DOI: 10.14529/jsfi140302

Research Problems and Opportunities in Memory Systems

Onur Mutlu[1]***, Lavanya Subramanian***[1]

The memory system is a fundamental performance and energy bottleneck in almost all computing systems. Recent system design, application, and technology trends that require more capacity, bandwidth, efficiency, and predictability out of the memory system make it an even more important system bottleneck. At the same time, DRAM technology is experiencing difficult *technology scaling* challenges that make the maintenance and enhancement of its capacity, energy-efficiency, and reliability significantly more costly with conventional techniques.

In this article, after describing the demands and challenges faced by the memory system, we examine some promising research and design directions to overcome challenges posed by memory scaling. Specifically, we describe three major *new* research challenges and solution directions: 1) enabling new DRAM architectures, functions, interfaces, and better integration of the DRAM and the rest of the system (an approach we call *system-DRAM co-design*), 2) designing a memory system that employs emerging non-volatile memory technologies and takes advantage of multiple different technologies (i.e., *hybrid memory systems*), 3) providing predictable performance and QoS to applications sharing the memory system (i.e., *QoS-aware memory systems*). We also briefly describe our ongoing related work in combating scaling challenges of NAND flash memory.

Keywords: memory systems, scaling, DRAM, flash, non-volatile memory, QoS, reliability.

Introduction

Main memory is a critical component of all computing systems, employed in server, embedded, desktop, mobile and sensor environments. Memory capacity, energy, cost, performance, and management algorithms must scale as we scale the size of the computing system in order to maintain performance growth and enable new applications. Unfortunately, such scaling has become difficult because recent trends in systems, applications, and technology greatly exacerbate the memory system bottleneck.

1. Memory System Trends

In particular, on the *systems/architecture front*, energy and power consumption have become key design limiters as the memory system continues to be responsible for a significant fraction of overall system energy/power [112]. More and increasingly heterogeneous processing cores and agents/clients are sharing the memory system [11, 36, 39, 60, 78, 79, 178, 181], leading to increasing demand for memory capacity and bandwidth along with a relatively new demand for predictable performance and quality of service (QoS) from the memory system [129, 137, 176].

On the *applications front*, important applications are usually very data intensive and are becoming increasingly so [17], requiring both real-time and offline manipulation of great amounts of data. For example, next-generation genome sequencing technologies produce massive amounts of sequence data that overwhelms memory storage and bandwidth requirements of today's high-end desktop and laptop systems [9, 111, 186, 196, 197] yet researchers have the goal of enabling low-cost personalized medicine, which requires even larger amounts of data and their effective analyses. Creation of new *killer applications* and usage models for computers likely depends on how well the memory system can support the efficient storage and manipulation of data in such

[1]Carnegie Mellon University

data-intensive applications. In addition, there is an increasing trend towards consolidation of applications on a chip to improve efficiency, which leads to the sharing of the memory system across many heterogeneous applications with diverse performance requirements, exacerbating the aforementioned need for predictable performance guarantees from the memory system [176, 182].

On the *technology front*, two major trends profoundly affect memory systems. First, there is increasing difficulty scaling the well-established charge-based memory technologies, such as DRAM [4, 10, 70, 90, 97, 102, 124] and flash memory [20, 21, 24, 98, 123], to smaller technology nodes. Such scaling has enabled memory systems with reasonable capacity and efficiency; lack of it will make it difficult to achieve high capacity and efficiency at low cost. Challenges with DRAM scaling were recently highlighted by a paper written by Samsung and Intel [83]. Second, some emerging resistive memory technologies, such as phase change memory (PCM) [102, 103, 159, 163, 192], spin-transfer torque magnetic memory (STT-MRAM) [31, 100] or resistive RAM (RRAM) [193] appear more scalable, have latency and bandwidth characteristics much closer to DRAM than flash memory and hard disks, and are non-volatile with little idle power consumption. Such emerging technologies can enable new opportunities in system design, including, for example, the unification of memory and storage subsystems [127]. They have the potential to be employed as part of main memory, alongside or in place of less scalable and leaky DRAM, but they also have various shortcomings depending on the technology (e.g., some have cell endurance problems, some have very high write latency/power, some have low density) that need to be overcome or tolerated.

2. Memory System Requirements

System architects and users have always wanted more from the memory system: high performance (ideally, zero latency and infinite bandwidth), infinite capacity, all at zero cost! The aforementioned trends do not only exacerbate and morph the above requirements, but also add some new requirements. We classify the requirements from the memory system into two categories: *exacerbated traditional requirements* and *(relatively) new requirements*.

