , $[]$ indicate bit-slicing operations, and $\{\}$ indicates concatenation of bits). State registers S2 and S3 are updated similarly, as defined by equations 2 and 3 respectively. Equation 4 defines the generator's output as

a function of the state registers. Seeding control of state registers S2 and S3 is indirect via seeding of S1, as is common practice for RNGs with multiple state registers [41, 44, 59]. Like the Taus88 RNG [36, 59], StAccato's HW RNG requires non-zero initialization of state registers. When S1 is reseeded, S2 and S3 are reset to known, non-zero values ('d8 and 'd16, respectively, in our experiments).

$$S1 = \text{reseed} ? \text{Seed} : \{S1[19:1], S1[18:6] \oplus S1[31:19]\}; \quad (1)$$

$$S2 = \{S2[27:3], S2[29:23] \oplus S2[31:25]\}; \quad (2)$$

$$S3 = \{S3[31:21], S3[28:8] \oplus S3[31:1]\}; \quad (3)$$

$$\text{Output} = S1 \oplus S2 \oplus S3; \quad (4)$$

3.2 StAccato Architecture

Modern systems often place random number generation on the critical path of execution, even though many stochastic workloads rely on independent and identically distributed (IID) samples. This results in unnecessary latency, as these values can be generated and consumed asynchronously. StAccato addresses this inefficiency by decoupling the production and use of stochastic values from the critical path. While StAccato's hardware random number generator (HWRNG) reduces the latency of value generation, Figure 1 shows that downstream computations, such as arithmetic transformations on stochastic values and their use in memory address calculations, still consume a significant portion of runtime. Our work targets these remaining bottlenecks by shifting them off the critical path to better exploit the inherent independence of the random stream and improve overall performance.

3.2.1 StAccato for Arithmetic Operations. The output of a random number generator is often not directly usable, as applications typically require scaling and format conversion. For example, an ideal 32-bit generator produces uniformly distributed unsigned integers in the range $[0, 4,294,967,295]$, which applications may transform to narrower ranges or convert to floating-point values. In the dop benchmark, uniformly distributed random numbers are further transformed into a normal distribution using the Box-Muller transform [9]. StAccato improves performance by coordinating a helper

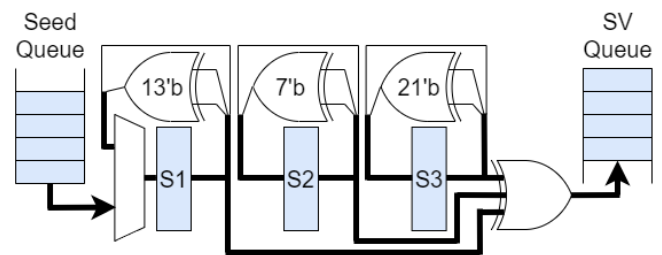


Figure 3: StAccato's 32-bit hardware random number generator. Bold lines represent wires of full 32-bit widths. Blue-labeled components represent the three 32-bit registers that encapsulate the state of the generator. Next state logic consists of XOR gates of varying widths. We augment the datapath with select logic to allow reseeding from an external component. Output logic only consists of 32 three-input XOR gates, allowing for fast number generation on demand.

