Shimmer: Implementing a Heterogeneous-Reliability DRAM Framework on a Commodity Server

Konstantinos Tovletoglou, Lev Mukhanov, Dimitrios S. Nikolopoulos, and Georgios Karakonstantis
Queen's University Belfast, United Kingdom {ktovletoglou01, l.mukhanov, d.nikolopoulos, g.karakonstantis}@qub.ac.uk

Abstract—In this paper, we present the implementation of a heterogeneous-reliability DRAM framework, *Shimmer*, on a commodity server with a fully fledged OS. *Shimmer* enables splitting of DRAM into multiple domains with varying reliability and allocation of data depending on their criticality. Compared to existing studies which use simulators, we consider practical restrictions stemming from the real hardware and investigate methods to overcome them. In particular, we reveal that the implementation of the heterogeneous-reliability memory framework requires disabling of the hardware memory interleaving, which results in a significant degradation of the system performance. To overcome the induced performance loss, we develop a software-based interleaving. We evaluate the performance, power and energy of the server using 35 benchmarks across three memory configurations: the baseline configuration; with disabled hardware memory interleaving and *Shimmer* with software-based memory interleaving. Our results show that *Shimmer* introduces a minor 6% performance overhead, while reducing the average DRAM power by 19.9% when memory operates under relaxed refresh rate and lowered memory supply voltage. As one of our main contributions we demonstrate that a heterogeneous-reliability framework based on Shimmer can be realized on a commodity server and save 9.1% of the total processor and memory energy.

Index Terms—Power efficiency, energy saving, reliability, DRAM, critical data, heterogeneous-reliability memory, memory interleaving.

1 Introduction

The advent of Cloud computing and the resultant rapid increase of data is driving the aggressive scaling of DRAM for meeting the needs of higher memory density and bandwidth. As a result of the high memory demand, projections forecast that the memory subsystem will soon be responsible for more than 40% of the overall power consumption within most multicore systems [1]. However, the DRAM scaling is mainly hampered by the adoption of frequent refresh cycles and conservative voltage margins of the *operating supply voltage* [2] (V_{DD}). Both of these parameters are selected based on the worst-case conditions to retain the stored data. Such an approach might guarantee error-free storage of data, but the incurred power and performance overheads raise doubts about its efficiency, especially as the spread in retention times increases due to worsening parametric variations [3].

Such a challenge has attracted the interest of many studies, the majority of which proposed the adoption of relaxed *refresh rate* (T_{REFI}) for the cells with a high retention time [3], [4]. However, recent studies [5] revealed that the retention time can dynamically change, thus limiting the effectiveness of the above schemes since the error-free properties of the 'strong' cells may not be guaranteed and threatening the operation of the system with errors. *Error detection and correction codes* (*ECC*) can be deployed to address any potential errors when the previous schemes fail [6]. However, ECC introduce considerable area and power overheads [6], which may negate any power gain resulted from relaxing the memory operating parameters.

Recent studies [4], [7] tried to limit such overheads by exploiting the *inherent error resilient properties* of various applications, allowing errors to occur. Even though an application can exhibit error resiliency for some data objects, there are data-structures, *critical data*, such as the OS data [4], where memory errors may result in an unacceptable output or even system crashes. Consequently, it is essential to ensure the reliable storage of the memory regions that store critical data, while the DRAM operating parameters, such as T_{REFI} and V_{DD}, for the remaining memory, that contains only error resilient objects, can be relaxed to save power. To enable DRAM regions with varying reliability, previous studies [7] introduced *Heterogeneous-Reliability Memory* (*HRM*) where the memory space is split into reliable regions and regions operating under relaxed parameters, that do not guarantee integrity of the data.

