QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips

Ataberk Olgun[§†] Minesh Patel[§] A. Giray Yağlıkçı[§] Haocong Luo[§]
Jeremie S. Kim[§] F. Nisa Bostancı[§†] Nandita Vijaykumar[§⊙] Oğuz Ergin[†] Onur Mutlu[§]

[§]*ETH Zürich* [†]*TOBB University of Economics and Technology* [⊙]*University of Toronto*

True random number generators (TRNG) sample random physical processes to create large amounts of random numbers for various use cases, including security-critical cryptographic primitives, scientific simulations, machine learning applications, and even recreational entertainment. Unfortunately, not every computing system is equipped with dedicated TRNG hardware, limiting the application space and security guarantees for such systems. To open the application space and enable security guarantees for the overwhelming majority of computing systems that do not necessarily have dedicated TRNG hardware (e.g., processing-in-memory systems), we develop QUAC-TRNG, a new high-throughput TRNG that can be fully implemented in commodity DRAM chips, which are key components in most modern systems.

QUAC-TRNG exploits the new observation that a carefully-engineered sequence of DRAM commands activates four consecutive DRAM rows in rapid succession. This QUadruple ACtivation (QUAC) causes the bitline sense amplifiers to non-deterministically converge to random values when we activate four rows that store conflicting data because the net deviation in bitline voltage fails to meet reliable sensing margins.

We experimentally demonstrate that QUAC reliably generates random values across 136 commodity DDR4 DRAM chips from one major DRAM manufacturer. We describe how to develop an effective TRNG (QUAC-TRNG) based on QUAC. We evaluate the quality of our TRNG using the commonly-used NIST statistical test suite for randomness and find that QUAC-TRNG successfully passes each test. Our experimental evaluations show that QUAC-TRNG reliably generates true random numbers with a throughput of 3.44 Gb/s (per DRAM channel), outperforming the state-of-the-art DRAM-based TRNG by 15.08× and 1.41× for basic and throughput-optimized versions, respectively. We show that QUAC-TRNG utilizes DRAM bandwidth better than the state-of-the-art, achieving up to 2.03× the throughput of a throughput-optimized baseline when scaling bus frequencies to 12 GT/s.

1. Introduction

True random numbers are used in a wide range of applications, including cryptography, scientific simulations, machine learning, and recreational entertainment [14, 16, 18, 37, 46, 61, 85, 97, 109, 112, 127, 130, 132, 146, 151, 160, 161, 166, 169, 170]. These applications often require a high-throughput true random number generator (TRNG) that is resilient to variations in operating conditions (e.g., temperature and voltage fluctuations) and is secure against malicious attacks [167].

Unfortunately, not all computing systems are provisioned with dedicated TRNG hardware, limiting their ability to run such applications effectively. In order to address this issue, many works have attempted to provide true random number generators purely using commodity hardware components that can be found in most systems today (e.g., DRAM [15, 81, 88, 126, 150] and SRAM [67, 68, 158]).

Using DRAM as the entropy source for generating true random numbers (i.e., DRAM-based TRNG) is a promising approach to providing a TRNG to a variety of computing systems ranging from high-performance servers, low-power edge devices, and systems that employ processing-in-memory [157] due to the widespread adoption of DRAM as main memory across these systems. However, prior proposals for DRAM-based TRNGs *(i)* have high latencies in generating random numbers because they rely on fundamentally slow processes (e.g., retention failures [63, 81, 149, 153], DRAM start-up values [47]) or *(ii)* generate random numbers at low throughput because they either use small portions of selected DRAM rows as an entropy source (e.g., t_{RCD} failure-based [88]) or use whole DRAM rows as an entropy source but fail to induce metastability in many sense amplifiers (e.g., t_{RP} failure-based [15]).

Our goal in this work is to develop a TRNG that uses commodity DRAM devices to generate random numbers with both high throughput and low latency. To achieve this, we leverage the novel observation that a carefully-engineered sequence of DRAM commands (described in Section 4) activates four DRAM rows in quick succession in commodity DRAM chips from one major DRAM manufacturer (SK Hynix), a process we refer to as *QUadruple ACtivation* (QUAC).

Our key idea is to leverage QUAC as a substrate for low-latency and high-throughput DRAM-based TRNGs. When activating rows that are initialized with conflicting data (e.g., data '0' in two rows and data '1' in the other two), bitline sense amplifiers non-deterministically converge to random values based on their individual circuit characteristics resulting from manufacturing process variation. Using QUAC operations to induce metastability in many DRAM sense amplifiers in parallel enables high-throughput and low-latency random number generation.

To this end, we develop QUAC-TRNG, a DRAM-based TRNG that repeatedly performs QUAC operations in DRAM and processes the results of these operations using a cryptographic hash function [50] to generate random numbers with high throughput. One QUAC-TRNG iteration consists of five key steps: QUAC-TRNG *(i)* identifies four consecutive DRAM rows, *(ii)* initializes the rows with conflicting data patterns (e.g., data '0' in two rows and data '1' in the other two), *(iii)* performs a QUAC operation on the rows by issuing a sequence of DRAM commands, *(iv)* reads the result of the operation from the sense amplifiers, and *(v)* performs the SHA-256 cryptographic hash function [50] to post-process the result and output random numbers. Our experimental evaluation using 136 real DDR4 DRAM chips from 17 real DDR4 modules (Section 6) shows

that QUAC-TRNG generates an average of 7664 bits of random data per iteration and each iteration takes 1940 ns.

Compared to previously-proposed DRAM-based TRNGs [15, 47, 74, 81, 88, 126, 150], QUAC-TRNG enables *(i)* lower latency because it only requires simultaneous activation of consecutive rows, which can be performed quickly using DRAM commands, and *(ii)* higher throughput because it uses QUAC operations to induce metastability in many sense amplifiers in parallel.

We evaluate QUAC-TRNG's quality by showing that random bitstreams generated using real DRAM chips pass the NIST statistical test suite [20] (Section 7.1). We then quantitatively evaluate QUAC-TRNG's performance against two state-of-the-art DRAM-based TRNG proposals [15, 88] (Section 7.4). For each prior proposal, we consider two configurations: *(i)* an unmodified *base* version as proposed in the original paper and *(ii)* an *enhanced* version that we believe represents a more fair comparison against our work. The enhanced versions incorporate optimizations to improve throughput and employ the SHA-256 hash function for post-processing. Our results show that QUAC-TRNG's throughput is 15.08× and 1.41× that of the best prior DRAM-based TRNG for the basic and enhanced configurations, respectively. We show that QUAC-TRNG scales quasi-linearly with available DRAM bandwidth, outperforming the *enhanced* configuration of the best prior DRAM TRNG by up to *2.03x* at future DRAM transfer rates. We also study and demonstrate how QUAC-TRNG can be integrated into a real system (Section 9) with minor performance, memory capacity, and CPU die area costs.

We make the following key contributions:

- We make the novel observation that a carefully engineered sequence of DRAM commands can activate four DRAM rows in quick succession. We refer to this operation as QUadruple ACtivation (QUAC). We show that QUAC operations can induce metastability in DRAM bitline sense amplifiers, which we exploit to generate true random numbers.
- We introduce QUAC-TRNG, a new high-throughput TRNG based on QUAC operations that is suitable for commodity DRAM chips. QUAC-TRNG combines the benefits of two components to generate high-quality true random numbers with high throughput: (i) massive parallelism in true random number generation available in DRAM sense amplifiers and (ii) randomness quality improvements provided by the SHA-256 hash function to generate random numbers at significantly higher throughput than previously-proposed DRAM-based TRNGs.
- We experimentally demonstrate that QUAC-TRNG is a high-quality TRNG by showing that the random bitstreams QUAC-TRNG generates pass *all* the standard NIST statistical test suite randomness tests [20].
- We show that QUAC-TRNG improves throughput over state-of-the-art DRAM-based TRNG proposals [15, 88], achieving 15.08× and 1.41× the throughput of basic and throughput-optimized baselines, respectively.
- We present a detailed experimental characterization study of the randomness provided by QUAC operations using 136 real DDR4 chips (from 17 DDR4 modules). We show that *(i)* QUAC-TRNG is suitable for implementation in commodity DRAM chips, and *(ii)* the randomness provided by QUAC operations remains stable over time.

2. Background

2.1. DRAM Structure and Organization

DRAM-based main memory is organized hierarchically. A processor is connected to one or many *memory channels*. Each channel has its own command, address, and data buses. Multiple *memory modules* can be plugged into a single channel. Each module contains several *DRAM chips*, which are grouped into *ranks*. Each rank contains multiple *banks* that are striped across the chips that form the rank but operate independently. Particular standards cluster multiple *banks* in *bank groups* [76, 77]. Data transfers between DRAM memory modules and processors occur at a *cache block* granularity.

DRAM Bank Organization. *A DRAM bank* is divided into multiple *subarrays* [33, 93, 140]. Each subarray comprises multiple wordline drivers and sense amplifiers (SAs), as shown in Figure 1-❶. Subarrays are further divided into *DRAM MATs*. Figure 1-❷ shows a DRAM MAT. DRAM MATs are separated from each other by *wordline (WL) drivers* that are activated to drive a DRAM wordline within the DRAM MAT. In a DRAM MAT, *DRAM cells* are organized into a two-dimensional structure over *bitlines* and *wordlines*. The set of cells over the same wordline forms a *DRAM row*, as depicted in Figure 1-❸.

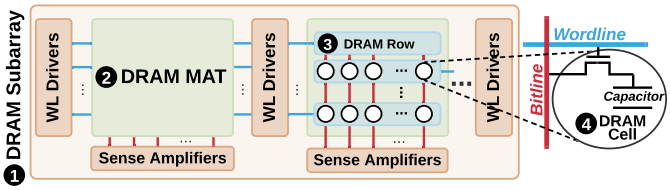

Figure 1: DRAM subarray, MAT, row and cell organization

Accessing DRAM. A DRAM cell (Figure 1-❹) stores data as a voltage level between the supply voltage (V_{DD}) and ground in its capacitor. Each cell is connected to a bitline via an *access transistor*. When all rows are closed, bitlines are precharged to the half of supply voltage ($V_{DD}/2$). Accessing a cell requires activating the corresponding row by issuing an (*ACT*) command. The activation process starts with enabling a wordline, which enables all access transistors in the row. As the access transistors are turned on, each cell shares its charge with the corresponding bitline, causing deviation on the bitline voltage either towards V_{DD} or ground. Each SA amplifies a bitline's voltage to either V_{DD} or 0 as the deviation in bitline voltage exceeds a threshold voltage (V_{th}). Read and write operations can be issued to SAs only after the row activation is completed. A precharge (*PRE*) command is used to close a row and set the bitline voltage to $V_{DD}/2$.

DRAM Timing Parameters. A memory controller must obey the DRAM timing parameters defined in standards set by JEDEC (e.g., DDR4 [76]) while scheduling DRAM commands. Figure 2 presents a timeline of DRAM commands on the command bus. Consecutive *ACT* and *PRE* commands on the command bus must be interleaved by at least t_{RAS} (i.e., ACT → PRE timing parameter) (❶). This is because a row needs to be active for at least as long as t_{RAS} to allow its cells to fully restore their charge. The time window between a *PRE* and an *ACT* command on the command bus must be at least t_{RP} (❷). This is required to settle the bitline voltage to $V_{DD}/2$ and to disable the activated wordline. Back-to-back *ACT* commands to different bank groups and to different DRAM banks (in the same bank group) must be interleaved with latencies of t_{RRD_S} (❸)

and t_{RRD_L} (❹), respectively. t_{RRD_S} and t_{RRD_L} are usually small (3.00, 4.90 ns in DDR4-2666 [76]). This enables overlapping the activation latency of DRAM rows in different banks or bank groups.