The traditional requirements of performance, capacity, and cost are greatly exacerbated today due to increased pressure on the memory system, consolidation of multiple applications/agents sharing the memory system, and difficulties in DRAM technology and density scaling. In terms of *performance*, two aspects have changed. First, today's systems and applications not only require low latency and high bandwidth (as traditional memory systems have been optimized for), but they also require new techniques to manage and control memory interference between different cores, agents, and applications that share the memory system [40, 129, 137, 176, 182] in order to provide high system performance as well as predictable performance (or quality of service) to different applications [176]. Second, there is a need for increased memory bandwidth for many applications as the placement of more cores and agents on chip make the memory pin bandwidth an increasingly precious resource that determines system performance [71], especially for memory-bandwidth-intensive workloads, such as GPGPUs [80, 81, 146], heterogeneous systems [11], and consolidated workloads [73, 74, 137]. In terms of *capacity*, the need for memory capacity is greatly increasing due to the placement of multiple data-intensive applications on the same chip and continued increase in the data sets of important applications. One recent work showed that given that the core count is increasing at a faster rate than DRAM capacity, the expected memory capacity per core is to drop by 30% every two years [113], an alarming trend since much of today's software innovations and

features rely on increased memory capacity. In terms of *cost*, increasing difficulty in DRAM technology scaling poses a difficult challenge to building higher density (and, as a result, lower cost) main memory systems. Similarly, cost-effective options for providing high reliability and increasing memory bandwidth are needed to scale the systems proportionately with the reliability and data throughput needs of today's data-intensive applications. Hence, the three traditional requirements of performance, capacity, and cost have become exacerbated.

The relatively new requirements from the main memory system are threefold. First, *technology scalability*: there is a new need for finding a technology that is much more scalable than DRAM in terms of capacity, energy, and cost, as described earlier. As DRAM continued to scale well from the above-100-nm to 30-nm technology nodes, the need for finding a more scalable technology was not a prevalent problem. Today, with the significant circuit and device scaling challenges DRAM has been facing below the 30-nm node [83], it is. Second, there is a relatively new need for providing *performance predictability and QoS* in the shared main memory system. As single-core systems were dominant and memory bandwidth and capacity were much less of a shared resource in the past, the need for predictable performance was much less apparent or prevalent [129]. Today, with increasingly more cores/agents on chip sharing the memory system and increasing amounts of workload consolidation, memory fairness, predictable memory performance, and techniques to mitigate memory interference have become first-class design constraints. Third, there is a great need for much higher *energy/power/bandwidth efficiency* in the design of the main memory system. Higher efficiency in terms of energy, power, and bandwidth enables the design of much more scalable systems where main memory is shared between many agents, and can enable new applications in almost all domains where computers are used. Arguably, this is not a *new* need today, but we believe it is another first-class design constraint that has not been as traditional as performance, capacity, and cost.

3. Solution Directions and Research Opportunities

As a result of these systems, applications, and technology trends and the resulting requirements, it is our position that researchers and designers need to fundamentally rethink the way we design memory systems today to 1) overcome scaling challenges with DRAM, 2) enable the use of emerging memory technologies, 3) design memory systems that provide predictable performance and quality of service to applications and users. The rest of this article describes *our solution ideas* in these three relatively new research directions, with pointers to specific techniques when possible.[2] Since scaling challenges themselves arise due to difficulties in enhancing memory components at *solely* one level of the computing stack (e.g., the device and/or circuit levels in case of DRAM scaling), we believe effective solutions to the above challenges will require cooperation across different layers of the computing stack, from algorithms to software to microarchitecture to devices, as well as between different components of the system, including

[2]Note that this paper is *not* meant or designed to be a survey of *all* recent works in the field of memory systems. There are many such insightful works, but we do not have space in this paper to discuss them all. This paper *is* meant to outline the challenges and research directions in memory systems as *we* see them. Therefore, many of the solutions we discuss draw heavily upon *our own* past, current, and future research. We believe this will be useful for the community as the directions we have pursued and are pursuing are hopefully fundamental challenges for which other solutions and approaches would be greatly useful to develop. We look forward to similar papers from other researchers describing their perspectives and solution directions/ideas.

processors, memory controllers, memory chips, and the storage subsystem. As much as possible, we will give examples of such cross-layer solutions and directions.

4. New Research Challenge 1: New DRAM Architectures

DRAM has been the choice technology for implementing main memory due to its relatively low latency and low cost. DRAM process technology scaling has enabled lower cost per unit area by enabling reductions in DRAM cell size for a long time. Unfortunately, further scaling of DRAM cells has become costly [4, 10, 70, 83, 90, 102, 124] due to increased manufacturing complexity/cost, reduced cell reliability, and potentially increased cell leakage leading to high refresh rates. Recently, a paper by Samsung and Intel [83] has also discussed the key scaling challenges of DRAM at the circuit level. They have identified three major challenges as impediments to effective scaling of DRAM to smaller technology nodes: 1) the growing cost of refreshes [114], 2) increase in write latency, and 3) variation in the retention time of a cell over time [115]. In light of such challenges, we believe there are at least the following key issues to tackle in order to design new DRAM architectures that are much more scalable:

1) reducing the negative impact of refresh on energy, performance, QoS, and density scaling [28, 83, 86, 114, 115],

2) improving reliability of DRAM at low cost [86, 97, 122, 145],

3) improving DRAM parallelism/bandwidth [28, 96], latency [109, 110], and energy efficiency [96, 109, 114],

4) minimizing data movement between DRAM and processing elements, which causes high latency, energy, and bandwidth consumption, by doing more operations on the DRAM and the memory controllers [167],

5) reducing the significant amount of waste in today's main memories in which much of the fetched/stored data can be unused due to coarse-granularity management [126, 153–155, 187, 199].