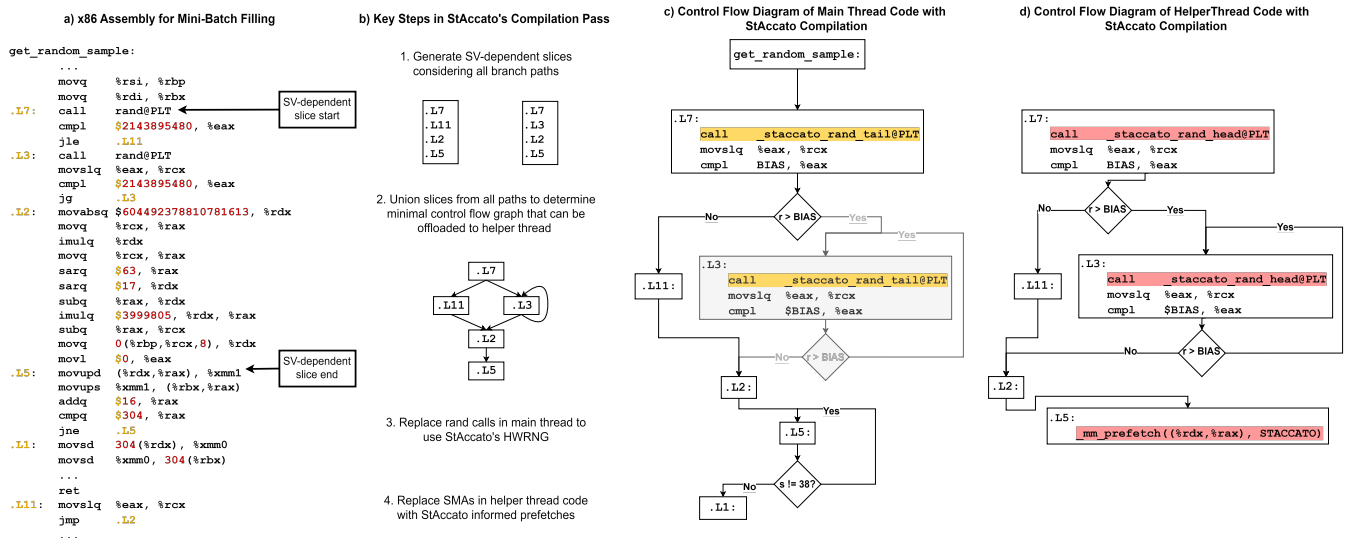


Figure 4: Automated changes to program control flow graph to utilize StAccato hardware. Subfigure a) shows the x86 assembly instructions from the code snippet shown in Figure 2. For clarity, some code portions have been removed. Subfigure b) shows the key steps performed when recruiting a helper thread. Subfigures c) and d) show the alterations to the control flow graphs to support StAccato for the main and helper threads' execution, respectively. Each box represents a basic block. Changes highlighted in yellow are needed to use StAccato's HWRNG alone, and those highlighted in red allow for further performance gains via StAccato's recruitment of a helper thread. Components highlighted in gray are skipped when the application runs in StAccato's helper thread due to perfect prefetching of SMAs and precomputation.

thread to lift these heavy-weight transformations off the critical path. To support this, the application allocates a small amount of additional storage for precomputed results, but this overhead is negligible, as only a few computations are required to stay sufficiently ahead of the main thread.

3.2.2 StAccato for Prefetching SMAs. Beyond post-processing operations, StAccato's helper thread also targets inherently unpredictable memory access patterns common in applications such as `sgd` and `canneal`, where large data structures are stochastically indexed to enable convergence or exploration. These workloads operate on data sets significantly larger than the L1 cache, with memory footprints of 1.2GB and 800KB, respectively, and exhibit little locality due to stochastic access patterns. In such cases, explicit prefetching is beneficial, and StAccato's helper thread mitigates the cost of stochastic memory accesses by leveraging its fast, decoupled HWRNG, which exposes both the current and future stochastic values consumed by the program. This enables the helper thread to issue prefetches N accesses ahead of demand, ensuring that prefetched cache lines will be used. Unlike traditional hardware prefetchers, which may speculatively fetch unused data due to limited control-flow awareness, StAccato's prefetches are correct by construction. To prevent premature eviction, StAccato reserves two entries in the data prefetch buffer, augmented with a single bit indicating StAccato ownership, ensuring that prefetched cache lines remain resident until consumed by the main thread.

3.3 Programmer Support

Minimal ISA changes are needed to provide programmer support for StAccato. StAccato provides the capability for a programmer

to seed StAccato's stochastic number generator when desired by augmenting the `rdseed` instruction with an optional source register.

RDSEED Rdest, Rsrc

Normally, the `rdseed` instruction only uses the destination register `Rdest` to return a seed from the hardware seed generation unit. If `Rsrc` is specified, it is fed into the Seed Queue of StAccato's random number generator. Next, we propose a similar augmentation to the `rdrand` instruction:

RDRAND Rdest, END {HEAD, TAIL}

In this augmented format, `Rdest` specifies the target register for the stochastic value, and `END` is an operand that guides which value is retrieved from the SV queue. When polled from the tail, the head of SV Queue is populated with a new value from StAccato's HWRNG. The modified instruction provides the necessary programmer support for users to utilize StAccato's HWRNG to reduce the performance impact of randomness.

However, StAccato is a strategy that targets other portions of work that depend on stochastic values. When a helper thread is utilized to precompute and prefetch work dependent on a stochastic value, `END` can be set to `HEAD` to retrieve the most recent stochastic value from StAccato's random number generator. To support StAccato's helper thread in alleviating the performance costs from applying arithmetic transforms to the random values and those stochastic memory accesses, we augment the `prefetch` instruction:

PREFETCHh Rsrc, Hint {T0, T1, T2, NTA, STACCATO}

As with the unmodified form of this instruction, invoking this instruction initiates a memory request for a cache line for the address specified in the source operand to a location directed by the hint. In

Table 2: Benchmark Summary

Benchmark	RNG Scheme	Random Values Needed	Sim. Instrs	Stochastic Influence
dop	LCG	51M	7.96B	70.1%
photon	LCG	33M	6.46B	62.3%
pi	LCG	20M	1.77B	83.9%
genetic	LCG	16M	4.97B	58.3%
sgd	LCG	1.28M	57.5M	56.3%
2048	LCG	558	1.18M	4.58%
swaptions	LCG	2.4M	719M	11.3%
canneal	mt19937	605K	244M	15.1%
leela	taus88	35M	15.3B	6.07%
homophonic	LCG	16K	3.64M	26.6%
tailwag	LCG	16K	3.04M	15.3%

modern processors, four types of hints are currently supported: T0, T1, T2, and NTA for fetching into all caches, L2 caches and higher, L3 caches and higher, or as a non-temporal load, respectively. We augment this instruction to also support the hint STACCATO. When used, this hint informs the prefetcher to demarcate the newly allocated prefetch buffer entry as an SMA. This ensures that this entry is moved into a non-next-to-be-evicted line in the cache when the data is returned.