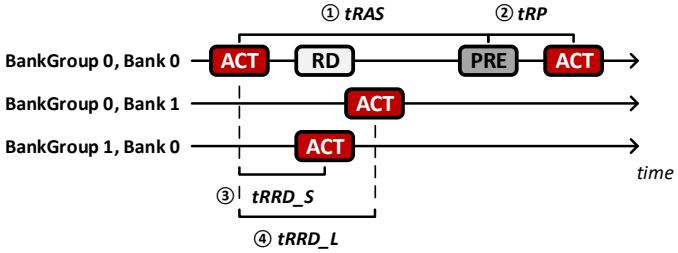

Figure 2: Timeline of key DDR4 commands.

DRAM manufacturers set large guardbands around DRAM timing parameters to guarantee correct operation [32, 35, 90, 101, 102]. A large body of work characterizes DRAM behavior under non-standard DRAM timing parameters to demonstrate that violating DRAM timing parameters allows improving DRAM access latency [30–32, 35, 45, 86, 90, 99, 101, 102], generating random numbers [15, 86, 88], implementing physical unclonable functions (PUFs) [15, 86, 87], and copying data and performing bitwise AND/OR in DRAM [53] on commodity DRAM devices.

2.2. True Random Number Generators

True random number generators (TRNGs) [96] harness entropy from random physical phenomena to generate random numbers. These entropy sources are often biased [96, 146], so practical TRNG designs often use post-processing methods to remove bias in their entropy sources, i.e., to strengthen the quality of the random numbers they produce (e.g., hashing [129] and other whitening algorithms [78, 161]). Post-processing can constrain TRNG throughput and latency, potentially requiring additional resources (e.g., output buffering) to offset its impact.

3. Motivation and Goal

High-quality random numbers are crucial to many technologies and applications [27, 37, 40, 42, 46, 61, 73, 85, 97, 108, 109, 112, 127, 132, 146, 161, 166, 169, 170]. In particular, random numbers are used widely in cryptographic communication protocols (e.g., key generation to initialize communication, signature and fingerprint generation to authenticate remote parties) to form secure channels between computing systems and networked devices. These protocols *require* an unpredictable, high-quality stream of *true random numbers* to remain secure against cryptographic attacks [37, 160] that aim to breach highly valuable, confidential user data. Some emerging key distribution protocols (e.g., quantum key distribution) provide even stronger security guarantees that make them resilient against a more diverse set of attacks [40, 108]. These protocols require TRNG throughputs on the order of several Gb/s [165]. Other than cryptography, high-throughput TRNGs are useful for other applications such as scientific simulations [27, 42, 73, 109], machine learning [112, 132, 166, 169], and gaming applications [146].

High-throughput TRNGs. Many prior works develop and demonstrate high-throughput TRNGs that use specialized hardware (e.g., optics [56, 104, 145, 156], ring oscillators [9, 29, 163, 167], chaotic circuits [43, 121]) to generate random numbers. Unfortunately, these proposals typically either *(i)* need to be integrated at design time, rendering them unsuitable for existing systems or *(ii)* are costly, limiting their potential for widespread adoption. To overcome these limitations and enable the aforementioned applications across computing systems ranging from high-performance servers to low-power edge devices, it is important to enable high-quality random number generation using existing commodity hardware.

DRAM-based TRNGs. DRAM is a promising substrate for true random number generation because DRAM chips are ubiquitous throughout contemporary computing platforms. DRAM-based TRNGs can be integrated into commodity systems at low cost with minimal effort [88], thereby enabling high-throughput random number generation across a broad spectrum of both *(i)* existing and *(ii)* future computing systems.

Synergy With PIM. Processing-in-memory (PIM) systems improve system performance and/or energy consumption by performing computations directly within a memory chip, thereby avoiding unnecessary data movement [25, 26, 57, 58, 60, 116, 118, 137, 139]. Prior works propose a broad range of PIM systems [5–8, 13, 22–24, 34, 38, 44, 48, 49, 54, 55, 58, 59, 65, 66, 71, 72, 89, 98, 100, 103, 107, 113, 115, 119, 120, 124, 133–135, 137–139, 142, 148, 164, 168] in the context of various workloads and memory devices. Enabling new PIM workloads (e.g., security applications) that rely on high-quality random numbers requires allowing the PIM system to perform TRNG operations directly within the memory to both (1) avoid inefficient off-chip communication to other possible TRNG sources, and (2) to enhance the overall security and privacy of PIM systems.

Shortcomings of Prior Work. Prior proposals for DRAM-based TRNGs either *(i)* have high latencies in generating random numbers because they rely on fundamentally slow processes (e.g., retention failures [63, 81, 149, 153], DRAM start-up values [47]) or *(ii)* generate random numbers at low throughput because they either use small portions of selected DRAM rows as entropy source (e.g., t_{RCD} failure-based [88]) or use whole DRAM rows as entropy source but fail to induce metastability on many sense amplifiers (e.g., t_{RP} failure-based [15]).

TRNGs based on DRAM start-up values [47] require a power cycle to generate random bits. This mechanism is impractical for a high-throughput TRNG because it both *(i)* incurs very high random number generation latency and *(ii)* precludes generating random bits in a streaming manner. TRNGs based on DRAM retention failures [81, 150] need to accumulate DRAM retention failures over long periods of time to harness enough entropy to generate random numbers. DRAM cells flip very infrequently due to retention failures as many DRAM cells retain data for hours [82, 106, 123, 128, 159]. The throughput of activation latency-based TRNGs [15, 88] is constrained by the amount of entropy they can harness from small portions of selected DRAM rows, a DRAM cache block. For example, D-RaNGe [88] can only use up to *4* out of the *64K* bits available for random number generation. Precharge latency-based TRNGs induce bit-flips on many DRAM cells in parallel on DRAM row granularity. However, the proportion of randomly-failing cells among all cells in a DRAM row following precharge latency failures is very low.[1]

We posit from our analysis of prior work that a high-throughput DRAM-based TRNG needs to *(i)* exploit DRAM failure mechanisms that are inherently fast and random (e.g.,

[1]Section 7.4 provides a rigorous analysis of prior DRAM-based TRNGs

timing failures), *(ii)* harness entropy from large portions of selected DRAM rows, and *(iii)* induce random behavior on a large proportion of sense amplifiers.

Our goal is to develop a new TRNG mechanism that uses commodity DRAM devices to robustly generate high-quality random numbers with higher throughput and low latency.

4. Quadruple Activation

We observe a new phenomenon, which we call **qu**adruple **ac**tivation (QUAC), in commodity DRAM modules. We find that by issuing a sequence of three standard DDR4 commands *(ACT → PRE → ACT)* with reduced timings (e.g., 2.5 ns), four consecutive DRAM rows in the same subarray are activated simultaneously. We identify the following two characteristics of QUAC. First, QUAC can simultaneously activate a set of four DRAM rows whose row addresses differ *only* in their two least significant bits (e.g., rows {0,1,2,3}). We refer to each such set of four DRAM rows as a *DRAM segment*. Second, we observe QUAC only when the two *ACT* commands target row addresses whose two least significant bits are inverted. In other words, the two *ACT* commands should target rows 0 and 3 (00 and 11 in base 2), or rows 1 and 2 (01 and 10 in base 2) within a DRAM segment.

To explain the potential mechanism behind QUAC, we examine the array architecture in state-of-the-art high-density DRAM chips. We hypothesize that the hierarchical design of wordlines allows QUAC to simultaneously activate four rows in a segment, and we present a hypothetical row decoder circuit that explains why the row addresses of the two *ACT* commands must have their two least significant bits set to inverted values.

4.1. Hierarchical Wordlines

High density and performance requirements have pushed DRAM designers to architect high-density, low-latency DRAM array architectures [114]. A commonly-used design pattern in architecting such DRAM arrays is to hierarchically organize DRAM wordlines to reduce latency and improve density [2, 36, 101, 155]. Figure 3 shows a DRAM MAT with the hierarchical wordline design.

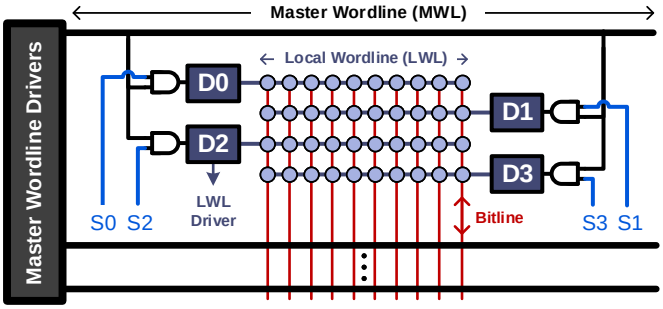

Figure 3: DRAM MAT with hierarchical wordlines

In the hierarchical wordline design, a DRAM row address is partitioned into two pieces. The higher-order bits of the row address are used to select and activate a master wordline (MWL). The MWL is connected to four local wordline (LWL) drivers (D0, D1, D2, D3 in Figure 3) that are used to activate four consecutive DRAM rows in a MAT. The least significant two bits of the row address are used to assert one of the four LWL select lines (S0 to S3) to enable an LWL driver and finally activate a DRAM row.

An activated MWL potentially drives four consecutive LWLs that form a segment. We hypothesize that the QUAC command sequence (ACT-PRE-ACT) asserts S0 to S3 approximately at the same time, resulting in simultaneous activation of four consecutive DRAM rows.

4.2. Hypothetical Row Decoder

We present a hypothetical row decoder circuit design that supports QUAC operations. The decoder design simultaneously activates four DRAM rows when the DRAM chip receives a series of ACT-PRE-ACT commands with violated timing parameters. Figure 4 illustrates our row decoder circuit, which operates on the least significant two bits of row addresses.

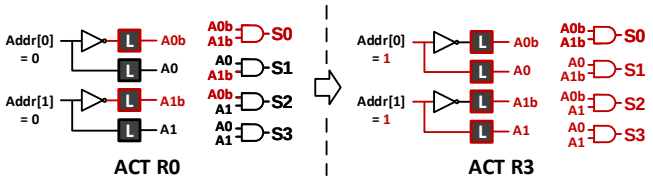

Figure 4: Hypothetical row decoder circuit that enables QUAC. The red and black colors represent asserted and de-asserted signals, respectively.

The first *ACT* command (Figure 4, left) targeting Row 0 (R0, Addr[1:0] = "00") sets the latches (L) that drive the signals *A0b* and *A1b*. These signals are combined through a logical-AND operation to form S0, which enables the LWL driver that activates R0. The following *PRE* command *cannot* deactivate R0 nor reset the latches that drive *A0b* and *A1b*, as the t_{RAS} parameter is violated. The second *ACT* command (Figure 4, right) targeting Row 3 (R3, Addr[1:0] = "11") sets the latches that drive the signals *A0* and *A1*. After the second *ACT* command, all four control signals (i.e., A0, A0b, A1, and A1b) are enabled since the previous *PRE* command fails to reset the latches. Together, these signals assert S1, S2, and S3, enabling the LWLs that activate R1, R2, and R3, respectively. Since R0 is still activated, this results in simultaneous activation of all four rows in a DRAM segment.

We confirm that QUAC activates four DRAM rows through an experiment with real DRAM chips. We first initialize a DRAM segment with a predefined data pattern. We then perform a QUAC operation on the DRAM segment to simultaneously activate four rows. Next, we write a new data pattern to the sense amplifiers while all four rows are active. Finally, we precharge the bank and individually read each row while obeying manufacturer-recommended DRAM timing parameters. We observe that all four rows are updated with the new data pattern we write. We observe valid QUAC operations in 136 DDR4 chips from one major DRAM manufacturer.

4.3. Future QUAC Interfaces

Even though current DDRX interfaces do not support QUAC, future DRAM chips can be built (and their interface accordingly specified) to take advantage of the same fundamental QUAC behavior to enable low-cost, high-throughput true random number generation (which we describe next in Section 5) as intended behavior.

5. QUAC-TRNG

QUAC-TRNG generates true random numbers at high-throughput by repeatedly performing QUAC.