The potential of *HRM* may have been showcased on simulators [7], [8] or used on experimental setups for DRAM characterization neglecting the impact on the performance [9]. In fact, there are practical architectural challenges related to the implementation on a real system that have not been addressed in previous studies. Particularly, in modern DRAM subsystems, *Hardware Memory Interleaving* (*HardInterlv*) is typically being used for distributing consecutive memory accesses across multiple memory channels and thus improving performance. Such a mechanism hinders the implementation of a *HRM* framework as it does not allow splitting of memory into regions with varying reliability. Current solutions ignore this issue and any performance overhead introduced when *HardInterlv* is disabled. To the best of our knowledge, none of the existing studies have ever tried to implement a *HRM* on a real system with a fully-fledged OS, neither to measure the performance overhead considering these limitations nor to mitigate the incurred overhead.

Our contributions can be summarized as follows:

- We implement a cross-platform *HRM* framework, *Shimmer*, on a real commodity server with a fully-fledged OS. *Shimmer* enables the reliable allocation of critical data and the usage of variably-reliable memory for the rest of the data.
- We evaluate the performance overhead, that can reach as high as 128%, incurred by the naive implementation of *HRM* in which the hardware memory interleaving is disabled, using 35 benchmarks from *SPEC* and *NAS* benchmark suites.
- We propose a software-based interleaving scheme, which limits the performance overhead induced by disabling the *HardInterlv* down to 6%.
- We demonstrate for the first time that *HRM* enables 9.1% energy savings on a real system when the variably-reliable memory domain allocates data in DRAM operating under relaxed V_{DD} and T_{REFI}.

The rest of the paper is organized as follows. Section 2 describes the DRAM background and the challenges, while Section 3 presents our proposed framework. Section 4 analyses the evaluation results. Finally, conclusions are drawn in Section 5.

2 DRAM Background and Challenges

A main memory subsystem based on DRAM is organized into *memory channel units* (*MCUs*) supporting a number of DRAM modules. Each *Dual In-line Memory Module* (*DIMM*) consists of ranks, each of which houses a number of banks that are two-dimensional arrays of cells.

Manuscript submitted: 04-Oct-2018. Manuscript accepted: 24-Dec-2018. Final manuscript received: 03-Jan-2019.

To improve the memory performance, modern servers implement *HardInterlv* [10]. Interleaving allows the uniform distribution of consecutive accesses across different MCUs, maximizing the memory bandwidth. Figure 1a illustrates how physical memory addresses are distributed across the MCUs when *HardInterlv* is enabled. We see that the physical addresses of each allocation, e.g. `0x00000-0x00100` (colored in green), are spread uniformly across all MCUs. As a result, all the accesses to these addresses will be distributed in the same way, that allows the parallel exploitation of these channels.

Challenges. *HardInterlv* poses a challenge to the implementation of *HRM* since all data is distributed across MCUs when it is enabled. Thereby, critical data would be also distributed across all MCUs, even on the ones that are configured with relaxed memory parameters. Thus, the integrity of critical data may be compromised which may lead to system crashes. To address this challenge, each MCU should have a distinguishable address space, which implies that *HardInterlv* must be disabled.

However, the configuration where *HardInterlv* is disabled (*NonInterlv*) may induce a performance degradation for memory-bound applications. In fact, our evaluation of the *NonInterlv* configuration showed that such an overhead can reach up to 128%, as we will discuss later in the paper. Figure 1b illustrates how the physical addresses are distributed across MCUs for the *NonInterlv* configuration. Specifically, we see that all the addresses from `0x00000-0x00100` range (colored in green) are assigned to a single MCU. Thus, the memory bandwidth in this case will be limited by the maximum bandwidth of one MCU compared to four MCUs when *HardInterlv* is enabled.

3 Proposed HRM Framework

In this section, we demonstrate our approach on mitigating the performance overheads introduced by disabling *HardInterlv* in the naive *HRM* scheme and present its implementation.