3.4 StAccato Compiler Support

To facilitate adoption, we present a compiler flow that applies required code transformations without manual programmer effort. The compiler introduces three new intrinsics and extends an existing one to expose the ISA changes described in Section 3.3: `_staccato_rand_tail()` and `_staccato_rand_head()` wrap the RDRAND instruction with TAIL and HEAD endpoints, `_mm_prefetch()` is extended with a STACCATO hint, and `_staccato_rand_seed()` seeds the hardware RNG. An additional compiler pass tracks the liveness of values produced by `rand()` to identify stochastic value dependent instruction slices that terminate at arithmetic post-processing or stochastic memory address generation. As shown in Figure 4 using the `sgd` benchmark, the compiler replaces `rand()` calls on the main thread with `_staccato_rand_tail()`, generates a helper thread that executes SV-dependent work ahead of demand using `_staccato_rand_head()`, and converts stochastic memory accesses into software-directed prefetches, allowing StAccato to hide generation, processing, and memory access latency with minimal code changes.

3.5 Total Storage Cost

StAccato’s generator consists of three, 32-bit wide state registers. StAccato can be reseeded, and need only be reseeded at low rate in order to match high quality, cryptographically-secure PRNGs, like `rdrand`. To ensure StAccato’s generator maintains a level of quality similar to `rdrand`, the Seed Queue contains at least 2 entries when utilizing a native seed generator like `rdseed` for new seeds.

While StAccato’s HWRNG eliminates the software cost of generating a stochastic value, further support is needed for StAccato.

Table 3: Sniper Configurations

Core	Intel x86-64 Sandy Bridge
	single core, 4-wide
L1 I&D Caches	Private, 8-way, 32KB, 4 cycle data access
L2 Cache	Unified, 8-way, 256KB, 8 cycle data access
L3 Cache	16-way, 8MB, 30-cycle data access
DTLB	4-way, 64KB
ITLB	4-way, 128KB
STLB	4-way, 512KB
Branch Predictor	8K TAGE-SC-L [54]

The helper thread in StAccato relies on looking ahead of the current stochastic value out of the generator in order to prefetch and precompute future SMAs and stochastic value-dependent tasks. We find that 8 entries of the SV Queue suffice to ensure that the helper thread can stay ahead of the main program’s consumption of stochastic values. Thus, StAccato requires 8 bytes of storage for the Seed Queue, 12 bytes for the generator’s state registers, and 32 bytes for the SV Queue, bringing the total footprint to 52 bytes of storage.

4 Methodology

4.1 Benchmarks

The selected benchmarks were retrieved from various open-source projects. Like the most relevant prior work [2], we examine a simplified version of the Black-Scholes application from the PARSEC 3.0 benchmark suite [8]: `dop`, from [24]. `photon` uses a Monte Carlo simulation to model the movement of light particles (photons) through a translucent slab [53] and uses the Henyey-Greenstein phase scattering model to estimate spin changes to the particles [26]. `pi` is a Monte Carlo application that approximates the mathematical constant π [45]. `2048` is a command line implementation of Gabriele Cirulli’s game, `2048` [60]. `genetic` is an application that runs a genetic algorithm to produce a string representation of an equation that solves to a specific target [12]. The `sgd` application performs mini-batched, stochastic gradient descent on a flight database from the US Department of Transportation [55]. Other benchmarks examined from the PARSEC 3.0 benchmark suite [8] included `swaptions` and `canneal`. `swaptions` is a financial analysis program that uses a Monte Carlo simulation to price a portfolio of swaptions [8]. `canneal` is a computer-aided design (CAD) program that uses simulated annealing to produce a minimal-cost routing design for a chip design. From the 2017 SPEC Benchmark suite, `leela` [47] is a game engine for the board game Go. `homophonic` is an encryption program based on the homophonic substitution cipher [31]. Finally, `tailwag` is an application that probabilistically models server workloads [67]. Table 2 provides additional information on the benchmarks.

4.2 Quality

We performed a series of trials to quantify an RNG’s quality as a distance from a known, high-quality RNG, e.g. `rdrand`. To quantify this, we vary the rate at which an RNG strategy needs to be reseeded in order to meet the same quality of results as Dieharder