5.1. Generating Random Output From QUAC

Figure 5 depicts how a QUAC operation generates a random output when the cells in rows R0 and R2 are initially *charged* (V_{DD}), and the cells in rows R1 and R3 are initially *discharged* (0) in a DRAM segment.

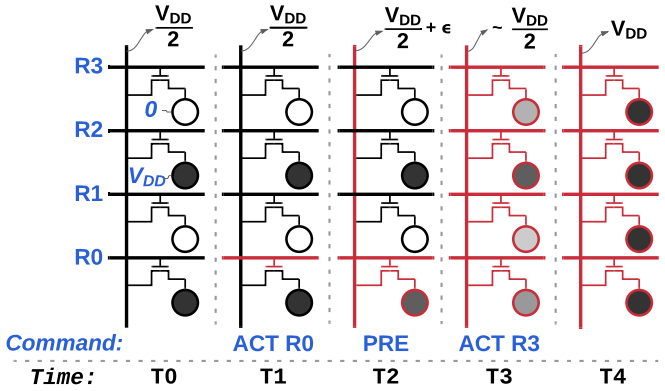

Figure 5: Timeline of changes in a DRAM bitline's state in a DRAM segment during a QUAC operation. Dashed vertical lines represent a state transition.

At T0, the bitline is precharged ($V_{DD}/2$). At T1, we enable wordline *R0* by quickly issuing an *ACT* command to R0. We interrupt the *ACT* command by issuing a *PRE* command at T2. Meanwhile, the cell on *R0* shares a portion of its charge with the bitline, reducing its voltage level ($< V_{DD}$). Before the *PRE* command closes the row and precharges the bitline, we issue another *ACT* command to *R3* at T3. The last *ACT* command interrupts the *PRE* command and enables wordlines R1, R2, and R3 simultaneously, in addition to the already enabled R0. Since QUAC opens four rows, all four cells on a DRAM bitline contribute to the bitline voltage. Following QUAC, at T4, the bitline ends up with a voltage level below reliable sensing margins. Thus, it is sampled as a random value by the sense amplifier; in Figure 5, the single depicted bitline is randomly sampled as V_{DD}.

To explain QUAC's true random number generation behavior, we hypothesize that QUAC produces random values in sense amplifiers by forcing each sense amplifier to attempt to amplify a differential voltage that is well below its reliable sensing margin (i.e., there is approximately no voltage difference between the sense amplifier's two terminals). Under these conditions, the sense amplifier fails to reliably develop and non-deterministically settles to either logical high or low based on thermal noise [21].[2] To achieve this, we initialize the four rows that will undergo QUAC with data patterns that ensure opposite charge values in DRAM cells along the same bitline. When charge sharing occurs amongst the four cells following a QUAC operation, the bitline remains close to the quiescent bitline voltage of $V_{DD}/2$. Therefore, any data pattern that programs the four cells with conflicting charge values will suffice.[3]

[2]We do *not* observe this behavior in *every* DRAM bitline within a DRAM segment. We attribute this to the effects of process variation across different components in the DRAM array, e.g., the capacitance of DRAM bitlines, the offset of differential sense amplifiers and the capacitance of DRAM cells.

[3]To analyze QUAC's data pattern dependency, we exhaustively test QUAC with 16 data patterns, as we describe in Section 6.1.

5.2. Mechanism

QUAC-TRNG leverages the random values in the sense amplifiers generated by QUAC operations as its source of entropy. QUAC-TRNG first performs a QUAC operation on a *high-entropy DRAM segment*[4] and generates random values in the sense amplifiers. QUAC-TRNG then uses the SHA-256 cryptographic hash [50] function to post-process the random values in the sense amplifiers to generate high-quality true random numbers.

Figure 6 depicts a DRAM subarray's logical organization when used for QUAC-TRNG and the three-step procedure of generating a 256-bit random number with QUAC-TRNG. QUAC-TRNG reserves six rows in a DRAM subarray to ensure that no other system component can access the reserved rows. Four of these rows form a segment that is used to perform QUAC. Two of them store *all-0s* and *all-1s* for initializing the segment with low latency.

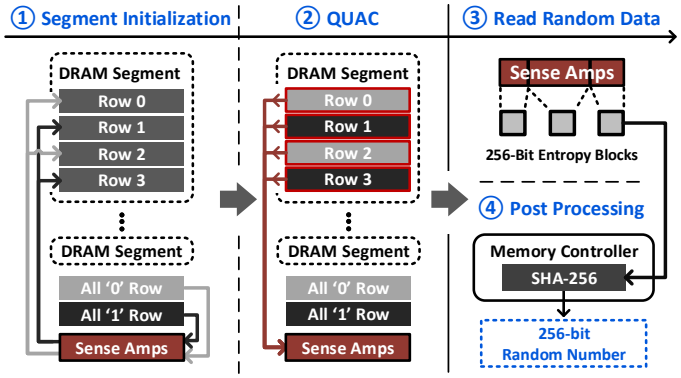

Figure 6: QUAC-TRNG mechanism.

To generate a 256-bit random number, QUAC-TRNG first selects a high-entropy DRAM segment and initializes the segment by performing four in-DRAM copy operations [53, 135] from the two reserved rows to each row in the segment ①. Second ②, it performs a QUAC operation on the segment to generate random data in the sense amplifiers. Third ③, the memory controller reads a block of bits from the sense amplifiers with a total amount of 256 bits of Shannon entropy (Section 6.1.1). Finally ④, the memory controller post-processes this block using the SHA-256 hash function to generate a 256-bit random number with improved quality of randomness.

6. Real DRAM Chip Characterization

6.1. Randomness in QUAC Operations

We experimentally study the entropy characteristics of QUAC operations across different data patterns and DRAM segments in real DRAM chips.

6.1.1. Experimental Methodology. To characterize the entropy in random values resulting from QUAC operations, we conduct experiments on 136 DRAM chips that come from 17 off-the-shelf DDR4 modules (see Appendix A, Table 3).

Infrastructure. We use a modified version of SoftMC [64] that enables precise control over DDR4 command timings, also used in [52, 91]. We test DDR4 modules (Figure 7-a) by issuing DDR4 command sequences that we send to the FPGA

[4]A high-entropy segment is a DRAM segment where QUAC operations generate *many* random values (i.e., with 1000s of bits of entropy) in the sense amplifiers, identified through a one-time characterization effort, as described in Section 6.1.2.

board (Figure 7-b) from the host machine through the PCIe interface (Figure 7-c). During our experiments, we control the temperature of DRAM chips on both sides of the module. To do so, we vertically connect the module to the FPGA board and heat the module as needed from both sides using rubber heaters (Figure 7-a). To control the heaters, we use a temperature controller (Figure 7-d) that performs a closed-loop PID control, which keeps the temperature constant at ± 0.1 °C of the desired temperature level (50 °C by default).

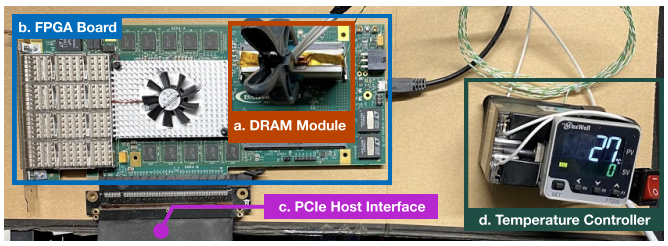

Figure 7: DDR4 SoftMC experimental setup.

Algorithm 1 describes the test procedure we use to extract true random numbers using QUAC operations. Algorithm 1 consists of three steps: step *(i)* initializes the DRAM segment with a data pattern (*Line 2*), step *(ii)* performs a QUAC operation on the DRAM segment (*Lines 3-7*), and step *(iii)* reads back the random values in the row buffer (*Lines 9-10*). To *simultaneously* enable all four rows in a segment, we activate the first and the fourth rows in the segment (e.g., Row_0 and Row_3) with two greatly violated timing parameters, t_{RAS}, and t_{RP}. First, we issue the *PRE* command (*Line 5*) earlier than the time delay (t_{RAS}) needed for charge restoration to complete. Second, we issue the second activation (*Line 7*) earlier than the time delay (t_{RP}), needed for bitlines to settle at $V_{dd}/2$. We obey the DRAM timing parameters while reading from every sense amplifier in the DRAM segment.[5]

Algorithm 1: Testing for QUAC's randomness

```
1  DRAM_QUAC_randomness_testing(data_pattern,
     DRAM_segment, DRAM_bank):
2    write data_pattern into all rows in DRAM_segment
3    activate(DRAM_segment : Row_0)
4    wait(2.5ns) // violate t_RAS
5    precharge(DRAM_bank)
6    wait(2.5ns) // violate t_RP
7    activate(DRAM_segment : Row_3)
8    wait(tRCD)
9    foreach SA in DRAM_segment:   // read each sense amplifier
10     record the value on the SA
```

Shannon Entropy. Shannon entropy [141] quantifies the amount of information present in a signal. We use Shannon entropy as a measure of the randomness in DRAM sense amplifiers following QUAC operations. We calculate a sense amplifier's Shannon entropy as in Equation 1, where $p(x_1)$ is the probability of observing a logical-0 value and $p(x_2)$ is the probability of observing a logical-1 value in the sense amplifier following QUAC operations. The total Shannon entropy (i.e., entropy) of a bitstream can be interpreted as the *effective* number of random bits within the bitstream.

$$H(x) = -\sum_{i=1}^{2} p(x_i) \log_2 p(x_i) \quad (1)$$

[5]We repeat Algorithm 1 for every DRAM segment in a DRAM bank in all DRAM modules.

6.1.2. Methodology to Measure Entropy in QUAC Operations. We measure the entropy of the random bitstreams generated in individual sense amplifiers by performing QUAC operations. We repeatedly perform QUAC (as shown in Algorithm 1) 1000 times and measure the entropy of each sense-amplifier by evaluating Equation 1 for the 1000-bit bitstream produced by each sense amplifier. We repeat this analysis on *8K* different DRAM segments (32K DRAM rows) using *16* different data patterns. We refer to the entropy of the bitstreams obtained from a sense amplifier connected to a bitline in a DRAM segment as that *bitline's entropy*.

6.1.3. Data Pattern Dependence. We analyze how the data patterns used in initializing DRAM segments affect the result of QUAC operations. We calculate the entropy for each cache block (i.e., 512 bitlines) in a DRAM module by aggregating the entropy of all bitlines in the cache block. We define two metrics *(i) average cache block entropy*, and *(ii) maximum cache block entropy*.[6] We calculate the *average cache block entropy* as the average entropy across all cache blocks in a DRAM module. The *maximum cache block entropy* is the entropy of the cache block with the highest entropy in a DRAM module. Figure 8 shows the average values of each of these metrics across all 17 modules we test. The error bars show the range (i.e., minimum and maximum) of the values across all modules. A larger entropy indicates more random behavior in DRAM sense amplifiers. We omit the data patterns that result in insufficient entropy in sense amplifiers following QUAC operations.

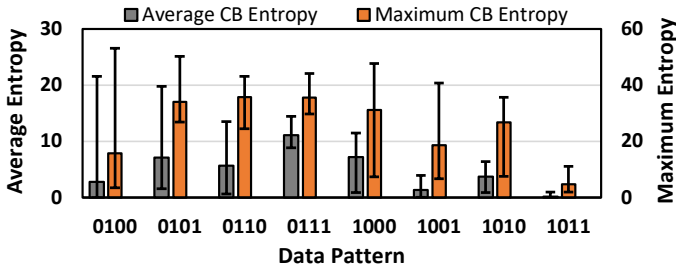

Figure 8: Average (grey bars, left Y-axis) and maximum (orange bars, right Y-axis) DRAM cache block entropies for different data patterns across 17 modules. The error bars show the range of the average and the maximum DRAM cache block entropy across all modules.