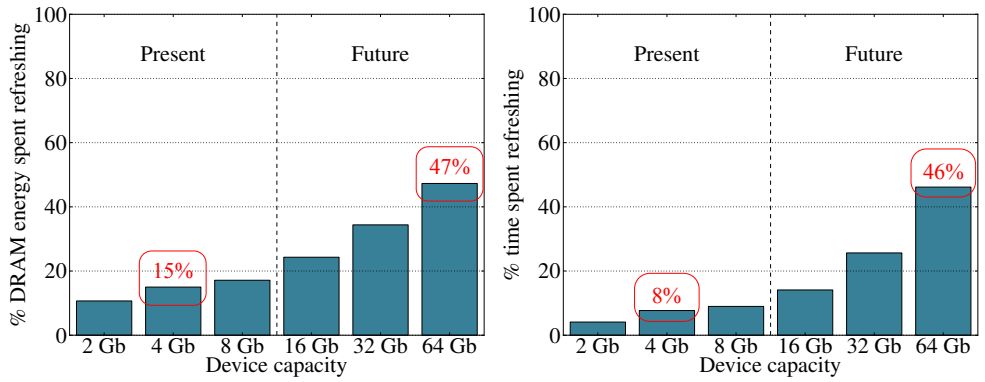
Traditionally, DRAM devices have been separated from the rest of the system with a rigid interface, and DRAM has been treated as a *passive* slave device that simply responds to the commands given to it by the memory controller. We believe the above key issues can be solved more easily if we rethink the DRAM architecture and functions, and redesign the interface such that DRAM, controllers, and processors closely cooperate. We call this high-level solution approach *system-DRAM co-design*. We believe key technology trends, e.g., the 3D stacking of memory and logic [5, 119, 188] and increasing cost of scaling DRAM solely via circuit-level approaches [70, 83, 90, 124], enable such a co-design to become increasingly more feasible. We proceed to provide several examples from our recent research that tackle the problems of refresh (and retention errors), parallelism, reliability, latency, energy efficiency, in-memory computation, and capacity and bandwidth waste.

4.1. Reducing Refresh Impact

With higher DRAM capacity, more cells need to be refreshed at likely higher rates than today. Our recent work [114] indicates that refresh rate limits DRAM density scaling: a hypothetical 64Gb DRAM device would spend 46% of its time and 47% of all DRAM energy for refreshing its rows, as opposed to typical 4Gb devices of today that spend 8% of the time and 15% of the DRAM energy on refresh (as shown in Figure 1). For instance, a modern supercom-

puter may have 1PB of memory in total [6]. If we assume this memory is built from 8Gb DRAM devices and a nominal refresh rate, 7.8kW of power would be expended, on average, just to refresh the entire 1PB memory. This is quite a large number, just to ensure the memory correctly keeps its contents! And, this power is *always* spent regardless of how much the supercomputer is utilized.

Today's DRAM devices refresh all rows at the same worst-case rate (e.g., every 64ms). However, only a small number of weak rows require a high refresh rate [86, 91, 115] (e.g., only ~1000 rows in 32GB DRAM require to be refreshed more frequently than every 256ms). Retention-Aware Intelligent DRAM Refresh (RAIDR) [114] exploits this observation: it groups DRAM rows into bins (implemented as Bloom filters [16] to minimize hardware overhead) based on the retention time of the weakest cell within each row. Each row is refreshed at a rate corresponding to its retention time bin. Since few rows need high refresh rate, one can use very few bins to achieve large reductions in refresh counts: our results show that RAIDR with three bins (1.25KB hardware cost) reduces refresh operations by ~75%, leading to significant improvements in system performance and energy efficiency as described by Liu et al. [114].



a) Power consumption b) Throughput Loss

Figure 1. Impact of refresh in current (DDR3) and projected DRAM devices. Reproduced from [114]

Like RAIDR, other approaches have also been proposed to take advantage of the retention time variation of cells across a DRAM chip. For example, some works proposed refreshing weak rows more frequently at a per-row granularity, others proposed not using memory rows with low retention times, and yet others suggested mapping critical data to cells with longer retention times so that critical data is not lost [7, 72, 89, 118, 149, 190] – see [114, 115] for a discussion of such techniques. Such approaches that exploit non-uniform retention times across DRAM require accurate retention time profiling mechanisms. Understanding of retention time as well as error behavior of DRAM devices is a critical research topic, which we believe can enable other mechanisms to tolerate refresh impact and errors at low cost. Liu et al. [115] provide an experimental characterization of retention times in modern DRAM devices to aid such understanding. Our initial results in that work, obtained via the characterization of 248 modern commodity DRAM chips from five different DRAM manufacturers, suggest that the retention time of cells in a modern device is largely affected by two phenomena: 1) Data Pattern Dependence, where the retention time of each DRAM cell is significantly affected by the data stored in other DRAM cells, 2) Variable Retention Time, where the retention time of a DRAM cell changes unpredictably over time. These two phenomena pose challenges against accurate and reliable determination of the retention time of DRAM cells, online or offline. A promising area of future