3.1 Software-based Memory Interleaving

Operating systems utilize virtual addresses for data allocation, which are translated to physical addresses when the accesses are forwarded to the main memory. The MCU translates the requested addresses based on an interleaving function to the location in a specific DIMM. Note that after disabling *HardInterlv*, OS handles the allocation process and can use bandwidth from multiple MCUs by allocating pages across the MCUs and utilizing them in sequence. This way, we still can enable memory interleaving at the software level by mapping the continuous virtual address space to the segments of physical addresses corresponding to different MCUs. We call such a scheme *Software-based Memory Interleaving* (*SoftInterlv*). *SoftInterlv* allows on-the-fly selection of the interleaving scheme and the number of MCUs that will be interleaved for each allocation, in order to allow a better usage of the memory bandwidth. However, even though *SoftInterlv* can mitigate the performance overhead of disabling *HardInterlv*, there are other systems that may be affected and exhibit a performance degradation, such as memory prefetchers. Figure 1c presents an example where *HardInterlv* is disabled and consecutive virtual addresses, `0x00000-0x00100` (colored in green), are mapped to memory from the same MCU. Furthermore, in Figure 1c, we see that when we enable *SoftInterlv* for the addresses from `0x10000-0x10100` range (colored in red), the allocation is distributed across 3 MCUs, which allows increasing the memory bandwidth as the memory accesses are handled in parallel.

3.2 Implementation

Server Platform. To enable our study, we develop the *NonInterlv* configuration and *Shimmer* on a commodity server featuring the *AppliedMicro X-Gene 2* [11] *System-on-a-Chip* (*SoC*), which hardware specification is provided in Table 1. All mem-

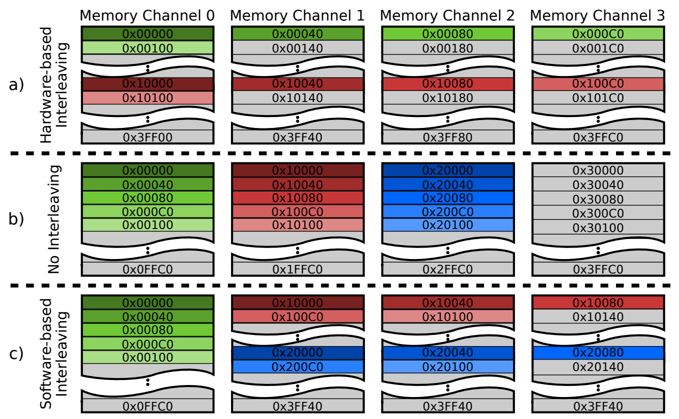

Fig. 1: Examples of allocation on a system *a)* in which *HardInterlv* is enabled; *b)* in which *HardInterlv* is disabled and *c)* in which a *SoftInterlv* policy is used for memory channels 1-3.

ory operations in X-Gene 2 are handled by 4 *MCUs*, which are divided in 2 groups of *Memory Controller Bridges* (*MCBs*). Each MCU can be populated with up to 2 DDR3 DIMMs running at 1866 MT/s. In our setup, we use 4 Micron DDR3 8GB DIMMs [12], one for each MCU.

The X-Gene 2 server features a dedicated processor to enable power management, monitoring of sensors and configuration of system parameters. Those parameters include MCU initialization and memory operating parameters (such as T_{REFI} and V_{DD}). During the MCU initialization, we can define the level of *HardInterlv*, either across all 4 MCUs or across the 2 MCUs of each MCB or completely disable it. The default configuration of X-Gene 2 is to have *HardInterlv* enabled across all MCUs.

Shimmer can be implemented in any server system that has the capability of modifying the memory operating parameters, such as T_{REFI} and V_{DD}. Furthermore, the server needs to have separate memory address space for each memory channel. For example, the Intel® Xeon processors [13] expose the parameter of T_{REFI} and the configuration of the memory interleaving mode between sockets and memory channels. While for tuning the V_{DD} of the memory, the motherboard needs to be equipped with adjustable voltage regulators.

On top of our system, we run CentOS 7 with the Linux kernel 4.11 for ARMv8, in which we apply our changes to implement *Shimmer*, as described in the following sections.