We make three observations from Figure 8. First, the average entropy varies across different data patterns. The average cache block entropy is the highest at 11.07 bits for the data pattern "0111" whereas it is the lowest at 0.17 bits for data pattern "1011". Second, we observe that the "0111" and "1000" data patterns lead to the highest entropy on average in all DRAM modules we test. This indicates that randomness increases when the first row QUAC activates (Row_0) is initialized with the inverted value of all other three rows (e.g., all-zeros in Row_0 and all-ones in the other three rows). This is because the cells in the first row have more time to share their charge with the bitlines as they are activated earlier than the other three rows. We hypothesize the bitline voltage is more likely to end up at a metastable level if all three later-activated rows simultaneously try to pull the bitline voltage in the opposite direction of the row that is activated first in QUAC operations. Third, we observe that cache block entropy in QUAC opera-

[6]The theoretical maximum entropy for a single cache block is 512 bits because each cache block is 512 bits (i.e., 64 bytes) wide.

tions can reach up to 53.0 bits with the "0100" data pattern. We hypothesize that this is a result of a combination of design-induced variation [101] and manufacturing process variation across DRAM segments. For example, variation in DRAM cell capacitance across DRAM segments may result in some DRAM segments to favor a certain data pattern (e.g., "0100"), i.e., performing QUAC on this segment keeps the bitline voltage below reliable sensing thresholds when the rows are initialized with that data pattern.

6.1.4. Spatial Distribution of Entropy. We study the spatial distribution of entropy in QUAC operations across segments in a DRAM bank. We calculate a segment's entropy as the sum of all bitline entropies in a DRAM segment. Figure 9 depicts how a segment's entropy (y-axis) varies across 8K DRAM segments in a DRAM bank (x-axis) across 136 DRAM chips, initialized with the data pattern that yields the largest average entropy ("0111"). There are three curves in Figure 9. The red curve shows the average segment entropy across all chips, with the error bars showing the maximum and minimum entropy values observed for any DRAM segment.[7] Black (dotted) and blue (dashed) curves provide representative samples of two main entropy variation trends (M1 and M2, respectively, depicting two selected DRAM modules) we observe across all chips.

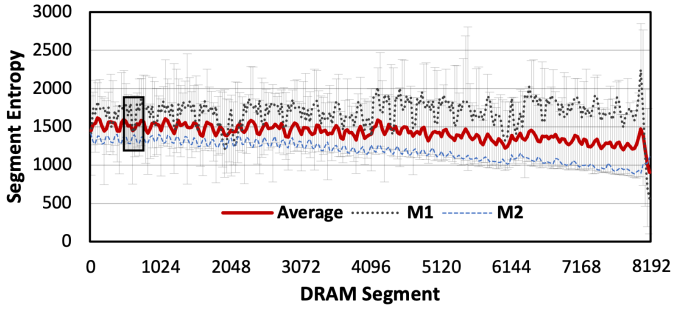

Figure 9: Average DRAM segment entropy across 17 modules (136 chips). The X-axis plots the DRAM segments and the Y-axis shows the segment entropy. We plot the segment entropy of two specific modules (M1 & M2) using black (dotted) and blue (dashed) lines.

We make three observations from Figure 9. First, the DRAM segment entropy behavior is different across modules. For example, the 640th segment (middle of the highlighted area on the figure) exhibits significantly lower entropy compared to nearby segments (i.e., leads to a local minimum) in module M1, but it exhibits a significantly higher entropy compared to its neighboring segments (i.e., leads to a local maximum) in module M2. Assuming the two modules' circuit designs are identical (since both modules are from the same manufacturer), we can potentially attribute this difference between modules to systematic process variation [111] and/or post-manufacturing row repair, where erroneous DRAM rows are remapped on a per-chip basis after manufacturing to improve yield [19, 41, 70, 75, 79, 80, 83, 84, 92, 101, 105, 122, 138, 144, 152]. Second, we observe that the overall segment entropy distribution follows a wave-like pattern. The segment entropy peaks and descends repeatedly as segment id (x-axis) increases (i.e., as DRAM row addresses increase) in the same DRAM bank. We hypothesize that this spatial pattern results from either the effects of systematic process variation or the structure of the local DRAM array. For example, a segment's entropy could be related to the segment's distance from the sense amplifiers. Third, a majority of modules experience a significant increase in segment entropy towards the 8000th segment, followed by a drop in segment entropy towards the end (i.e., 8192nd segment) of the DRAM bank. This could potentially be explained by systematic process variation or the micro-architectural characteristics of the DRAM bank. For example, the subarrays at the end of the bank might be differently sized than the rest of the subarrays, placing some segments further away from the sense amplifiers.

We calculate a *cache block's entropy (cache block entropy)* as the sum of the entropy of all bitlines in that cache block. We use the highest average-entropy data pattern ("0111") to initialize DRAM segments and find each cache block's entropy in the highest-entropy DRAM segment in each DRAM module. Figure 10 plots the average value of each cache block's entropy in the highest-entropy DRAM segment, and the error bars show the range (i.e., minimum and maximum) of the values across all 17 modules. We observe that the cache block entropy peaks around the middle of the DRAM segment and deteriorates towards the end of the DRAM segment. This indicates that the bitlines in the higher-numbered cache blocks are less random than the bitlines in the lower- or middle-numbered cache blocks.

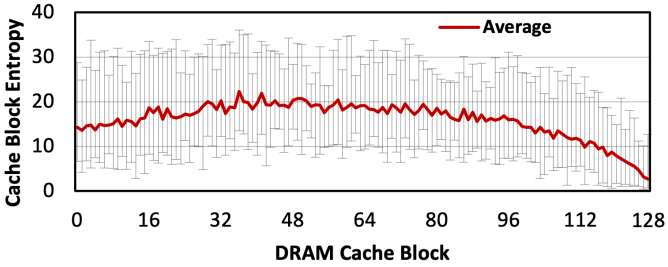

Figure 10: Average entropy of each cache block in the highest-entropy segment in all modules. The error bars show the range of the values across all modules.

We conclude from our analysis that the entropy provided by QUAC operations is distributed non-uniformly across DRAM segments and DRAM cache blocks. We hypothesize that the entropy distribution could be related to the micro-architectural characteristics of DRAM banks (e.g., distance of segments from the sense amplifiers), systematic variation in manufacturing processes [111], or post-manufacturing row-repair.

6.2. True Random Bitlines in QUAC Operations

We conduct a SoftMC experiment to demonstrate that QUAC operations, when performed repeatedly, generate random bitstreams in DRAM sense amplifiers. The SoftMC experiment works in three steps: *(i)* initializes the DRAM segment with a data pattern, *(ii)* performs a QUAC operation on the DRAM segment to generate random values in the sense amplifiers, *(iii)* reads out the DRAM segment. We collect one bit from each sense amplifier in the DRAM segment with each iteration of our experiment. We iterate one million times to collect 1 Mb bitstreams from every sense amplifier in the DRAM segment. Our entropy analysis shows that the values produced by QUAC operations on all sense amplifiers are biased towards a binary (logic-0 or logic-1) value (i.e., more likely to produce either one of the binary values). We use post-processing methods (Von Neumann Corrector [162] and SHA-256 [50]) to improve the quality of random bitstreams generated by QUAC operations.

[7] The theoretical maximum entropy of a single segment is 64K bits because there are 64K bitlines in each DRAM segment.

We apply the Von Neumann Corrector (VNC) [162] to all bitstreams to remove bias and improve the quality of the random number sequence. The VNC first splits all bits into groups of two bits. Then it applies one of the three transformations: *(i)* removes the group if both of the bits have the same value, *(ii)* removes the group and inserts a logic-1 if the first bit in the group is logic-0 and the second one is logic-1 (i.e., the generator transitions from logic-0 to logic-1), or *(iii)* removes the group and inserts a logic-0 otherwise. E.g., the bitstream "0010" after post-processing using the VNC becomes "0".

We use the NIST Statistical Test Suite (STS) [20] to validate the randomness of the output of our TRNG. NIST STS formulates several statistical tests to test a specific *null hypothesis*, H_0, which states that the number sequence under test is *random*. The suite outputs a *p-value* for all of the statistical tests that it runs on the random number sequence. We say that H_0 holds for a statistical test if it outputs a p-value greater than a chosen *level of significance* denoted as α. That is, if the p-value of a test is greater than α, then the number sequence is random according to that test. We choose α as 0.001 based on the suggested *level of significance* range ([0.01, 0.001]) in the NIST STS specification [20].

We collect bitstreams from every sense amplifier (64K in one DRAM segment) in a DRAM segment following QUAC operations. We test 8K DRAM segments in every DRAM module. We observe that 1 Mbit bitstreams collected from 22 sense amplifiers can pass all NIST STS tests.

Table 1 presents the average p-values for the NIST STS test results on two types of bitstreams that pass all 15 tests: *(i)* the output of the Von Neumann Corrector ("VNC") and *(ii)* the output of the post-processing step we describe in Section 5.2 ("SHA-256"). We conclude that QUAC generates number sequences that are indistinguishable from true random number sequences. We discuss the randomness of post-processed results (SHA-256 column) in Section 7.1.

Table 1: NIST STS Randomness Test Results

NIST STS Test	VNC* (p-value)	SHA-256 (p-value)
monobit	0.430	0.500
frequency_within_block	0.408	0.528
runs	0.335	0.558
longest_run_ones_in_a_block	0.564	0.533
binary_matrix_rank	0.554	0.548
dft	0.538	0.364
non_overlapping_template_matching	>0.999	0.488
overlapping_template_matching	0.513	0.410
maurers_universal	0.493	0.387
linear_complexity	0.483	0.559
serial	0.355	0.510
approximate_entropy	0.448	0.539
cumulative_sums	0.356	0.381
random_excursion	0.164	0.466
random_excursion_variant	0.116	0.510

*VNC: Von Neumann Corrector

7. QUAC-TRNG Evaluation

We evaluate QUAC-TRNG using real DRAM chip experiments and simulation studies to show that QUAC-TRNG *(i)* produces high-quality random bitstreams, and *(ii)* outperforms prior DRAM-based TRNG proposals.

7.1. QUAC-TRNG Output Quality

To demonstrate that QUAC-TRNG produces high-quality bitstreams of random values, we experimentally extract nine bitstreams from three DDR4 modules (24 DRAM chips).[8] Our results show that the bitstreams pass all of the NIST STS tests.

We extract a single bitstream using five steps: we *(i)* initialize the DRAM segment with the *highest-entropy* data pattern ("0111"), *(ii)* perform a QUAC operation on the DRAM segment, *(iii)* read out the DRAM segment, *(iv)* split the DRAM segment into blocks that each have 256 bits of entropy based on our characterization of cache block entropy in Section 6.1.2, and *(v)* input the 256-bit entropy blocks to the SHA-256 hash function to obtain 256-bit random numbers.

We partition 1 Gb bitstreams obtained from each highest-entropy DRAM segment into 1 Mb random number sequences and test 1024 number sequences per DRAM segment using NIST STS. We find that 99.28% of the sequences pass all NIST STS tests. This pass rate is larger than the acceptable rate[9] (98.84%) that NIST specifies [20].

Table 1, column "SHA-256" shows the average p-value for each test. We conclude that QUAC-TRNG generates high-quality uncorrelated, random bitstreams.

7.2. QUAC-TRNG Throughput

We analytically model QUAC-TRNG's throughput for a module in terms of *(i)* the number of input blocks with 256 bits of entropy in the highest-entropy segment (SIB: SHA Input Blocks) and *(ii)* the overall latency of one QUAC operation (L). QUAC-TRNG generates $256 \times SIB$ random bits per DRAM bank in L ns, resulting in a throughput of $(256 \times SIB)/(L \times 10^{-9})$ bits per second. SIB is calculated directly from the entropy of the highest-entropy segment as $\lfloor segment\_entropy/256 \rfloor$. We calculate L by tightly scheduling the DRAM commands required to (i) initialize four DRAM rows with data patterns, (ii) perform QUAC, and (iii) read random values from the sense amplifiers into the memory controller.