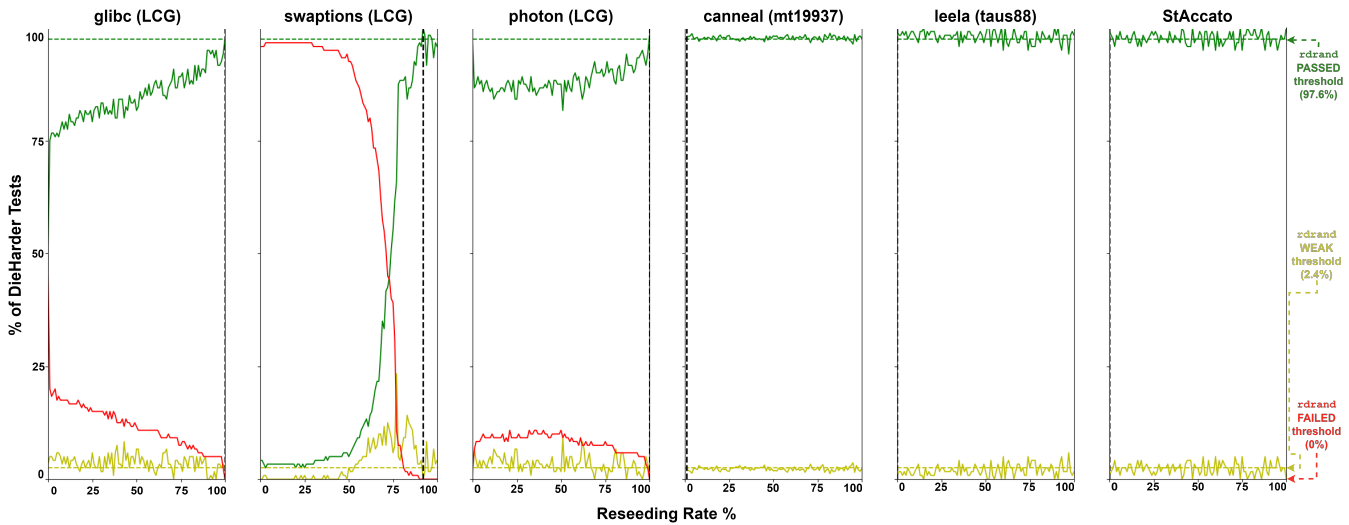


Figure 5: Dieharder results from select random number generators. The horizontal axis indicates a run of the Dieharder test suite varying the starting seed for the generator in question. The frequency of reseeding the generator is examined from 0% to 100%. Green, yellow, and red solid lines indicate the number of PASSED, WEAK, and FAILED tests, respectively. Dotted colored lines indicate quality from `rdrand`. The dotted black line indicates the level of reseeding needed to achieve `rdrand` quality.

produced from `rdrand`. To reseed the generators, we use a high-quality seed generator, namely, `rdseed`. For more details on these two instructions, please refer back to Section 2.3.

4.3 Performance Modelling

To evaluate the performance of our proposed accelerator, we use a combination of the parallel, multi-core, x86 Sniper 7.4 [14] simulator in addition to an in-house analytical model.

4.3.1 Simulator. Table 3 shows the configurations used for our baseline model. The branch predictor’s misprediction penalty was modeled as 10 cycles. All levels of caches use an LRU eviction policy, have 64B blocks, and incur a single cycle for tag accesses. The simulator was modified to tag and categorize cycles that executed SV-dependent code. All benchmarks were executed with their default RNG strategy, as noted in Table 2.

4.3.2 Analytical Model. To model the performance improvements offered by StAccato, we isolate and measure each major source of stochastic value (SV)-related overhead: random number generation, arithmetic processing of SVs, and SV-based memory accesses. Experiments were conducted on real hardware under a contention free model as well as under realistic conditions using the Linux stress utility to simulate contention. The stress command, and its variant stress-ng, have commonly been used to emulate realistic contention models [4, 21, 48]. Measurements were collected via the `_rdtscp()` intrinsic on an 8-core, 2.2 GHz SMT-enabled machine, with 2 SMT contexts per core. By introducing realistic resource contention, stress implicitly models high core utilization, providing insight into how StAccato’s helper-thread performance may scale under heavily loaded systems.

SV generation: StAccato’s hardware RNG reduces the cost of generating a stochastic value to a single cycle.

Post-processing SVs: To evaluate helper-thread benefits, we examined dop, which features the longest post-processing segment.

In two experiments, we first removed random number generation to establish a baseline, and then offloaded post-processing to a helper thread. The difference isolates the performance gain achievable through helper-thread acceleration.

SV-based memory access: We evaluated the sgd application in three configurations under simulated contention: (1) single-threaded with software-directed prefetching to find the optimal distance, (2) two-threaded with a helper issuing prefetches ahead of the main thread, and (3) a version with non-productive helper activity to quantify the cost of helper-thread usage. These experiments allow us to model the tradeoff between prefetching benefit and helper overhead.

5 Results

5.1 Stochastic Quality

Figure 5 depicts the results of Dieharder’s 114 tests for the 6 random number generation schemes. The horizontal axis indicates the rate at which the generator is reseeded from `rdseed`, while the vertical axis shows the distribution of assessments from the Dieharder tests. Solid green lines indicate the percentage of Dieharder tests that earned a PASSED assessment, solid yellow lines indicate WEAK assessments, and solid red lines indicate FAILED assessments. The dotted green and yellow lines indicate the quality performance from the `rdrand` instruction. Dashed black lines indicate what level of reseeding is necessary to equal the `rdrand`’s random number generation quality.

As seen in Figure 5, the LCGs require a very high level of reseeding to achieve quality that matches `rdrand`. Both `glibc` and `drand48()` require reseeding operations in 99.9% stochastic value generations, with the `swaption`’s generator requiring a reseed rate of 92%. In contrast, `mt19937`, `taus88`, and StAccato require infrequent reseeding operations to match `rdrand`, requiring reseed rates of 2.0%, 1.0%, and 1.0%, respectively.