research is to devise techniques that can identify retention times of DRAM cells in the presence of data pattern dependence and variable retention time. Khan et al.'s recent work [86] provides more analysis of the effectiveness of conventional error mitigation mechanisms for DRAM retention failures and proposes *online retention time profiling* as a solution for identifying retention times of DRAM cells as a potentially promising approach in future DRAM systems. We believe developing such system-level techniques that can detect and exploit DRAM characteristics online, during system operation, will be increasingly valuable as such characteristics will become much more difficult to accurately determine and exploit by the manufacturers due to the scaling of technology to smaller nodes.

4.2. Improving DRAM Reliability: Better DRAM Error Management

As DRAM technology scales to smaller node sizes, its reliability becomes more difficult to maintain at the circuit and device levels. In fact, we already have evidence of the difficulty of maintaining DRAM reliability from the DRAM chips operating in the field today. Our recent research [97] showed that a majority of the DRAM chips manufactured between 2010-2014 by three major DRAM vendors exhibit a particular failure mechanism called *row hammer*: by activating a row enough times within a refresh interval, one can corrupt data in nearby DRAM rows. The source code is available at [3]. This is an example of a *disturbance error* where the access of a cell causes disturbance of the value stored in a nearby cell due to cell-to-cell coupling (i.e., interference) effects, some of which are described by our recent works [97, 115]. Such interference-induced failure mechanisms are well-known in any memory that pushes the limits of technology, e.g., NAND flash memory (see Section 7). However, in case of DRAM, manufacturers have been quite successful in containing such effects until recently. Clearly, the fact that such failure mechanisms have become difficult to contain and that they have already slipped into the field shows that failure/error management in DRAM has become a significant problem. We believe this problem will become even more exacerbated as DRAM technology scales down to smaller node sizes. Hence, it is important to research both the (new) failure mechanisms in future DRAM designs as well as mechanisms to tolerate them. Towards this end, we believe it is critical to gather insights from the field by 1) experimentally characterizing DRAM chips using controlled testing infrastructures [86, 97, 110, 115], 2) analyzing large amounts of data from the field on DRAM failures in large-scale systems [164, 172, 173], 3) developing models for failures/errors based on these experimental characterizations, and 4) developing new mechanisms at the system and architecture levels that can take advantage of such models to tolerate DRAM errors.

Looking forward, we believe that increasing cooperation between the DRAM device and the DRAM controller as well as other parts of the system, including system software, would be greatly beneficial for identifying and tolerating DRAM errors. For example, such cooperation can enable the communication of information about *weak* (or, unreliable) cells and the characteristics of different rows or physical memory regions from the DRAM device to the system. The system can then use this information to optimize data allocation and movement, refresh rate management, and error tolerance mechanisms. Low-cost error tolerance mechanisms are likely to be enabled more efficiently with such coordination between DRAM and the system. In fact, as DRAM technology scales and error rates increase, it might become increasingly more difficult to maintain the illusion that DRAM is a perfect, error-free storage device (the *row hammer* failure mechanism [97] already provides evidence for this). DRAM may start looking increasingly

like flash memory, where the memory controller manages errors so that an acceptable specified uncorrectable bit error rate is satisfied [20, 22]. We envision a *DRAM Translation Layer (DTL)*, not unlike the *Flash Translation Layer (FTL)* of today in spirit (which is decoupled from the processor and performs a wide variety of management functions for flash memory, including error correction, garbage collection, read/write scheduling, data mapping, etc.), can enable better scaling of DRAM memory into the future by not only enabling easier error management but also opening up new opportunities to perform computation, mapping and metadata management close to memory. This can become especially feasible in the presence of the trend of combining the DRAM controller and DRAM via 3D stacking. What should the interface be to such a layer and what should be performed in the DTL are promising areas of future research.