QUAC-TRNG's latency (L) is dominated by the time it takes to initialize four DRAM rows in a DRAM segment. We apply two optimizations to amortize the initialization overhead and increase the peak throughput of QUAC-TRNG. First, we concurrently execute QUAC operations across multiple banks by exploiting bank-level parallelism. In particular, for DDR4, we interleave across bank groups due to DDR4's short ACT-to-ACT ($tRRD\_S$) timing constraint. Second, we use *in-DRAM copy* operations to initialize DRAM segments at row granularity by adopting ComputeDRAM's [53] RowClone-based [135] in-DRAM copy procedure in our DDR4 modules. Using in-DRAM copy, we significantly reduce the DRAM segment initialization latency.

Figure 11 shows QUAC-TRNG's random number throughput under three configurations: *(i) One Bank*, where we use a single DRAM bank to generate random numbers, *(ii) BGP* (Bank Group Parallelism), where we use four banks from different bank groups and overlap DRAM command latencies to fully utilize the available DRAM bandwidth, and *(iii) RC* (RowClone) + *BGP*, where we initialize DRAM segments using in-DRAM copy to alleviate the overheads of segment initialization and use four banks from different bank groups. We plot the aver-

[8] We test a total of nine bitstreams, each sized 1 Gb, obtained from three DRAM modules to demonstrate that QUAC-TRNG can produce statistically uncorrelated streams of random numbers while maintaining a reasonable testing time.

[9] Based on the formula $(1-\alpha) \pm 3\sqrt{\alpha(1-\alpha)/k}$, where k is the sequence population (1024) and α is the significance level (0.005)

age, maximum, and minimum TRNG throughput QUAC-TRNG provides across all DRAM modules.

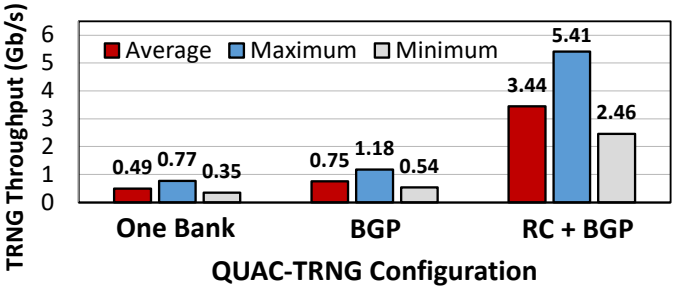

Figure 11: QUAC-TRNG's random number generation throughput (per DRAM channel) under three (One Bank, BGP, RC + BGP) configurations.

We observe that, on average, *One Bank* achieves 0.49 Gb/s, *BGP* achieves 0.75 Gb/s, and *RC + BGP* achieves 3.44 Gb/s random number throughput. The TRNG throughput of QUAC-TRNG varies across modules as the maximum segment entropy for each module varies. We conclude that QUAC-TRNG greatly benefits from in-DRAM copy to achieve high true random number generation throughput.

7.3. System Performance Study

To understand the maximum throughput that QUAC-TRNG can provide *without* reducing the *total off-chip memory bandwidth* available to concurrently-running applications, we run an experiment using memory traces from the SPEC2006 benchmark suite. We simulate a 3.2 GHz core with four DRAM channels of DDR4 memory using Ramulator [4, 94] to calculate the time each memory channel spends idle. We inject DDR4 commands that are issued in QUAC-TRNG iterations into these idle intervals. Figure 12 shows the random number generation throughput QUAC-TRNG provides while each SPEC2006 workload is running.[10] QUAC-TRNG generates random numbers at 10.2 Gb/s on average with a minimum (maximum) throughput of 3.22 Gb/s (14.3 Gb/s). We observe that by fully utilizing the idle intervals in the memory channels, QUAC-TRNG achieves on average, 74.13% of the empirical average throughput determined in Section 7.2 (i.e., 13.76 Gb/s for 4 DRAM channels).

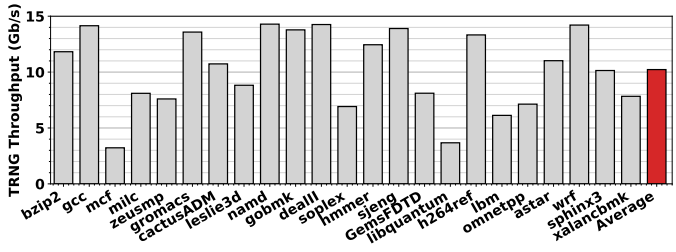

Figure 12: Available TRNG throughput during idle DRAM cycles while running SPEC2006 workloads.

7.4. Comparison With Prior Work

We quantitatively compare high-throughput ($> 100Mb/s$) DRAM-based TRNGs with QUAC-TRNG in this section. We scale each prior work's TRNG throughput and latency according to the simulated system with 4 DRAM channels described in Section 7.3. Table 2 presents a summary of our analysis, including the *low throughput* ($< 100Mb/s$) TRNGs, which we briefly discuss in Section 10.

[10]We use four banks from different bank groups in each channel.

Table 2: Summary of prior DRAM-TRNGs vs QUAC-TRNG

Proposal	Entropy Source	TRNG Throughput	256-bit TRNG Latency
QUAC-TRNG	Quadruple ACT	13.76 Gb/s	274 ns
Talukder+ [15]	Precharge Failure	0.68 - 6.13 Gb/s	249 ns - 201 ns
D-RaNGe [88]	Activation Failure	0.92 - 9.73 Gb/s	260 ns - 36 ns
D-PUF [150]	Retention Failure	0.20 Mb/s	40 s
DRNG [47]	DRAM Start-up	N/A	700 µs
Keller+ [81]	Retention Failure	0.025 Mb/s	40 s
Pyo+ [126]	DRAM Cmd Schedule	2.17 Mb/s	112.5 µs

We rigorously compare QUAC-TRNG to two state-of-the-art works that propose high-throughput DRAM-based TRNGs [15, 88]. We calculate both *(i)* the maximum random number generation throughput and *(ii)* the minimum latency for generating 256-bit random numbers for each of the high-throughput TRNGs. To do so, we tightly schedule the sequence of DDR4 commands each TRNG needs to issue.

7.4.1. D-RaNGe [88]. D-RaNGe generates random numbers in DRAM by leveraging failures due to reading a cache block before the row activation latency (t_{RCD}) is satisfied [88]. We analyze the throughput of D-RaNGe under two configurations: *(i) D-RaNGe-Basic*, where we evaluate D-RaNGe as proposed in [88], and *(ii) D-RaNGe-Enhanced*, where we characterize the entropy in t_{RCD} failures in real DDR4 devices to estimate the throughput of D-RaNGe combined with post-processing.

D-RaNGe-Basic. We calculate the throughput of D-RaNGe-Basic by carefully scheduling the required DDR4 commands to induce activation latency failures and read a cache block. For our analysis, we augment D-RaNGe-Basic to exploit bank-group-level parallelism in DDR4 devices. D-RaNGe observes that there are as many as four TRNG cells per cache block. We optimistically use the largest observed randomness (4 bits in a cache block) in calculating D-RaNGe-Basic's throughput. We do not use in-DRAM copy operations to further improve D-RaNGe-Basic's throughput because D-RaNGe does not benefit from the highly parallel DRAM row initialization provided by in-DRAM copy operations. D-RaNGe only needs to initialize one DRAM cache block, which can be done efficiently using DRAM write commands. Based on these observations and assumptions, we estimate D-RaNGe-Basic's maximum throughput as 916.9 Mb/s and minimum latency for generating 256-bit random numbers as 260 ns.

D-RaNGe-Enhanced. To calculate D-RaNGe-Enhanced's TRNG throughput, we evaluate 136 real DDR4 chips from 17 DDR4 modules using SoftMC and find the average cache block entropy provided by activation latency failures. For each DRAM cache block in a DRAM bank, one iteration of our SoftMC experiment: *(i)* initializes one DRAM row with an all-0s data pattern (found to induce the most random behavior [88]) and *(ii)* accesses the DRAM row with reduced t_{RCD}. We repeat this experiment *1000* times and calculate each cache block's entropy. We find the maximum cache block entropy for each DRAM module. We find the average of the maximum cache block entropy across all DRAM modules to calculate how many times D-RaNGe-Enhanced needs to access DRAM with reduced tRCD to gather sufficient entropy (256-bits). On average, D-RaNGe-Enhanced can harness 46.55 bits of entropy from a DRAM cache block (out of 512 bits of theoretical maximum entropy). We calculate that D-RaNGe-Enhanced needs to perform *6* reduced t_{RCD} accesses to generate a 256-bit random number. For a fair comparison, we apply the same

post-processing (SHA-256) to D-RaNGe's output as we do in QUAC-TRNG. D-RaNGe with post-processing achieves up to 9.73 Gb/s throughput. D-RaNGe-Enhanced's latency of generating a 256-bit random number is 36 ns, including the latency of the SHA-256 hash function. We conclude that post-processing using SHA-256 can significantly improve D-RaNGe's TRNG throughput as it enables utilizing a larger portion of the cache block for random number generation.

7.4.2. Talukder+ [15]. Talukder et al. propose generating random numbers in DRAM by leveraging bit failures due to activating a DRAM row *before* bitlines are precharged to $V_{DD}/2$ [15]. The authors use SHA-256 to post-process bitstreams that are read from DRAM. Talukder+'s mechanism (i) induces precharge latency failures on multiple DRAM rows, (ii) accumulates the random failures in DRAM cells, (iii) reads these DRAM cells, (iv) post-processes them using the SHA-256 hash function. We augment their algorithm to exploit bank-group-level parallelism in DDR4 devices. We use in-DRAM copy to initialize rows before inducing precharge latency failures. We analyze the throughput of Talukder+'s mechanism under two configurations: *(i) Talukder+-Basic*, where we estimate the throughput of the mechanism based on the authors' analysis on random cells, *(ii) Talukder+-Enhanced*, where we characterize the entropy provided by precharge latency failures in real DDR4 devices to estimate the throughput.

Talukder+-Basic. We calculate Talukder+-Basic's TRNG throughput using the results provided by the authors. The authors report that, on average, there are 130.6 random cells in a DRAM row. To accumulate 256-bits of entropy in input blocks of the SHA-256 hash function, Talukder+'s mechanism needs to read 3 DRAM rows. Based on this, the throughput of Talukder+'s mechanism is 681.2 Mb/s, and the latency of generating a 256-bit random number is 249 ns.

Talukder+-Enhanced. To calculate Talukder+-Enhanced's TRNG throughput, we evaluate 136 real DDR4 chips from 17 DDR4 modules using SoftMC and find the average DRAM row entropy (i.e., the sum of the entropy of all bitlines in a DRAM row) in precharge latency failures. We find the *maximum row entropy* for each DRAM module. We find the average of the maximum row entropy across all DRAM modules to calculate how many SHA-256 input blocks with sufficient entropy (256-bits) that Talukder+-Enhanced can extract from a high-entropy DRAM row. We find that, on average, Talukder+-Enhanced can harness 1023.64 bits of entropy from a high-entropy DRAM row (out of 64K bits of theoretical maximum entropy) following reduced t_{RP} accesses. On average, Talukder+-Enhanced can extract 3 SHA-256 input blocks with sufficient entropy from a DRAM row. We calculate Talukder+-Enhanced's throughput as 6.13 Gb/s. The latency of generating a 256-bit random number for the Talukder+-Enhanced is 201 ns.

Figure 13 plots the average throughput of Talukder+-Basic/Enhanced, D-RaNGe-Basic/Enhanced, and QUAC-TRNG. We project the throughput of the evaluated mechanisms to various DDR4 data transfer rates (MT/s).