Linux Kernel Modifications. We extend the interface of *Non-Uniform Memory Access* (*NUMA*) domains in Linux OS to enable *HRM*. Systems with multiple sockets may have varying memory access time depending on which socket the DIMM accessed by a processor belongs to. In such systems, a NUMA domain is a group of DIMMs connected to the same socket. In our study, we use fake NUMA domains and bind each of them with a separate MCU, even though the memory access time is identical for each X-Gene 2 core.

We introduce 4 fake NUMA domains and assign each one

TABLE 1: Hardware platform specifications.

Description	Value
System	AppliedMicro X-Gene 2
Architecture	ARMv8
Core Frequency	2.4 GHz
Number of Cores	8
Last Level Cache	8 MB (shared)
# of Memory Controllers	4
Memory Size	8 GB/MCU, Total 32 GB
Memory Type	DDR3-1866, SECDED ECC
Memory Characteristics	2 rank/MCU, 8 banks/rank, 64 K rows/bank, 64 B cache line

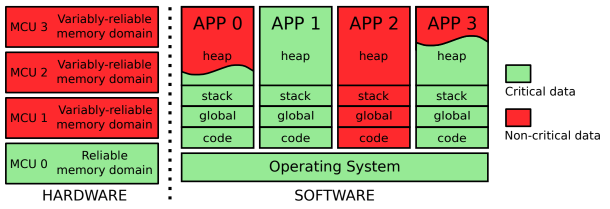
MCU 3 Variably-reliable memory domain
MCU 2 Variably-reliable memory domain
MCU 1 Variably-reliable memory domain
MCU 0 Reliable memory domain
HARDWARE
APP 0
APP 1
APP 2
APP 3
heap
stack
global
code
Operating System
SOFTWARE
Critical data
Non-critical data

Fig. 2: Hardware setup for the variably-reliable memory and possible allocations of applications in the memory.

to the memory address range of one MCU. By default, applications allocate data using the first NUMA domain that we use as the *Reliable Memory* (*RelMem*) domain. This domain is assigned to the MCU which uses the nominal memory operating parameters. The rest of the domains, *Variably-reliable Memory* (*VarMem*) domains, are assigned to MCUs in which DRAM reliability may vary as the memory operating parameters can be relaxed. Our framework can be adopted in most servers that have separate memory address space for each memory channel or that memory channel interleaving can be disabled. *Shimmer* can be also used in multi-socket systems since it does not restrict the use of the NUMA interface.

Allocation Interface of *HRM.* By default, the Linux kernel and every application, which uses the standard system functions, allocate data in *RelMem*. Furthermore, *Shimmer* allows the user to control the memory allocations for an application through the NUMA interface using the `numactl` command. Particularly, the parameter `--membind` can be passed to `numactl` to define the NUMA memory domain that will be used for data allocation. In Figure 2, *APP 1* and *APP 2* are examples of such an allocation, where all application data is assigned to either the *RelMem* or the *VarMem* domain.

We enable *SoftInterlv* of the memory in *Shimmer* by extending the `numactl` command with a new parameter `--interleave` across the NUMA memory domains. In such a way, we can utilize the memory channel parallelism and exploit the bandwidth of multiple MCUs, similarly to *HardInterlv*. We enable *SoftInterlv* using 3 interleaved MCUs, which can be assigned to the *VarMem* domain, so that the remaining MCU is used for the *RelMem* domain to store critical data.

To choose the reliability domain for a specific data allocation within an application, we extend the standard memory allocation interface, i.e. `malloc`, with a new interface, `numa_alloc_onnode`, that allocates data in a particular NUMA domain. The programmer can allocate non-critical data structures in the *VarMem* domain using this interface by modifying the source code of an application. Figure 2 illustrates such an allocation, where applications *APP 0* and *APP 3* allocate a part of the application's heap data in the *VarMem* domain.

The programmer must analyze the workloads to identify the criticality of the data structures and therefore select in which reliability domain to allocate the data. Recent research work [14] presented an automated technique to identify the resilience of data structures through analytical models, however, analysis of the data criticality is out of the scope of this paper.