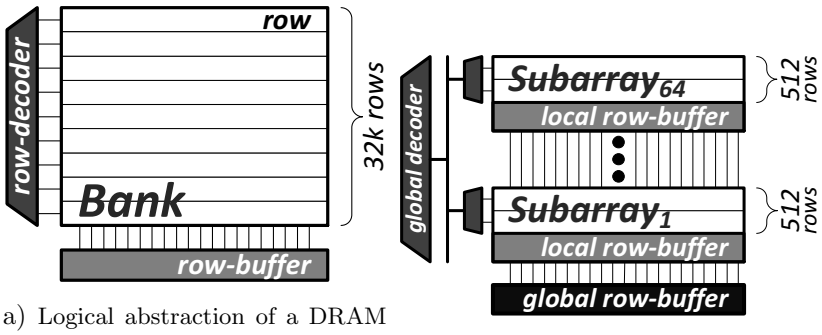
4.3. Improving DRAM Parallelism

A key limiter of DRAM parallelism is bank conflicts. Today, a bank is the finest-granularity independently accessible memory unit in DRAM. If two accesses go to the same bank, one has to *completely* wait for the other to finish before it can be started (see Figure 2). We have recently developed mechanisms, called SALP (subarray level parallelism) [96], that exploit the internal subarray structure of the DRAM bank (Figure 2) to *mostly* parallelize two requests that access the same DRAM bank. The key idea is to reduce the hardware sharing between DRAM subarrays so that accesses to the same bank but different subarrays can be initiated in a pipelined manner. This mechanism requires the exposure of the internal subarray structure of a DRAM bank to the controller and the design of the controller to take advantage of this structure. Our results show significant improvements in performance and energy efficiency of main memory due to parallelization of requests and improvement of row buffer hit rates (as row buffers of different subarrays can be kept active) at a low DRAM area overhead of 0.15%. Exploiting SALP achieves most of the benefits of increasing the number of banks at much lower area and power overhead. Exposing the subarray structure of DRAM to other parts of the system, e.g., to system software or memory allocators, can enable data placement and partitioning mechanisms that can improve performance and efficiency even further.

Note that other approaches to improving DRAM parallelism especially in the presence of refresh and long write latencies are also promising to be investigated. Chang et al. [28] discuss mechanisms to improve the parallelism between refreshes and reads/write requests, and Kang et al. [83] discuss the use of SALP as a promising method to tolerate long write latencies to DRAM, which they identify as one of the three key scaling challenges for DRAM, in addition to refresh and variable retention time. We refer the reader to these works for more information about the proposed parallelization techniques.

4.4. Reducing DRAM Latency and Energy

The DRAM industry has so far been primarily driven by the cost-per-bit metric: provide maximum capacity for a given cost. As shown in Figure 3, DRAM chip capacity has increased by approximately 16x in 12 years while the DRAM latency reduced by only approximately 20%. This is the result of a deliberate choice to maximize capacity of a DRAM chip while minimizing its cost. We believe this choice needs to be revisited in the presence of at least two key trends. First, DRAM latency is becoming more important especially for response-time critical workloads that require QoS guarantees [45]. Second, DRAM capacity is becoming very hard to scale and as



a) Logical abstraction of a DRAM bank

b) Implementation of a DRAM bank

Figure 2. DRAM Bank Organization. Reproduced from [96]

a result manufacturers likely need to provide new values for the DRAM chips, leading to more incentives for the production of DRAMs that are optimized for objectives other than mainly capacity maximization.

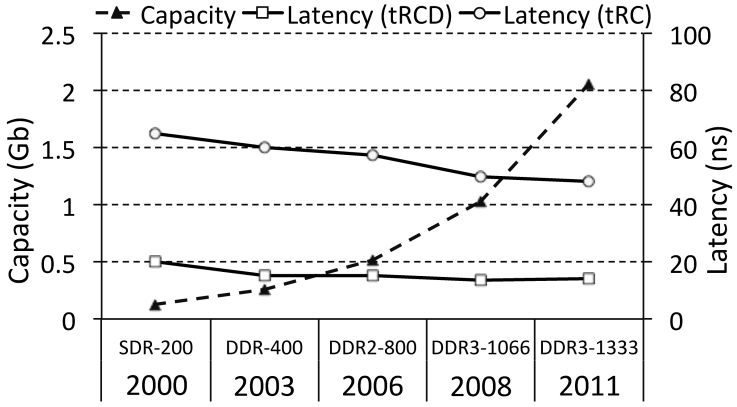


Figure 3. DRAM Capacity & Latency Over Time. Reproduced from [109]

To mitigate the high area overhead of DRAM sensing structures, commodity DRAMs (shown in Figure 4a) connect many DRAM cells to each sense-amplifier through a wire called a bitline. These bitlines have a high parasitic capacitance due to their long length, and this bitline capacitance is the dominant source of DRAM latency. Specialized low-latency DRAMs (shown in Figure 4b) use shorter bitlines with fewer cells, but have a higher cost-per-bit due to greater sense-amplifier area overhead. We have recently shown that we can architect a heterogeneous-latency bitline DRAM, called Tiered-Latency DRAM (TL-DRAM) [109], shown in Figure 4c, by dividing a long bitline into two shorter segments using an isolation transistor: a low-latency segment can be accessed with the latency and efficiency of a short-bitline DRAM (by turning off the isolation transistor that separates the two segments) while the high-latency segment enables high density, thereby reducing cost-per-bit. The additional area overhead of TL-DRAM is approximately 3% over commodity DRAM. Significant performance and energy improvements can be achieved by exposing the two segments to the memory controller and system software such that appropriate data is cached or allocated into the low-latency segment. We expect such approaches that design and exploit heterogeneity to enable/achieve the best of multiple worlds [132] in the memory system can lead to other novel mechanisms that can overcome difficult contradictory tradeoffs in design.