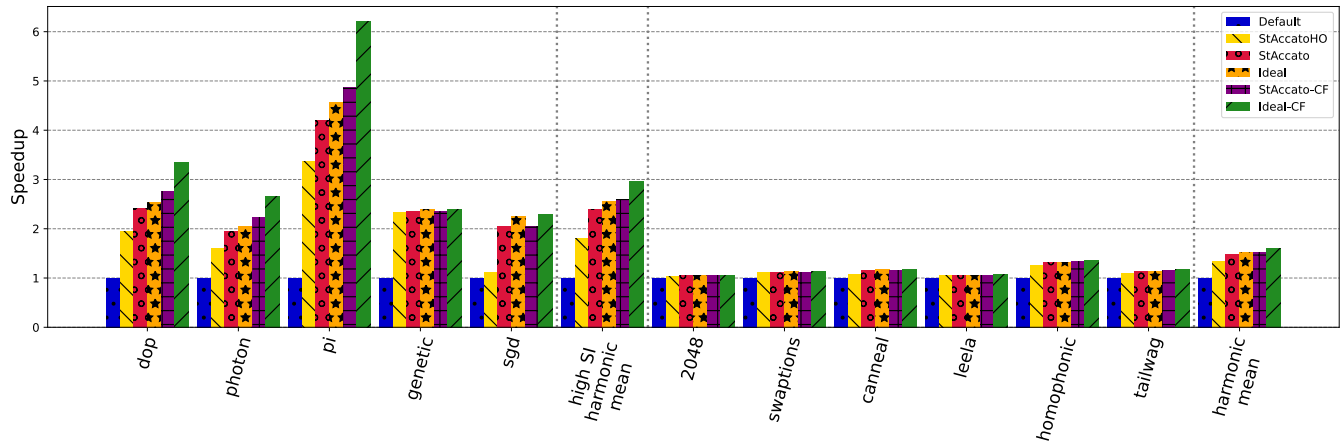


Figure 6: Performance improvement of StAccato. Default is shown in blue (dots), StAccato’s HWRNG Only is shown in yellow (back slashes), StAccato under stress is shown in red (circles), Ideal under stress is shown in orange (stars), StAccato under contention free model in purple (grid notations), and Ideal under contention free model (green forward slashes). Label “high SI harmonic mean” reports the harmonic mean over the subset of applications that exhibit high stochastic influence, designated by >50.0% in table 2. Label “harmonic mean” reports the harmonic mean speedups across all applications.

5.2 Performance

We find that StAccato can lift 86.4% and 73.4% of the time dedicated to arithmetic operations and transforms off the critical path of the program under multithreaded contention and contention-free models, respectively. Based on our experiments on the *sgd* application described in Section 4.3.2, we find that StAccato can hide 89.7% of the time dedicated to servicing stochastic memory accesses.

Figure 6 shows our speedups of StAccato. We see an harmonic mean speedup of 1.34x from StAccato’s HWRNG alone (labeled as ‘StaccatoHO’) across our selected benchmarks. In applications where the cost of producing a random value is small relative to the rest of the program, as in *2048*, *sgd*, *canneal*, *leela*, and *tailwag*, the speedups are less significant. However, in applications where generating a stochastic value represents a significant portion of the program, we see highly pronounced speedups, as in *dop*, *photon*, and *genetic*, seeing as large as a 3.36x speedup in *pi*. In applications heavily influenced by stochasticity (where stochastic influence >50.0% as report in Table 2), as in *dop*, *photon*, *pi*, and *sgd*, StAccato’s HWRNG provides a 1.81x speedup. Through the recruitment of helper threads StAccato provides additional performance improvements. Labels “StAccato” and “StAccato-CF” annotate these improvements for experiments using multithreaded contention and contention free models. Under multithreaded contention, StAccato provides 2.40x speedup for applications with high stochastic influence and 1.49x improvements across all workloads. When SMT resources are readily available (contention free) StAccato provides 2.60x speedup for applications with high stochastic influence and 1.53x improvements across all workloads. Performance in contention free experiments is marginally worse as the main thread’s execution is faster, diminishing the amount of helper thread latency that can be hidden.

To better evaluate the efficacy of StAccato in successfully eliminating and hiding latencies associated with data dependencies on stochastic values, we model the theoretical limit that can be achieved when 100% of the stochastic value-dependent work can

be lifted off the critical path of an application. In Figure 6, these are denoted by the bars in orange with stars labeled “Ideal” and in green with forward slashes labeled “Ideal-CF,” where “CF” notates contention free model. Under contention free settings we see that using StAccato’s HWRNG alone can realize 61.1% and 71.0% of ideal speedup for the high stochastic influenced subset of workloads and across all workloads, respectively. By targeting additional portions of random number-dependent work beyond generating a stochastic value using helper threads, StAccato can achieve 94.2% and 87.8% of the ideal speedup for the subset of workload with high stochastic influence under multithreaded contention and contention free models, respectively. Across all workloads, StAccato achieves 97.9% and 95.8% for the same two workload designations.