Error Detection and Correction. The system reports all errors, either correctable or uncorrectable, detected by ECC and provides information about the memory location where the error occur: specifically, the MCU, rank, bank, row and column. Importantly, in our experiments with relaxed operating parameters set for the *VarMem* domain, all of the errors were corrected by the available ECC. This observation implies that the execution of benchmarks is not affected by memory errors in our experiments. Nonetheless, uncorrectable errors may occur in the *VarMem* domain, e.g. under high DRAM temperatures, and thus, each application allocating data in this domain should use a fault tolerance technique to handle such errors. We note that research of such techniques is also beyond the scope of our study.

4 Experimental Results

In this section, we evaluate the performance, power and energy of the server using a variety of benchmarks across three memory configurations: *i)* the default *HardInterlv* configuration; *ii)* the *NonInterlv* configuration and *iii) Shimmer*.

4.1 Workloads

We choose a set of workloads, ranging from micro-benchmarks that are typically used for evaluation of the memory bandwidth to real-world applications. These workloads stress the processor and memory in a different way, which allows us to investigate systematically the impact of the memory configurations on performance, power and energy.

We execute *LMbench* [15] to identify any deficiency in utilization of the memory bandwidth. From *SPEC CPU2006* [16], we perform experiments with 28 of the benchmarks using the *ref* data inputs, while running a single-threaded instance of the benchmark for each core in parallel. We also run 7 benchmarks from *NAS Parallel Benchmarks* [17] using the `C` input class and the OpenMP programming model. In all our experiments, we use 8 cores for parallel execution.

Note that we allocate all application data from both benchmark suites in the *VarMem* domain. For the *NonInterlv* system and *Shimmer*, the *VarMem* domain is configured to utilize 3 MCUs (1, 2, 3). Meanwhile, OS data is allocated in the *RelMem* domain, i.e. MCU 0, which operates under nominal parameters.

4.2 Performance Evaluation

The disabling of *HardInterlv* is accompanied with a performance degradation of the entire system, since the memory accesses are not distributed across all MCUs. To measure the degradation, we run *LMbench* and use 8 GB of memory, in order to restrict the allocation within a single MCU. The maximum memory bandwidth of the system with enabled *HardInterlv* is measured at 22.3 GB/s. Once we disable *HardInterlv* and enable *Shimmer*, the bandwidth reaches 16.4 GB/s. The 27% drop is explained by the fact that only three MCUs are used instead of four. While the bandwidth per MCU drops slightly from 5.57 GB/s when *HardInterlv* is enabled to 5.46 GB/s for *Shimmer*.

Figure 3 presents the performance overhead introduced by the *NonInterlv* system and *Shimmer* compared to the baseline configuration for the SPEC and NAS benchmarks. We calculate this overhead by measuring the execution time of each benchmark for the two configurations normalized to the execution times obtained for the default server configuration. Note that, in this figure, we use benchmark identification numbers instead of the whole *SPEC* benchmark names. We see that the *NonInterlv* configuration introduces an average performance overhead of 49.39%, which is expected as the memory bandwidth is not

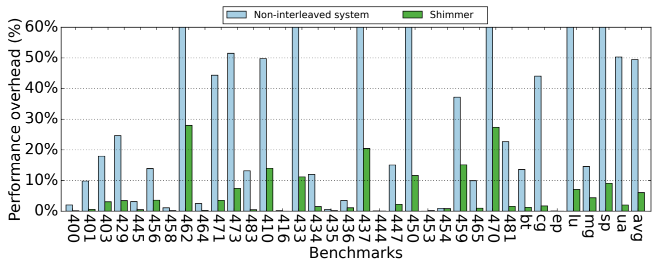

Fig. 3: Performance overhead manifested on the *NonInterlv* system and *Shimmer* compared to the *HardInterlv* system for the *SPEC* and *NAS* benchmarks.