A recent paper by Lee et al. [110] exploits the extra margin built into DRAM timing parameters to reliably reduce DRAM latency when such extra margin is really not necessary (e.g.,

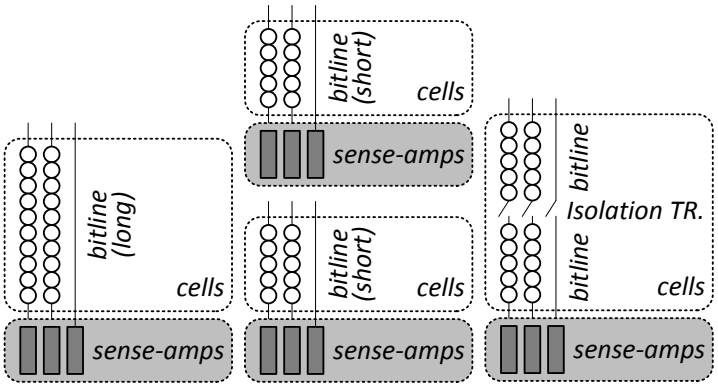


Figure 4. Cost Optimized Commodity DRAM (a), Latency Optimized DRAM (b), Tiered-Latency DRAM (c). Reproduced from [109]

when the operating temperature is low). The standard DRAM timing constraints are designed to ensure correct operation for the cell with the *lowest retention time* at the *highest acceptable operating temperature*. Lee et al. [110] make the observation that a significant majority of DRAM modules do not exhibit the worst case behavior and that most systems operate at a temperature much lower than the highest acceptable operating temperature, enabling the opportunity to significantly reduce the timing constraints. They introduce Adaptive-Latency DRAM (AL-DRAM), which dynamically measures the operating temperature of each DIMM and employs timing constraints optimized for that DIMM at that temperature. Results of their profiling experiments on 82 modern DRAM modules show that AL-DRAM can reduce the DRAM timing constraints by an average of 43% and up to 58%. This reduction in latency translates to a 14% average improvement in overall system performance across a wide variety of applications on the evaluated real systems. We believe such approaches to reducing memory latency (and energy) by exploiting *common-case device characteristics and operating conditions* are very promising: instead of *always* incurring the worst-case latency and energy overheads due to homogeneous, one-size-fits-all parameters, adapt the parameters dynamically to fit the common-case operating conditions.

Another promising approach to reduce DRAM energy is the use of dynamic voltage and frequency scaling (DVFS) in main memory [44, 46]. David et al. [44] make the observation that at low memory bandwidth utilization, lowering memory frequency/voltage does not significantly alter memory access latency. Relatively recent works have shown that adjusting memory voltage and frequency based on predicted memory bandwidth utilization can provide significant energy savings on both real [44] and simulated [46] systems. Going forward, memory DVFS can enable dynamic heterogeneity in DRAM channels, leading to new customization and optimization mechanisms. Also promising is the investigation of more fine-grained power management methods within the DRAM rank and chips for both active and idle low power modes.

4.5. Exporting Bulk Data Operations to DRAM: Enabling In-Memory Computation

Today's systems waste significant amount of energy, DRAM bandwidth and time (as well as valuable on-chip cache space) by sometimes unnecessarily moving data from main memory to processor caches. One example of such wastage sometimes occurs for bulk data copy and

initialization operations in which a page is copied to another or initialized to a value. If the copied or initialized data is not immediately needed by the processor, performing such operations within DRAM (with relatively small changes to DRAM) can save significant amounts of energy, bandwidth, and time. We observe that a DRAM chip internally operates on bulk data at a row granularity. Exploiting this internal structure of DRAM can enable page copy and initialization to be performed entirely within DRAM without bringing any data off the DRAM chip, as we have shown in recent work [167]. If the source and destination page reside within the same DRAM subarray, our results show that a page copy can be accelerated by more than an order of magnitude (~11 times), leading to an energy reduction of ~74 times and *no* wastage of DRAM data bus bandwidth [167]. The key idea is to capture the contents of the source row in the sense amplifiers by 1) activating the row, then 2) *deactivating* the source row (using a new command which introduces very little hardware cost, amounting to less than 0.03% of DRAM chip area), and 3) immediately activating the destination row, which causes the sense amplifiers to drive their contents into the destination row, effectively accomplishing the page copy (shown at a high level in fig. 5). Doing so reduces the latency of a 4KB page copy operation from ~1000ns to less than 100ns in an existing DRAM chip. Applications that have significant page copy and initialization experience large system performance and energy efficiency improvements [167]. Future software can be designed in ways that can take advantage of such fast page copy and initialization operations, leading to benefits that may not be apparent in today's software that tends to minimize such operations due to their current high cost.