We make two observations. First, D-RaNGe cannot make use of the additional DRAM bandwidth because D-RaNGe needs to frequently induce activation latency failures to sustain the high throughput of random numbers. Therefore, D-RaNGe's peak throughput is bound by DRAM access latency and does not scale with increasing DRAM external bandwidth.

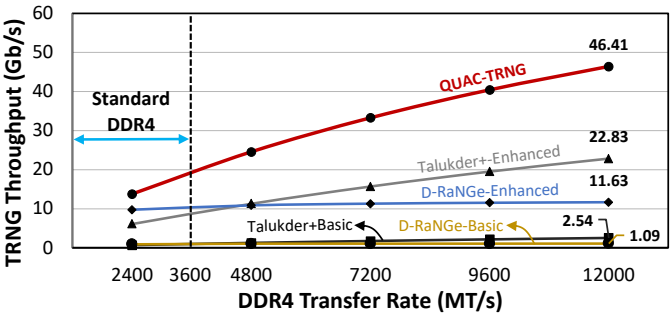

Figure 13: Throughput of DRAM-based TRNGs projected on DDR4 transfer rate. We plot transfer rates beyond the DDR4 standard [76].

Second, Talukder+ and QUAC-TRNG can scale with increasing DRAM transfer rate as they are bound by the DRAM bandwidth. QUAC-TRNG outperforms the basic (enhanced) versions of *Talukder+* and *D-RaNGe* by **20.20**× (2.24×) and **15.08**× (1.41×), respectively, at DDR4 2400 MT/s. At a future 12 GT/s transfer rate, QUAC-TRNG outperforms *enhanced* configurations of *Talukder+* and *D-RaNGe* in TRNG throughput by **2.03**× and **3.99**×, respectively.

Although QUAC-TRNG has a higher latency than Talukder+ and D-RaNGe, this latency for generating true random numbers can be hidden by accumulating random numbers in a buffer. Commodity systems that employ TRNGs already implement buffers to store random numbers [10]. QUAC-TRNG can fill this buffer at a significantly higher rate compared to state-of-the-art DRAM TRNGs because QUAC-TRNG achieves greater throughput.

8. Sensitivity Analysis

Temperature Dependence. We study the effects of temperature on the entropy of QUAC operations by recording bitline entropies at 50°C, 65°C, and 85°C on 40 real DRAM chips from 5 DRAM modules. We observe two trends: *Trend-1*, bitline entropy increases with temperature (24 chips), and *trend-2*, bitline entropy decreases with temperature (16 chips). We calculate the maximum and the average segment entropy (sum of all bitline entropies in that segment) independently for chips that follow *trend-1* and *trend-2*. Figure 14 plots the maximum and average segment entropy at 50°C, 65°C, and 85°C.

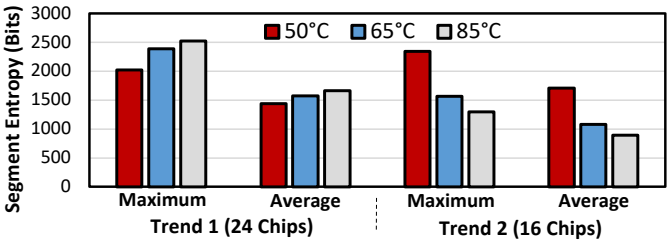

Figure 14: Maximum and average segment entropy at different temperatures.

We observe that the entropy in QUAC operations changes with temperature. The maximum (average) segment entropy is 2019.6 (1442.0), 2389.8 (1569.5) and 2520.1 (1659.6) at 50°C, 65°C and 85°C for DRAM chips that follow *trend-1*, respectively. The maximum (average) segment entropy is 2344.2 (1710.6), 1565.8 (1083.1) and 1293.5 (892.5) at 50°C, 65°C and 85°C for DRAM chips that follow *trend-2*, respectively. We conclude that a QUAC-TRNG implementation needs to account for changes

in temperature while generating true random numbers, as segment entropy changes with temperature.

To maintain the same amount of entropy (256-bits) in SHA-256 input blocks at different temperatures, the memory controller stores a list of *column address sets* for non-overlapping temperature ranges. This list is initialized by identifying high-entropy DRAM segments at different temperatures during a one-time offline characterization step. QUAC-TRNG accesses an element in the list depending on DRAM temperature (e.g., measured via temperature sensors [76]) and retrieves a set of column addresses, where each address points to a contiguous range of cache blocks in the DRAM segment with 256-bits of entropy. QUAC-TRNG uses these sets to split the data read from the high-entropy DRAM segment into SHA-256 input blocks. In this way, QUAC-TRNG ensures that SHA-256 input blocks always contain 256-bits of entropy at different temperatures.

Time Dependence. To understand whether the quality of the random numbers that QUAC-TRNG generates changes over time, we study the entropy generated by QUAC operations at the beginning and end of a 30-day period using 40 chips from five modules. The *average segment entropy* for the highest-entropy data pattern ("0111", Section 6.1.2) does not change significantly. The difference between the average entropy of 8K segments at the beginning and at the end of the testing period is on average (*maximum, minimum*) 2.4% (*5.2%, 0.9%*) across five modules (see Appendix A, Table 3). We conclude that the entropy generated by QUAC operations is not significantly affected by time elapsed on the order of a month, so the characterized segment entropy is valid for *at least* 30 days. Therefore, in the worst-case, QUAC-TRNG needs to re-characterize segment entropy only once a month.

9. System Integration

We discuss how QUAC-TRNG can be integrated into a real system. QUAC-TRNG generates random values by repeatedly (i) performing QUAC on the *highest-entropy* (Section 6.1.4) DRAM segments in four banks from four different DRAM bank groups, and (ii) post-processing the result of QUAC operations using the SHA-256 hash function.

Post Processing. QUAC-TRNG uses a cryptographic hash function to post-process random bitstreams produced by QUAC operations. We choose to evaluate QUAC-TRNG using SHA-256 as the post-processing function since SHA-256 is a secure cryptographic hash function that can be implemented efficiently in hardware at low area and latency costs [3, 17, 131]. This makes SHA-256 well-suited to implementation in the memory controller. We account for the costs of SHA-256 hardware in our evaluations based on values reported by recent work [17]: 65 clock cycle latency (at 5.15 GHz), 19.7 Gb/s throughput, and 0.001 mm^2 area at a 7 nm process technology node.

QUAC-TRNG User Application Interface. QUAC-TRNG generates random numbers using QUAC operations. To perform QUAC operations, the memory controller needs to issue an ACT $\rightarrow$ PRE $\rightarrow$ ACT command sequence with reduced t_{RAS} and t_{RP} timing parameters. Upon receiving a request for a random number, the memory controller checks if there is available DRAM bandwidth to perform QUAC operations and issues the command sequence with reduced timing parameters. This functionality can be implemented in a simple state machine in the memory controller's command scheduling logic. To eliminate delays when an application requests random numbers, the memory controller may periodically utilize available DRAM bandwidth to generate and store random numbers in a small buffer in the memory controller, as proposed in D-RaNGe [88]. In this way, an application's request for random numbers can be fulfilled immediately (up to the buffer size).

In order to use QUAC-TRNG in a real system, the designer needs to expose an interface to user applications. There are numerous possible ways to implement this interface, including memory- or PCIe- mapped configuration status registers, CPU co-processor and I/O instructions, and specialized extensions to the ISA. We leave it to the system designer to choose the best approach that meets the design goals for their system.

Memory Overhead. QUAC-TRNG allocates a small number of DRAM rows from one bank in four bank groups. We allocate one DRAM segment (four rows) to perform QUAC operations on and two DRAM rows to initialize the DRAM segment using in-DRAM copy operations. To fully utilize the DDR4 bandwidth, QUAC-TRNG simultaneously activates four segments in four bank groups (one bank in each bank group) and reads data from each bank in an interleaved manner. (Section 7.2). Thus, we allocate four segments (for QUAC) and 8 DRAM rows (for bulk initialization) across four banks in different bank groups. This amounts to 192 KB of total reserved space, which makes up only 0.002% of the capacity of an 8 GB DDR4 module.

Area Overhead. QUAC-TRNG stores *4 DRAM row addresses* to point to the starting row addresses of the highest-entropy DRAM segments and *8 DRAM row addresses* to point to the source operands for in-DRAM copy operations in four DRAM banks from four different bank groups. QUAC-TRNG also stores *11 DRAM column addresses*[11] to indicate the non-overlapping cache block ranges that contain 256-bits of entropy each. These cache block ranges change according to system temperature (Section 8). We assume there are as many as 10 distinct temperature ranges in calculating the area overhead. In total, to store the row and column addresses, QUAC-TRNG uses 1316 bits of storage. We model the required area for this storage using CACTI [1] and find that it is 0.0003 mm^2. With the SHA-256 core, QUAC-TRNG requires 0.0014 mm^2 area to implement in 7nm process technology, which is only 0.04% the chip area of a contemporary CPU designed at 7nm [11, 147].

10. Related Work

To our knowledge, this is the first work to *(i)* demonstrate that quadruple row activation (QUAC) in DRAM chips leads to random values by inducing metastability in DRAM sense amplifiers, *(ii)* exploit this phenomenon to design a new true random number generator, QUAC-TRNG. We have already extensively compared QUAC-TRNG to two state-of-the-art high-throughput TRNG designs [15, 88] in Section 7.4. In this section, we describe other related works.

10.1. Low-throughput DRAM-based TRNGs

Pyo et al. [126] (Table 2, Pyo+) generate random numbers using the unpredictability in DRAM command schedule as the entropy source. We calculate the peak theoretical throughput for Pyo+ as 2.17 Mb/s from the number of CPU cycles (45000) that it takes to obtain an 8-bit random number for the system

[11] To sustain the maximum 5.4 Gb/s TRNG throughput (Section 7.2) in modules where there are 11 SHA-256 input blocks with 256-bits of entropy in the highest-entropy segment.

we describe in Section 7.3. We find the latency of obtaining a 256-bit random number to be 112.5us.

Retention-based TRNGs [81, 150] *(i)* pause DRAM refresh to accumulate a sufficient amount of retention failures [82] that is used as the entropy source for true random number generation, *(ii)* read the portion of the DRAM array that contains the retention failures, and *(iii)* post-process the read data using hash functions (e.g., SHA-256) to finally obtain a random number.

D-PUF [150] (Table 2, D-PUF) partitions the DRAM into 4 MiB large regions and pauses DRAM refresh for 40 seconds for a region to accumulate a sufficient amount of retention failures in DRAM. D-PUF uses the SHA-256 hash function to post-process the data read from each region to generate a 256-bit random number. This incurs a minimum latency of 40 seconds to generate random numbers. We optimistically calculate the throughput of D-PUF assuming a four-channel system with 128 GiBs of DRAM. We also ignore the time it takes to read out 128 GiBs of data. When 1% of available DRAM (i.e., approximately *327* 4 MiB large regions) is reserved for retention failures, D-PUF's TRNG throughput is 0.002 Mb/s. Even when all DRAM (32K regions) is used, D-PUF can achieve only 0.20 Mb/s peak throughput.

Keller+ [81] (Table 2, Keller+) partitions the DRAM into 1 MiB large regions and pauses DRAM refresh for 320 seconds. Following an analysis similar to ours on D-PUF [150], we find Keller+'s TRNG latency for a 256-bit random number to be 320 seconds and its TRNG throughput to be only 0.025 Mb/s, assuming a four-channel system with 128 GiB DRAM fully utilized for true random number generation.

Startup value-based TRNGs [47] (Table 2, DRNG) use the startup values in DRAM cells that are accessed immediately after a DRAM device is powered up. These TRNGs *cannot* be used as a streaming true random number source as they require a DRAM power cycle to generate random numbers. We estimate the minimum latency of this category of TRNGs from the time it takes to execute a DDR4 power-up initialization sequence [143], which is 700 µs.