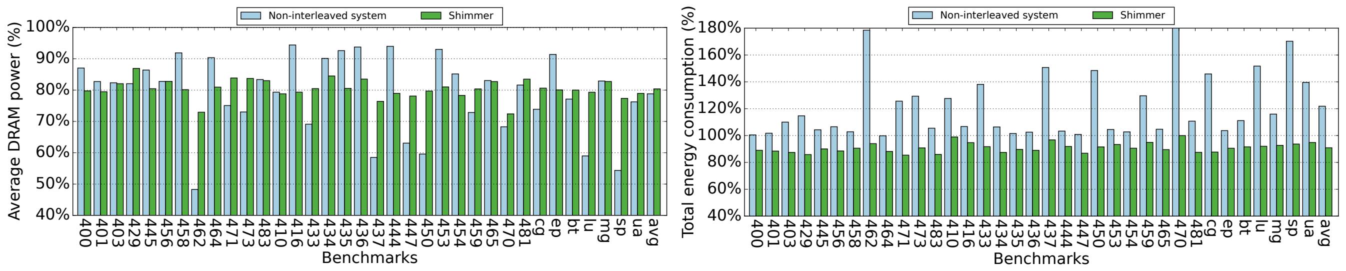

Fig. 4: *a)* The average DRAM power and *b)* the total energy consumption of the *NonInterlv* system and *Shimmer* normalized to the power and energy of the *HardInterlv* system across the *SPEC* and *NAS* benchmarks.

fully exploited. However, in the case of *Shimmer*, the average overhead decreases down to 6%. We also observe that the performance overhead varies across benchmarks, while the highest overhead is manifested for the `462.libquantum` and `470.lbm` benchmarks, 111% and 128% respectively for the *NonInterlv* configuration. Notably, the same benchmarks have also the highest overhead for *Shimmer*, which is about 28%.

4.3 Power and Energy Evaluation

To achieve the maximum power gain, we configure the 3 MCUs of the *VarMem* domain using the least reliable settings for V_{DD} and T_{REFI}. Note that our platform allows lowering of V_{DD} on the DIMMs associated with the second MCB (MCU 2 and 3) only, while we are able to relax T_{REFI} for MCU 1, 2 and 3. We set the V_{DD} to the minimum voltage specified by the DIMM's datasheet [12], i.e. we reduce V_{DD} from 1.5 V to 1.425 V (5% reduction), and increase T_{REFI} from the nominal 64 ms to 2.283 s (35$\times$ increase) that is the maximum allowed T_{REFI} in the X-Gene 2 platform. We measure power of DRAM and SoC using real voltage and current sensors that are integrated with voltage regulators on the X-Gene 2 motherboard.

Figure 4a shows the memory power consumption for the *NonInterlv* system and *Shimmer*, operating under relaxed DRAM parameters, normalized to the power measured for the baseline configuration under nominal V_{DD} and T_{REFI}. We see that the highest DRAM power reduction is achieved for the *NonInterlv* system, which is about 23%. While we can lower the average DRAM power consumption by 19.9% when *Shimmer* is enabled. Notably, in the case of the `462.libquantum` benchmark, which has the highest power consumption, *Shimmer* enables 27.1% DRAM power savings that is the maximum power reduction achieved among all benchmarks.

We also evaluate the total energy consumed by the server when running the *SPEC* and *NAS* benchmarks. We calculate the total energy consumption of the system (E_{total}) by integrating the DRAM and SoC power (P_{total}) over the duration (T) for each benchmark: $E_{total} = \int_0^T P_{total}(t)dt$. Figure 4b shows the total energy for the *NonInterlv* system and *Shimmer* normalized to the energy consumed by the server using the default configuration when DRAM operates under nominal V_{DD} and T_{REFI}. We see that in the case of the *NonInterlv* system, the average energy consumption grows by 21.8%, and no benchmark achieves any energy savings. By contrast, *Shimmer* achieves an average energy reduction of 9.1%. Moreover, only 3 benchmarks, i.e. `470.lbm`, `410.bwaves` and `444.namd`, have an energy reduction less than 5% using *Shimmer*.