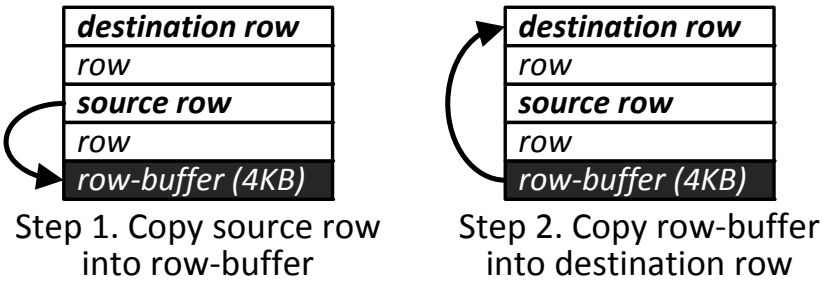


Figure 5. High-level idea behind RowClone's in-DRAM page copy mechanism

Going forward, we believe acceleration of other bulk data movement and computation operations in or very close to DRAM, via similar low-cost architectural support mechanisms, can enable promising savings in system energy, latency, and bandwidth. Given the trends and requirements described in Section 1, it is time to re-examine the partitioning of computation between processors and DRAM, treating memory as a first-class accelerator as an integral part of a heterogeneous parallel computing system [132].

4.6. Minimizing Memory Capacity and Bandwidth Waste

Storing and transferring data at large granularities (e.g., pages, cache blocks) within the memory system leads to large inefficiency when most of the large granularity is not needed [82, 101, 125, 126, 157, 166, 187, 199, 200]. In addition, much of the data stored in memory has significant redundancy [8, 13, 52, 59, 153–155, 198]. Two promising research directions are to develop techniques that can 1) efficiently provide fine granularity access/storage when enough and large granularity access/storage only when needed, 2) efficiently compress data in main memory and caches without significantly increasing latency and system complexity. Our

results with new low-cost, low-latency cache compression [153] and memory compression [154] techniques and frameworks are promising, providing high compression ratios at low complexity and latency. For example, the key idea of *Base-Delta-Immediate compression* [153] is that many cache blocks have low dynamic range in the values they store, i.e., the differences between values stored in the cache block are small. Such a cache block can be encoded using a base value and an array of much smaller (in size) differences from that base value, which together occupy much less space than storing the full values in the original cache block. This compression algorithm has low decompression latency as the cache block can be reconstructed using a vector addition (or potentially even vector concatenation). It reduces memory bandwidth requirements, better utilizes memory/cache space, while minimally impacting the latency to access data. Granularity management and data compression support can potentially be integrated into DRAM controllers or partially provided within DRAM, and such mechanisms can be exposed to software, which can enable higher energy savings and higher performance improvements. Management techniques for compressed caches and memories (e.g., [155]) as well as flexible granularity memory system designs, software techniques/designs to take better advantage of cache/memory compression and flexible-granularity, and techniques to perform computations on compressed memory data are quite promising directions for future research.

4.7. Co-Designing DRAM Controllers and Processor-Side Resources

Since memory bandwidth is a precious resource, coordinating the decisions made by processor-side resources better with the decisions made by memory controllers to maximize memory bandwidth utilization and memory locality is a promising area of more efficiently utilizing DRAM. Lee et al. [106] and Stuecheli et al. [175] both show that orchestrating last-level cache writebacks such that dirty cache lines to the same row are evicted together from the cache improves DRAM row buffer locality of write accesses, thereby improving system performance. Going forward, we believe such coordinated techniques between the processor-side resources and memory controllers will become increasingly more effective as DRAM bandwidth becomes even more precious. Mechanisms that predict and convey slack in memory requests [42, 43], that orchestrate the on-chip scheduling of memory requests to improve memory bank parallelism [108] and that reorganize cache metadata for more efficient bulk (DRAM row granularity) tag lookups [168] can also enable more efficient memory bandwidth utilization.

5. New Research Challenge 2: Emerging Memory Technologies

While DRAM technology scaling is in jeopardy, some emerging technologies seem more scalable. These include phase-change memory PCM, spin-transfer torque magnetoresistive RAM (STT-MRAM) and resistive RAM (RRAM). These emerging technologies usually provide a tradeoff, and seem unlikely to *completely* replace DRAM (evaluated in [102–104] for PCM and in [100] for STT-MRAM), as they are not strictly superior to DRAM. For example, PCM is advantageous over DRAM because it 1) has been demonstrated to scale to much smaller feature sizes [102, 163, 192] and can store multiple bits per cell [202, 203], promising higher density, 2) is non-volatile and as such requires no refresh (which is a key scaling challenge of DRAM as we discussed in Section 4.1), and 3) has low idle power consumption. On the other hand, PCM has significant shortcomings compared to DRAM, which include 1) higher read latency and read energy, 2) *much* higher write latency and write energy, 3) limited endurance for a given PCM

cell, a problem that does not exist (practically) for a DRAM cell, and 4) potentially difficult-to-handle reliability issues, such as the problem of *resistance drift* [165]. As a result, an important research challenge is how to utilize such emerging technologies at the system and architecture levels so that they can augment or perhaps even replace DRAM.

Our initial experiments and analyses [102–104] that evaluated the complete replacement of DRAM with PCM showed that one would require reorganization of peripheral circuitry of PCM chips (with the goal of absorbing writes and reads before they update or access the PCM cell array) to enable PCM to get close to DRAM performance and efficiency. These initial results are reported in Lee et al. [102–104] and they show that the performance, energy, and endurance of PCM chips can be greatly improved with the proposed techniques. We have also reached a similar conclusion upon evaluation of the complete replacement of DRAM with STT-MRAM [100]: reorganization of peripheral circuitry of STT-MRAM chips (with the goal of minimizing the number of writes to the STT-MRAM cell array, as write operations are high-latency and high-energy in STT-MRAM) enables an STT-MRAM based main memory to be more energy-efficient than a DRAM-based main memory.