6 Conclusion

In this work, we present StAccato, a hardware-software co-designed accelerator with complete compiler support to target the performance loss due to randomness. We see that using StAccato’s fast, processor-decoupled, lightweight, high-quality HWRNG provides a 1.34x speedup across our selected benchmarks. StAccato’s HWRNG successfully eliminates long-latency operations needed to produce a stream of stochastic values programmatically. With compiler support, a helper thread can be used to realize further performance gains by targeting remaining stochastic value-dependent work and lifting these operations off the critical path of the program. StAccato’s helper thread successfully hides the latency of arithmetic operations performed to transform stochastic values as well as latencies of memory accesses whose addresses depend on stochastic values. Under realistic contention for multithreaded resources, StAccato achieves a 1.49x speedup over the baseline applications, exploiting 97.9% of realizable speedup.

References

- [1] Emile Aarts and Jan Korst. 1989. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Inc., USA.

- [2] Almutaz Adileh, David J. Lilja, and Lieven Eeckhout. 2018. Architectural Support for Probabilistic Branches. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Fukuoka, Japan, 108–120. doi:10.1109/MICRO.2018.00018
- [3] Benjamin Antunes and David R. C. Hill. 2026. Random Numbers for Machine Learning: A Comparative Study of Reproducibility and Energy Consumption. *Journal of Data Science and Intelligent Systems* 4, 1 (Jan. 2026), 26–38. doi:10.47852/bonviewJDSIS42024012
- [4] Mohammad S. Aslanpour, Adel N. Toosi, Raj Gaire, and Muhammad Aamir Cheema. 2021. WattEdge: A Holistic Approach for Empirical Energy Measurements in Edge Computing. In *Service-Oriented Computing*, Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik (Eds.). Springer International Publishing, Cham, 531–547.
- [5] Vittorio Bagini and Marco Bucci. 1999. A Design of Reliable True Random Number Generator for Cryptographic Applications. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*. Springer-Verlag, Berlin, Heidelberg, 204–218.
- [6] Mohammed Bakiri, Christophe Guyeux, Jean-François Couchot, Luigi Marangio, and Stefano Galatolo. 2018. A Hardware and Secure Pseudorandom Generator for Constrained Devices. *IEEE Transactions on Industrial Informatics* 14, 8 (2018), 3754–3765. doi:10.1109/TII.2018.2815985
- [7] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzky, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. 2000. Piranha: a scalable architecture based on single-chip multiprocessing. *SIGARCH Comput. Archit. News* 28, 2 (may 2000), 282–293. doi:10.1145/342001.339696
- [8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. Technical Report TR-811-08. Princeton University.
- [9] George Edward Pelham Box and Mervin E. Muller. 1958. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics* 29, 2 (1958), 610–611. doi:10.1214/aoms/1177706645
- [10] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. 2002. Speculative Precomputation on Chip Multiprocessors.
- [11] Robert G. Brown, Dirk Edelbuettel, and David Bauer. 2020. Dieharder: A Random Number Test Suite Version 3.31.1. <https://scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-simulation.html> Accessed on November 22nd, 2024.
- [12] Mat Bucklan. 2013. Genetic Algorithm Example. <https://www.codemiles.com/c-examples/genetic-algorithm-example-17548.html> Accessed on November 22nd, 2024.
- [13] Brendon Cahoon and Kathryn S. McKinley. 2001. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*. IEEE Computer Society, USA, 280–291.
- [14] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, United States, 1–12. doi:10.1145/2063384.2063454
- [15] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*. IEEE, Atlanta, GA, USA, 186–195. doi:10.1109/ISCA.1999.765950
- [16] Shailender Chaudhry, Paul Caprioli, Sherman Yip, and Marc Tremblay. 2005. High-performance throughput computing. *IEEE Micro* 25, 3 (2005), 32–45. doi:10.1109/MM.2005.49
- [17] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, Gothenburg, Sweden, 14–25. doi:10.1109/ISCA.2001.937427
- [18] Michel Dubois. 2004. Fighting the memory wall with assisted execution. In *Proceedings of the 1st Conference on Computing Frontiers (Ischia, Italy) (CF '04)*. Association for Computing Machinery, New York, NY, USA, 168–180. doi:10.1145/977091.977116
- [19] Agner Fog. 2022. Instruction Tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. <https://www.agner.org/optimize/> Accessed on November 22nd, 2024.
- [20] GNU. 2022. drand48-iter.c implementation. https://codebrowser.dev/glibc/glibc/stdlib/drand48-iter.c.html#_drand48_iterate Accessed on November 22nd, 2024.