5 Conclusion

In this paper, we present a cross-platform heterogeneous-reliability DRAM framework, *Shimmer*, implemented for the first time on a commercial server allowing us to obtain energy savings. We demonstrate that one of the main obstacles that hampers the implementation of heterogeneous-reliability memory on a real server is the hardware memory interleaving, which has been neglected by previous studies. We systematically investigate the performance overhead incurred when the interleaving is disabled, and we implement a software-based interleaving technique through the NUMA interface to reduce the incurred overhead significantly. Our results show that even though *Shimmer* introduces an average performance overhead of 6%, it enables the operation of parts of the DRAM under relaxed parameters and reduces the DRAM power by 19.9 % on average. Overall, *Shimmer* allows us to save 9.1% of the system energy when DRAM operates under relaxed parameters, while allocating data in the error-free domain of memory, and thus, offering protection of the critical data, including the OS data.

Acknowledgements

This work is funded by the H2020 Programme of the European Union under grant no. 688540 (the UniServer Project).

References

[1] B. Giridhar *et al.*, "Exploring dram organizations for energy-efficient and resilient exascale memories," in *Proc. Int. Conf. on High Perform. Comp., Netw., Storage and Analysis*, 2013, pp. 1–12.
[2] K. K. Chang *et al.*, "Understanding reduced-voltage operation in modern dram devices: Experimental characterization, analysis, and mechanisms," in *Proc. ACM SIGMETRICS*, 2017, pp. 52–52.
[3] J. Liu *et al.*, "Raidr: Retention-aware intelligent dram refresh," in *Proc. Int. Symposium on Computer Architecture*, 2012, pp. 1–12.
[4] A. Raha *et al.*, "Quality-aware data allocation in approximate dram," in *Proc. Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2015, pp. 89–98.
[5] M. K. Qureshi *et al.*, "Avatar: A variable-retention-time (vrt) aware refresh for dram systems," in *Proc. Int. Conf. on Dependable Sys. and Netw.*, 2015, pp. 427–437.
[6] P. J. Nair *et al.*, "Archshield: Architectural framework for assisting dram scaling by tolerating high error rates," in *Proc. Int. Symposium on Computer Architecture*, 2013, pp. 72–83.
[7] S. Liu *et al.*, "Flikker: Saving dram refresh-power through critical data partitioning," *SIGARCH Comp. Archit. News*, p. 213, 2011.
[8] M. Jung *et al.*, "Omitting refresh: A case study for commodity and wide i/o drams," in *Proc. Int. Symp. on Mem. Sys.*, 2015, pp. 85–91.
[9] C. Chalios *et al.*, "Dare: Data-access aware refresh via spatial-temporal application resilience on commodity servers," *Int. Journal of High Perf. Comp. Applic.*, vol. 32, no. 1, pp. 74–88, 2018.
[10] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proc. ACM/IEEE Int. Symp. on Microarchitecture*, 2000, pp. 32–41.
[11] G. Singh *et al.*, "AppliedMicro X-Gene 2," in *IEEE Hot Chips 26 Symp.*, 2014, pp. 1–24.
[12] Micron Technology, "MT18JSF1G72AZ-1G9 - 8GB," 2015.
[13] Intel, "Intel xeon processor e5-1600/2400/2600/4600 (e5-product family) product families datasheet, volume two," *Intel*, 2012.
[14] V. Vassiliadis *et al.*, "Towards automatic significance analysis for approximate computing," in *Proc. Int. Symposium on Code Generation and Optimization*, 2016, pp. 182–193.
[15] L. McVoy *et al.*, "Lmbench: Portable tools for performance analysis," in *Proc. USENIX Annual Technical Conf.*, 1996, pp. 279–294.
[16] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, 2006.
[17] D. H. Bailey *et al.*, "Nas parallel benchmark results," in *Proc. ACM/IEEE Conf. on Supercomputing*, 1992, pp. 386–393.