All these DRAM-based TRNGs provide very low random number generation throughput and incur high latency. Low-throughput TRNGs are unlikely to be useful in satisfying today's workloads with high throughput random number requirements (e.g., machine learning, cryptography, simulations [27, 37, 40, 42, 46, 61, 73, 85, 97, 108, 109, 112, 127, 132, 146, 161, 166, 169, 170]). QUAC-TRNG, on the other hand, can satisfy the high-throughput requirements of these workloads.

10.2. Non-DRAM-based TRNGs That Require Specialized Hardware

Many prior works design high-throughput TRNGs that are based on specialized hardware [9, 21, 28, 29, 43, 56, 68, 69, 95, 104, 110, 121, 125, 145, 154, 163, 167]. Unfortunately, it is costly to integrate these substrates into especially low-cost commodity systems as well as future processing-in-memory systems for true random number generation. Existing TRNGs in some commodity systems [10, 12, 78] both (i) consume die area to implement specialized circuitry (e.g., ring oscillators [117]) that harnesses entropy from physical phenomena and (ii) are limited in throughput. For example, the TRNG in a recent high-end AMD Zen3 processor can provide up to 3.18 Gb/s throughput per core, assuming a 4 GHz clock rate [51], which is only 23.11% of the throughput QUAC-TRNG can provide (on a four-channel DDR4-2400 system).

In general, choosing a TRNG is a design-time decision that requires balancing needs with costs. QUAC-TRNG provides high-throughput true random number generation without introducing dedicated hardware for TRNGs. Instead, QUAC-TRNG leverages widely-used commodity DRAM as an entropy source. Therefore, QUAC-TRNG offers a new design point that can enable new applications that were previously infeasible with alternative TRNGs, especially for systems where the costs of on-chip TRNGs may be prohibitive (e.g., heavily constrained embedded systems, processing-in-memory architectures). For example, QUAC-TRNG would enable processing-in-memory systems [62, 116, 137, 157] to execute security workloads as it enables true random number generation directly within a DRAM chip.

10.3. Multiple Row Activation In DRAM

Ambit [137] and ComputeDRAM [53]. Seshadri et al. [134, 136, 137, 140] introduce the idea of triple row activation in DRAM, showing that this operation leads to a bitwise majority function across the three activated rows. ComputeDRAM [53] shows that a similar behavior can be observed in real off-the-shelf DRAM chips by carefully reducing the timing parameters between consecutive DRAM commands. We build on these works and introduce quadruple activation (QUAC), which leads to a fundamentally different phenomenon on real off-the-shelf DRAM chips, i.e., simultaneous activation of four DRAM rows. We exploit this phenomenon to generate true random numbers at high-throughput and low-latency.

CROW [65] and MCR-DRAM [39] propose a DRAM-based substrate to simultaneously activate multiple DRAM rows with the same data content to reduce access latency. **RowClone [135]** enables *consecutive* activation of two DRAM rows to copy data in DRAM. These mechanisms *(i)* require changes to DRAM chips and *(ii)* do not generate random numbers.

11. Conclusion

We introduce QUAC-TRNG, a high-throughput and low-latency DRAM-based TRNG that can be implemented in commodity systems at low cost. The key idea of QUAC-TRNG is to induce metastability on many DRAM sense amplifiers in parallel by exploiting a phenomenon we observe, quadruple row activation (QUAC), which simultaneously activates four DRAM rows in real DRAM chips. Via a detailed characterization of 136 real DRAM chips, we show that QUAC-TRNG produces random bitstreams that pass all 15 NIST STS tests, and generates high-quality true random numbers at 3.44 Gb/s throughput. We compare QUAC-TRNG against prior work that we evaluate under two configurations, basic (as proposed) and enhanced (throughput-optimized). QUAC-TRNG outperforms the state-of-the-art DRAM-based TRNG in throughput by 15.08× and 1.41× for the basic and the enhanced configurations, respectively. QUAC-TRNG scales well with DRAM bandwidth and outperforms the enhanced version of the state-of-the-art by 2.03× at projected future DRAM transfer rates (12 GT/s). We conclude that QUAC-TRNG reliably generates true random numbers at high-throughput and low-latency in real DRAM chips.

Acknowledgements

We thank the anonymous reviewers of ISCA 2021 for feedback and the SAFARI group members for feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners: Google, Huawei, Intel, Microsoft, and VMware.

References

[1] "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," https://www.hpl.hp.com/research/cacti/.
[2] "DRAM Power Model," https://www.rambus.com/energy/.
[3] "Fast Hashing Cores," https://www.heliontech.com/fast_hash.htm.
[4] "Ramulator Source Code," https://github.com/CMU-SAFARI/ramulator.
[5] S. Aga *et al.*, "Compute Caches," in *HPCA*, 2017.
[6] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[7] J. Ahn *et al.*, "PIM-Enabled Instructions: a Low-overhead, Locality-aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[8] B. Akin *et al.*, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.
[9] T. Amaki *et al.*, "An Oscillator-based True Random Number Generator with Process and Temperature Tolerance," in *DAC*, 2015.
[10] AMD, "AMD Random Number Generator," https://www.amd.com/system/files/TechDocs/amd-random-number-generator.pdf.
[11] AMD, "AMD Zen2 Microarchitecture," https://en.wikichip.org/wiki/amd/microarchitectures/zen_2.
[12] ARM, "ARM True Random Number Generator (TRNG) Technical Reference Manual Revision r0p0," https://developer.arm.com/documentation/100976/0000/Introduction/Features.
[13] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.
[14] V. Bagini and M. Bucci, "A Design of Reliable True Random Number Generator for Cryptographic Applications," in *CHES*, 1999.
[15] B. M. S. Bahar Talukder *et al.*, "Exploiting DRAM Latency Variations for Generating True Random Numbers," in *ICCE*, 2019.
[16] M. Bakiri *et al.*, "Survey on Hardware Implementation of Random Number Generators on FPGA: Theory and Experimental Analyses," *CSR*, 2018.
[17] L. Baldanzi *et al.*, "Cryptographically Secure Pseudo-Random Number Generator IP-Core Based on SHA2 Algorithm," *Sensors*, 2020.
[18] M. Barangi *et al.*, "Straintronics-Based True Random Number Generator for High-Speed and Energy-Limited Applications," in *IEEE Trans. Magn*, 2016.
[19] A. Barenghi *et al.*, "Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks," in *IVSW*, 2018.
[20] L. Bassham *et al.*, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," Special Publication (NIST SP), 2010.
[21] M. Bhargava *et al.*, "Robust True Random Number Generator Using Hot-Carrier Injection Balanced Metastable Sense Amplifiers," in *HOST*, 2015.
[22] A. Boroumand *et al.*, "Mitigating Edge Machine Learning Inference Bottlenecks: An Empirical Study on Accelerating Google Edge Models," arXiv:2103.00768, 2021.
[23] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[24] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," in *CAL*, 2017.
[25] A. Boroumand *et al.*, "Polynesia: Enabling effective hybrid transactional/analytical databases with specialized hardware/software co-design," arXiv:2103.00798, 2021.
[26] A. Boroumand *et al.*, "CONDA: Efficient Cache Coherence Support for Near-Data Accelerators," in *ISCA*, 2019.
[27] R. Botha, "The Development of a Hardware Random Number Generator for Gamma-ray Astronomy," PhD Dissertation, North-West University, 2005.
[28] R. Brederlow *et al.*, "A Low-power True Random Number Generator using Random Telegraph Noise of Single Oxide-traps," in *ISSCC*, 2006.
[29] M. Bucci *et al.*, "A High-speed Oscillator-based Truly Random Number Source for Cryptographic Applications on a Smart Card IC," in *TC*, 2003.
[30] K. Chandrasekar *et al.*, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *DATE*, 2014.
[31] K. K. Chang, "Understanding and Improving Latency of DRAM-Based Memory Systems," PhD Dissertation, Carnegie Mellon University, 2017.
[32] K. K. Chang *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[33] K. K. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[34] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[35] K. K. Chang *et al.*, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
[36] N. Chatterjee *et al.*, "Architecting an Energy-Efficient DRAM System for GPUs," in *HPCA*, 2017.
[37] A. Cherkaoui *et al.*, "A Very High Speed True Random Number Generator with Entropy Assessment," in *CHES*, 2013.
[38] P. Chi *et al.*, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *ISCA*, 2016.
[39] J. Choi *et al.*, "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM," in *ISCA*, 2015.
[40] P. J. Clarke *et al.*, "Robust Gigahertz Fiber Quantum Key Distribution," *Applied Physics Letters*, 2011.
[41] L. Cojocar *et al.*, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *S&P*, 2020.
[42] P. Davis and P. Rabinowitz, "Some Monte Carlo Experiments in Computing Multiple Integrals," *Mathematical Tables and Other Aids to Computation*, 1956.
[43] M. Degaldo-Restituto *et al.*, "Nonlinear switched-current CMOS IC for random signal generation," *Electronics Letters*, 1993.
[44] F. Devaux, "The True Processing in Memory Accelerator," in *HCS*, 2019.
[45] Donghyuk Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," PhD Dissertation, Carnegie Mellon University, 2016.
[46] M. Drutarovsky and P. Galajda, "A Robust Chaos-based True Random Number Generator Embedded in Reconfigurable Switched-Capacitor Hardware," in *Radioelektronika*, 2007.
[47] C. Eckert *et al.*, "DRNG: DRAM-based Random Number Generation Using its Startup Value Behavior," in *MWSCAS*, 2017.
[48] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.
[49] I. Fernandez *et al.*, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," 2020.
[50] FIPS, PUB, "180-2: Secure hash standard (SHS)," *US Department of Commerce, National Institute of Standards and Technology (NIST)*, 2012.
[51] A. Fog, "Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs," https://www.agner.org/optimize/instruction_tables.pdf.
[52] P. Frigo *et al.*, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.
[53] F. Gao *et al.*, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *MICRO*, 2019.
[54] M. Gao *et al.*, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[55] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.
[56] T. Gehring *et al.*, "Ultra-Fast Real-Time Quantum Random Number Generator with Correlated Measurement Outcomes and Rigorous Security Certification," arXiv:1812.05377, 2020.
[57] S. Ghose *et al.*, "Processing-in-Memory: A Workload-Driven Perspective," *IBM JRD*, 2019.
[58] S. Ghose *et al.*, "Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions," *arXiv:1802.00320*, 2018.
[59] C. Giannoula *et al.*, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *HPCA*, 2021.
[60] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: Understanding a Modern Processing-in-Memory Architecture," arXiv:2105.03814, 2021.
[61] Z. Gutterman *et al.*, "Analysis of the Linux Random Number Generator," in *SP*, 2006.
[62] N. Hajinazar *et al.*, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.
[63] M. S. Hashemian *et al.*, "A Robust Authentication Methodology Using Physically Unclonable Functions in DRAM Arrays," in *DATE*, 2015.
[64] H. Hassan *et al.*, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
[65] H. Hassan *et al.*, "CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability," in *ISCA*, 2019.
[66] S. M. Hassan *et al.*, "Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore," in *MEMSYS*, 2015.
[67] D. E. Holcomb *et al.*, "Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags," in *RFID*, 2007.
[68] D. E. Holcomb *et al.*, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers," *ToC*, 2009.
[69] J. Holleman *et al.*, "A 3mu W CMOS True Random Number Generator with Adaptive Floating-Gate Offset Cancellation," *JSSC*, 2008.
[70] M. Horiguchi, "Redundancy Techniques for High-Density DRAMs," in *ISIS*, 1997.
[71] K. Hsieh *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[72] K. Hsieh *et al.*, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[73] T. E. Hull and A. R. Dobell, "Random Number Generators," *SIAM Review*, 1962.
[74] K. Humood *et al.*, "DTRNG: Low Cost and Robust True Random Number Generator Using DRAM Weak Write Scheme," in *ISCAS*, 2021.
[75] K. Itoh, *VLSI Memory Chip Design*. Springer, 2001.
[76] JEDEC, "DDR4," *JEDEC Standard JESD79–4*, 2012.
[77] JEDEC, "Graphics Double Data Rate (GDDR5) SGRAM Standard," 2016.
[78] B. Jun and P. Kocher, "The Intel Random Number Generator (White Paper)," *Cryptography Research Inc.*, 1999.
[79] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
[80] B. Keeth and R. Baker, *DRAM Circuit Design: A Tutorial*. Wiley, 2001.
[81] C. Keller *et al.*, "Dynamic Memory-based Physically Unclonable Function for the Generation of Unique Identifiers and True Random Numbers," in *ISCAS*, 2014.
[82] S. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.