- [21] Yanan Guo, Dingyuan Cao, Xin Xin, Youtao Zhang, and Jun Yang. 2023. Uncore Encore: Covert Channels Exploiting Uncore Frequency Scaling. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (Toronto, ON, Canada) (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 843–855. doi:10.1145/3613424.3614259
- [22] Z. Gutterman, B. Pinkas, and T. Reinman. 2006. Analysis of the Linux random number generator. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, Berkeley/Oakland, CA, USA, 15 pp.–385. doi:10.1109/SP.2006.5
- [23] Ioan Hadade, Timothy M. Jones, Feng Wang, and Luca di Mare. 2020. Software Prefetching for Unstructured Mesh Applications. *ACM Trans. Parallel Comput.* 7, 1, Article 3 (mar 2020), 23 pages. doi:10.1145/3380932
- [24] Michael Halls-Moore. 2013. Digital option pricing with C++ via Monte Carlo methods. <https://www.quantstart.com/articles/Digital-option-pricing-with-C-via-Monte-Carlo-methods/> Accessed on November 22nd, 2024.
- [25] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Taipei, Taiwan, 1–12. doi:10.1109/MICRO.2016.7783764
- [26] Louis George Henyey and Jesse Leonard Greenstein. 1941. Diffuse radiation in the Galaxy. *Astrophysical Journal* 93 (Jan. 1941), 70–83. doi:10.1086/144246
- [27] Andrew Hilton and Amir Roth. 2010. BOLT: Energy-efficient Out-of-Order Latency-Tolerant execution. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, Bangalore, India, 1–12. doi:10.1109/HPCA.2010.5416634
- [28] S Jarp, A Lazzaro, J Leduc, and A Nowak. 2012. *Evaluation of the Intel Sandy Bridge-EP server processor*. Technical Report. CERN, Geneva. <https://cds.cern.ch/record/1434748>
- [29] Yutana Jewajinda. 2014. Parallel hardware architecture and FPGA implementation of a differential evolution algorithm. In *TENCON 2014 - 2014 IEEE Region 10 Conference*. IEEE, Bangkok, Thailand, 1–4. doi:10.1109/TENCON.2014.7022429
- [30] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (Newport Beach, California, USA) (ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 393–404. doi:10.1145/1950365.1950411
- [31] Ravee Khandagale. 2017. Homophonic-Substitution-Cipher. <https://github.com/enRaved/Homophonic-Substitution-Cipher>
- [32] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro* 25, 2 (2005), 21–29. doi:10.1109/MM.2005.35
- [33] Shiro Konuma and Shuichi Ichikawa. 2005. Design and Evaluation of Hardware Pseudo-Random Number Generator MT19937* This Work Partially Appeared as an Extended Abstract in the 2005 Annual Meeting Record IEE Japan, Vol.3, Pp.89–90 (March 2005). *IEICE - Trans. Inf. Syst.* E88-D, 12 (dec 2005), 2876–2879. doi:10.1093/ietisy/e88-d.12.2876
- [34] Sammy H. M. Kwok and Edmund Y. Lam. 2006. FPGA-based High-speed True Random Number Generator for Cryptographic Applications. In *TENCON 2006 - 2006 IEEE Region 10 Conference*. IEEE, Hong Kong, China, 1–4. doi:10.1109/TENCON.2006.344013
- [35] S. H. Lam. 1992. On the RNG theory of turbulence. *Physics of Fluids A: Fluid Dynamics* 4, 5 (05 1992), 1007–1017. arXiv:https://pubs.aip.org/aip/pof/article-pdf/4/5/1007/12263755/1007_1_online.pdf doi:10.1063/1.858517
- [36] Pierre L'Ecuyer. 1996. Maximally Equidistributed Combined Tausworthe Generators. *Math. Comp.* 65, 213 (1996), 203–213. <http://www.jstor.org/stable/2153840>
- [37] Edward Lee and Craig Zilles. 2008. Branch-on-Random. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (Boston, MA, USA) (CGO '08)*. Association for Computing Machinery, New York, NY, USA, 84–93. doi:10.1145/1356058.1356070
- [38] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. 2005. Dynamic helper threaded prefetching on the Sun UltraSPARC/spl reg/ CMP processor. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, Barcelona, Spain, 12 pp.–104. doi:10.1109/MICRO.2005.18
- [39] George Marsaglia. 1968. Random Numbers Fall Mainly in the Planes. *Proceedings of the National Academy of Sciences* 61, 1 (1968), 25–28. arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.61.1.25 doi:10.1073/pnas.61.1.25
- [40] Susumu Mashimo, Ryota Shioya, and Koji Inoue. 2020. Energy Efficient Runahead Execution on a Tightly Coupled Heterogeneous Core. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (Fukuoka, Japan) (HPCA Asia '20)*. Association for Computing Machinery, New York, NY, USA, 207–216. doi:10.1145/3368474.3368496
- [41] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (jan 1998), 3–30. doi:10.1145/272991.272995
- [42] Roland McGrath and Ulrich Drepper. 1995. random_r.c implementation. https://sourceware.org/git/?p=glibc.git;a=blob:f=stdlib/random_r.c;hb=glibc-2.28#1353 Accessed on November 22nd, 2024.
- [43] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Massachusetts, USA) (ASPLOS V)*. Association for Computing Machinery, New York, NY, USA, 62–73. doi:10.1145/143365.143488
- [44] Melissa E. O'Neill. 2014. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. <https://www.cs.hmc.edu/tr/>

- hmc-cs-2014-0905.pdf
- [45] Geroge Ostrouchov, Preston Shires, , and Ryan M. Adamson. 2013. Serial to parallel: Monte Carlo operation. <https://github.com/olcf/Serial-to-Parallel-Monte-Carlo-Pi/blob/master/serialpi.c> Accessed on November 22nd, 2024.
- [46] Fabio Pareschi, Gianluca Setti, and Riccardo Rovatti. 2006. A Fast Chaos-based True Random Number Generator for Cryptographic Applications. In *2006 Proceedings of the 32nd European Solid-State Circuits Conference*. IEEE, Montreaux, Switzerland, 130–133. doi:10.1109/ESSCIR.2006.307548
- [47] Gian-Carlo Pascutto. 2020. SPEC 2017 541.leela_r. https://www.spec.org/cpu2017/Docs/benchmarks/541.leela_r.html Accessed on November 22nd, 2024.
- [48] Maryam Pourreza and Priya Narasimhan. 2023. An Empirical Study of Resource-Stressing Faults in Edge-Computing Applications. In *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking (Rome, Italy) (EdgeSys '23)*. Association for Computing Machinery, New York, NY, USA, 54–59. doi:10.1145/3578354.3592873
- [49] Tanasú Ramírez, Alex Pajuelo, Oliverio Jesus Santana, Onur Mutlu, and Mateo Valero. 2010. Efficient runahead threads. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (Vienna, Austria) (PACT '10)*. Association for Computing Machinery, New York, NY, USA, 443–452. doi:10.1145/1854273.1854328
- [50] Tanasú Ramírez, Alex Pajuelo, Oliverio J. Santana, and Mateo Valero. 2008. Runahead Threads to improve SMT performance. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, Salt Lake City, UT, USA, 149–158. doi:10.1109/HPCA.2008.4658635
- [51] Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. *The Annals of Mathematical Statistics* 22, 3 (1951), 400 – 407. doi:10.1214/aoms/117729586
- [52] Christian de Schryver, Ivan Shcherbakov, Frank Kienle, Norbert Wehn, Henning Marxen, Anton Kostiuk, and Ralf Korn. 2011. An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model. In *2011 International Conference on Reconfigurable Computing and FPGAs*. IEEE, Cancun, Mexico, 468–474. doi:10.1109/ReConFig.2011.11
- [53] Scratchapixel. 2022. Monte Carlo Methods in Practice. <https://scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-simulation.html> Accessed on November 22nd, 2024.
- [54] André Seznec. 2016. TAGE-SC-L branch predictors again.
- [55] Daniel Sharp. 2018. Implementation of Stochastic Gradient Descent in CUDA. <https://dsharp.github.io/SGD/> Accessed on November 22nd, 2024.
- [56] Teja Singh, Alex Schaefer, Sundar Rangarajan, Deepesh John, Carson Henrion, Russell Schreiber, Miguel Rodriguez, Stephen Kosonocky, Samuel Naffziger, and Amy Novak. 2018. Zen: An Energy-Efficient High-Performance × 86 Core. *IEEE Journal of Solid-State Circuits* 53, 1 (2018), 102–114. doi:10.1109/JSSC.2017.2752839
- [57] Jacques Stern. 1987. Secret linear congruential generators are not cryptographically secure. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. IEEE, Los Angeles, CA, USA, 421–426. doi:10.1109/SFCS.1987.51
- [58] Bharath Narasimha Swamy, Alain Ketterlin, and André Seznec. 2014. Hardware/Software Helper Thread Prefetching on Heterogeneous Many Cores. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, Paris, France, 214–221. doi:10.1109/SBAC-PAD.2014.39
- [59] Robert C. Tausworthe. 1965. Random Numbers Generated by Linear Recurrence Modulo Two. 201–209 pages. doi:10.1090/S0025-5718-1965-0184406-1
- [60] Marc Tiehuis. 2019. 2048-cli. <https://github.com/tiehuis/2048-cli> Accessed on November 22nd, 2024.
- [61] Nathan Tuck and Dean M. Tullsen. 2003. Initial observations of the simultaneous multithreading Pentium 4 processor. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, New Orleans, LA, USA, 26–34. doi:10.1109/PACT.2003.1237999
- [62] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News* 23, 2 (may 1995), 392–403. doi:10.1145/225830.224449
- [63] Kenzo Van Craeynest, Stijn Eyerman, and Lieven Eeckhout. 2009. MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor. In *High Performance Embedded Architectures and Compilers*. André Seznec, Joel Emer, Michael O'Boyle, Margaret Martonosi, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 110–124.
- [64] I. Vattulainen, K. Kankaala, J. Saarinen, and T. Ala-Nissila. 1995. A comparative study of some pseudorandom number generators. *Computer Physics Communications* 86, 3 (1995), 209–226. doi:10.1016/0010-4655(95)00015-8
- [65] Vincent von Kaenel and Toshinari Takayanagi. 2007. Dual True Random Number Generators for Cryptographic Applications Embedded on a 200 Million Device Dual CPU SoC. In *2007 IEEE Custom Integrated Circuits Conference*. IEEE, San Jose, CA, USA, 269–272. doi:10.1109/CICC.2007.4405730
- [66] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (Williamsburg, VA, USA) (ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 593–607. doi:10.1145/3173162.3173197
- [67] Heng Zhuo and Mikko Herman Lipasti. 2023. TailWAG: Tail Latency Workload Analysis and Generation. In *Proceedings of the 5th International Workshop on Benchmarking in the Data Center (Montreal, QC, Canada) (BID '23)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. doi:10.1145/3583060.3583170