[83] S. Khan *et al.*, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
[84] S. Khan *et al.*, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
[85] J. Kim *et al.*, "Nano-Intrinsic True Random Number Generation: A Device to Data Study," *IEEE TCAS*, 2019.
[86] J. S. Kim, "Improving DRAM Performance, Security, and Reliability by Understanding and Exploiting DRAM Timing Parameter Margins," PhD Dissertation, Carnegie Mellon University, 2020.
[87] J. S. Kim *et al.*, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices," in *HPCA*, 2018.
[88] J. S. Kim *et al.*, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.
[89] J. S. Kim *et al.*, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-memory Technologies," *BMC Genomics*, 2018.
[90] J. S. Kim *et al.*, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.
[91] J. S. Kim *et al.*, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020.
[92] Y. Kim *et al.*, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[93] Y. Kim *et al.*, "A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[94] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," in *CAL*, 2016.
[95] D. Kinniment and E. Chester, "Design of an On-chip Random Number Generator using Metastability," in *ESSCIRC*, 2002.
[96] Ç. K. Koç, "About Cryptographic Engineering," in *Cryptographic Engineering*, 2009.
[97] S. H. Kwok and E. Y. Lam, "FPGA-based High-speed True Random Number Generator for Cryptographic Applications," in *TENCON*, 2006.
[98] Y.-C. Kwon *et al.*, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.
[99] D. Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," PhD Dissertation, Carnegie Mellon University, 2016.
[100] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," in *TACO*, 2016.
[101] D. Lee *et al.*, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[102] D. Lee *et al.*, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
[103] S. Li *et al.*, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.
[104] Z. Limeng *et al.*, "640-Gbit/s Fast Physical Random Number Generation Using a Broadband Chaotic Semiconductor Laser," *Scientific Reports*, 2017.
[105] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[106] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[107] Z. Liu *et al.*, "Concurrent Data Structures for Near-Memory Computing," in *SPAA*, 2017.
[108] X. Lu *et al.*, "FPGA Based Digital Phase-coding Quantum Key Distribution System," *Science China Physics, Mechanics & Astronomy*, 2015.
[109] X. Ma *et al.*, "Quantum Random Number Generation," *Quantum Inf.*, 2016.
[110] S. K. Mathew *et al.*, "2.4 Gbps, 7 mW All-digital PVT-variation Tolerant True Random Number Generator for 45 nm CMOS High-performance Microprocessors," in *JSSC*, 2012.
[111] V. Mehrotra, "Modeling the Effects of Systematic Process Variation of Circuit Performance," PhD Dissertation, Massachusetts Institute of Technology, 2001.
[112] Y. Miché *et al.*, "Machine Learning Techniques based on Random Projections," in *ESANN*, 2010.
[113] A. Morad *et al.*, "GP-SIMD Processing-in-Memory," in *TACO*, 2015.
[114] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[115] O. Mutlu *et al.*, "Processing Data Where it Makes Sense: Enabling In-Memory Computation," *Microprocessors and Microsystems*, 2019.
[116] O. Mutlu *et al.*, "A Modern Primer on Processing in Memory," arXiv:2012.03112, 2020.
[117] L. Ning *et al.*, "Design and Validation of High Speed True Random Number Generators Based on Prime-length Ring Oscillators," *The Journal of China Universities of Posts and Telecommunications*, 2015.
[118] G. F. Oliveira *et al.*, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," arXiv:2105.03725, 2021.
[119] L. Orosa *et al.*, "Dataplant: Enhancing system security with low-cost in-dram value generation primitives," arXiv:1902.07344, 2019.
[120] L. Orosa *et al.*, "CODIC: A Low-cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations," in *ISCA*, 2021.
[121] F. Pareschi *et al.*, "A Fast Chaos-based True Random Number Generator for Cryptographic Applications," in *ESSCIRC*, 2006.
[122] M. Patel *et al.*, "Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics," in *MICRO*, 2020.
[123] M. Patel *et al.*, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
[124] A. Pattnaik *et al.*, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.
[125] C. S. Petrie and J. A. Connelly, "A Noise-based IC Random Number Generator for Applications in Cryptography," in *Trans. Circuits Syst. I*, 2000.
[126] C. Pyo *et al.*, "DRAM as Source of Randomness," in *IET*, 2009.
[127] Quintessence Labs, "Random Number Generators White Paper," 2015.
[128] M. K. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[129] R. Rivest, "The MD5 Message-Digest Algorithm," in *RFC*, 1992.
[130] A. Röck, "Pseudorandom Number Generators for Cryptographic Applications," Master's thesis, Paris-Lodron-Universität Salzburg, 2005.
[131] A. Satoh and T. Inoue, "ASIC Hardware Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS," in *ITCC*, 2005.
[132] W. F. Schmidt *et al.*, "Feedforward Neural Networks with Random Weights," in *ICPR*, 1992.
[133] V. Seshadri, "Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems," PhD Dissertation, Carnegie Mellon University, 2016.
[134] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE CAL*, 2015.
[135] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[136] V. Seshadri *et al.*, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," arXiv:1611.09988, 2016.
[137] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[138] V. Seshadri *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
[139] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers*, 2017.
[140] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," arXiv:1905.09822, 2020.
[141] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, 1948.
[142] G. Singh *et al.*, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *FPL*, 2020.
[143] SK Hynix, "DDR4 SDRAM Device Operation."
[144] R. T. Smith *et al.*, "Laser Programmable Redundancy and Yield Improvement in a 64K DRAM," *JSSC*, 1981.
[145] A. Stefanov *et al.*, "Optical Quantum Random Number Generator," in *J. Mod. Opt*, 2000.
[146] M. Stipčević and Ç. K. Koç, "True Random Number Generators," in *Open Problems in Mathematics and Computational Science*, 2014.
[147] D. Suggs *et al.*, "The AMD "Zen 2" Processor," *Hot Chips*, 2020.
[148] Z. Sura *et al.*, "Data Access Optimization in a Processing-in-Memory System," in *CF*, 2015.
[149] S. Sutar *et al.*, "D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication and Random Number Generation," in *TECS*, 2018.
[150] S. Sutar *et al.*, "D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication in Embedded Systems," in *CASES*, 2016.
[151] S. Tao and E. Dubrova, "TVL-TRNG: Sub-Microwatt True Random Number Generator Exploiting Metastability in Ternary Valued Latches," in *ISMVL*, 2017.
[152] A. Tatar *et al.*, "Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.
[153] F. Tehranipoor *et al.*, "Robust Hardware True Random Number Generators using DRAM Remanence Effects," in *HOST*, 2016.
[154] C. Tokunaga *et al.*, "True Random Number Generator with a Metastability-based Quality Control," in *JSSC*, 2008.
[155] A. N. Udipi *et al.*, "Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores," in *ISCA*, 2010.
[156] K. Ugajin *et al.*, "Real-time fast physical random number generator with a photonic integrated circuit," *Optics Express*, 2017.
[157] UPMEM, "Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM accelerator (White Paper)," 2018.
[158] V. van der Leest *et al.*, "Efficient Implementation of True Random Number Generator Based on SRAM PUFs," in *Cryptography and Security: From Theory to Applications*, 2012.
[159] R. K. Venkatesan *et al.*, "Retention-aware Placement in DRAM (RAPID): Software Methods for Quasi-non-volatile DRAM," in *HPCA*, 2006.
[160] P. Vincent *et al.*, "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," *JMLR*, 2010.
[161] V. von Kaenel and T. Takayanagi, "Dual True Random Number Generators for Cryptographic Applications Embedded on a 200 Million Device Dual CPU SOC," in *CICC*, 2007.
[162] J. von Neumann, "Various Techniques Used in Connection with Random Digits," in *Monte Carlo Method*, ser. NBS Applied Mathematics Series, 1951.
[163] X. Wang *et al.*, "10-Gbps True Random Number Generator Accomplished in ASIC," in *RT*, 2016.
[164] Y. Wang *et al.*, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.
[165] Y. Wang *et al.*, "Theory and Implementation of a Very High Throughput True Random Number Generator in Field Programmable Gate Array," *RSI*, 2016.
[166] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and Computing*, 1998.
[167] K. Yang *et al.*, "An All-digital Edge Racing True Random Number Generator Robust Against PVT Variations," in *JSSC*, 2016.
[168] D. Zhang *et al.*, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.
[169] L. Zhang and P. Suganthan, "A Survey of Randomized Algorithms for Training Neural Networks," *Information Sciences*, 2016.
[170] T. Zhang *et al.*, "High-speed True Random Number Generation Based on Paired Memristors for Security Electronics," *Nanotechnology*, 2017.

A. Appendix

Table 3: Sample population of 17 DDR4 modules

Module	*Module Identifier*	*Chip Identifier*	*Freq. (MT/s)*	*Organization*			*Segment Entropy*		
				Size (GB)	*Chips*	*Pins*	*Avg.*	*Max.*†	*Avg. (after 30 days)*
$M1$	*Unknown*	H5AN4G8NAFR-TFC	2133	4	8	x8	1688.1	2247.4	–
$M2$	*Unknown*	*Unknown*	2133	4	8	x8	1180.4	1406.1	–
$M3$	*Unknown*	H5AN4G8NAFR-TFC	2133	4	8	x8	1205.0	1858.3	1192.9
$M4$	76TT21NUS1R8-4G	H5AN4G8NAFR-TFC	2133	4	8	x8	1608.1	2406.5	1588.0
$M5$	*Unknown*	T4D5128HT-21	2133	4	8	x8	1618.2	2121.6	–
$M6$	TLRD44G2666HC18F-SBK	H5AN4G8NMFR-VKC	2666	4	8	x8	1211.5	1444.6	–
$M7$	TLRD44G2666HC18F-SBK	H5AN4G8NMFR-VKC	2666	4	8	x8	1177.7	1404.4	–
$M8$	TLRD44G2666HC18F-SBK	H5AN4G8NMFR-VKC	2666	4	8	x8	1332.9	1600.9	1407.0
$M9$	TLRD44G2666HC18F-SBK	H5AN4G8NMFR-VKC	2666	4	8	x8	1137.1	1370.9	–
$M10$	TLRD44G2666HC18F-SBK	H5AN4G8NMFR-VKC	2666	4	8	x8	1208.5	1473.2	1251.8
$M11$	TLRD44G2666HC18F-SBK	H5AN4G8NMFR-VKC	2666	4	8	x8	1176.0	1382.9	1165.1
$M12$	TLRD44G2666HC18F-SBK	H5AN4G8NMFR-VKC	2666	4	8	x8	1485.0	1740.6	–
$M13$	KSM32RD8/16HDR	H5AN4G8NAFA-UHC	2400	4	8	x8	1853.5	2849.6	–
$M14$	F4-2400C17S-8GNT	H5AN4G8NMFR-UHC	2400	8	8	x8	1369.3	1942.2	–
$M15$	F4-2400C17S-8GNT	H5AN4G8NMFR-UHC	3200	8	8	x8	1545.8	2147.2	–
$M16$	KSM32RD8/16HDR	H5AN8G8NDJR-XNC	3200	16	8	x8	1634.4	1944.6	–
$M17$	KSM32RD8/16HDR	H5AN8G8NDJR-XNC	3200	16	8	x8	1664.7	2016.6	–

†The maximum possible entropy in a DRAM segment is 64K (65,536) bits.