One can achieve more efficient designs of PCM (or STT-MRAM) chips by taking advantage of the non-destructive nature of reads, which enables simpler and narrower row buffer organizations [125]. Unlike in DRAM, the entire memory row does not need to be buffered in a device where reading a memory row does not destroy the data stored in the row. Meza et al. [125] show that having narrow row buffers in emerging non-volatile devices can greatly reduce main memory dynamic energy compared to a DRAM baseline with large row sizes, without greatly affecting endurance, and for some NVM technologies, lead to improved performance. Going forward, designing systems, memory controllers and memory chips taking advantage of the specific property of non-volatility of emerging technologies seems promising.

We believe emerging technologies enable at least three major system-level opportunities that can improve overall system efficiency: 1) hybrid main memory systems, 2) non-volatile main memory, 3) merging of memory and storage. We briefly touch upon each.

5.1. Hybrid Main Memory

A hybrid main memory system [29, 47, 126, 156, 159, 162, 201] consists of multiple different technologies or multiple different types of the same technology with differing characteristics, e.g., performance, cost, energy, reliability, endurance. A key question is how to manage data allocation and movement between the different technologies so that one can achieve the best of (or close to the best of) the desired performance metrics. In other words, we would like to exercise the advantages of each technology as much as possible while hiding the disadvantages of any technology. Potential technologies include DRAM, 3D-stacked DRAM, embedded DRAM, PCM, STT-MRAM, other resistive memories, flash memory, forms of DRAM that are optimized for different metrics and purposes, etc. An example hybrid main memory system consisting of a large amount of PCM as main memory and a small amount of DRAM as its cache is depicted in Figure 6.

The design space of hybrid memory systems is large, and many potential questions exist. For example, should all memories be part of main memory or should some of them be used as a cache of main memory (or should there be configurability)? What technologies should be software visible? What component of the system should manage data allocation and movement? Should these tasks be done in hardware, software, or collaboratively? At what granularity should

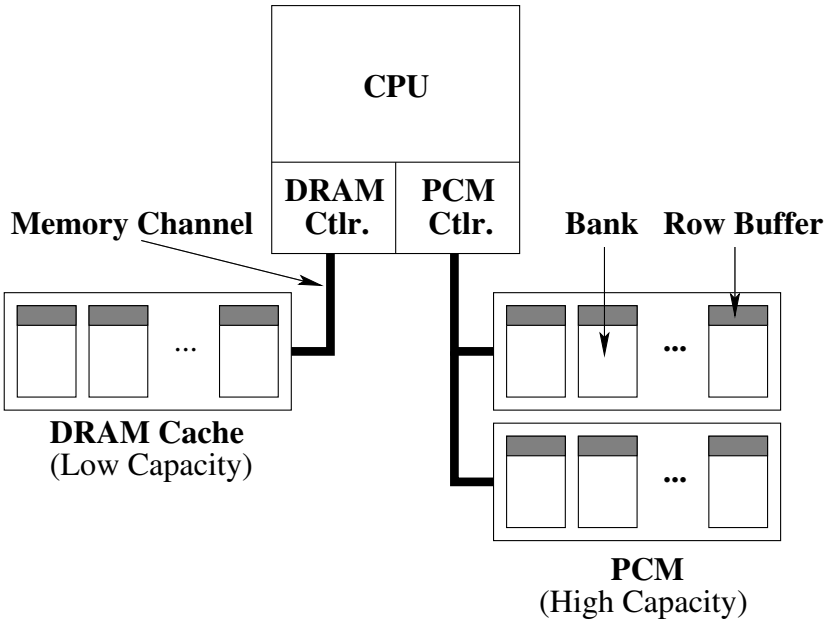


Figure 6. An example hybrid main memory system organization using PCM and DRAM chips. Reproduced from [201]

data be moved between different memory technologies? Some of these questions are tackled in [29, 47, 126, 156, 159, 162, 201], among other works recently published in the computer architecture community. For example, Yoon et al. [201] make the key observation that row buffers are present in both DRAM and PCM (see fig. 6), and they have (or can be designed to have) the same latency and bandwidth in both DRAM and PCM. Yet, row buffer misses are much more costly in terms of latency, bandwidth, and energy in PCM than in DRAM. To exploit this, they devise a policy that avoids accessing in PCM data that frequently causes row buffer misses. Hardware or software can dynamically keep track of such data and allocate/cache it in DRAM while keeping data that frequently hits in row buffers in PCM. PCM also has much higher write latency/power than read latency/power: to take this into account, the allocation/caching policy is biased such that pages that are written to more likely stay in DRAM [201].

Note that hybrid memory does not need to consist of completely different underlying technologies. A promising approach is to combine multiple different DRAM chips, optimized for different purposes. For example, recent works proposed the use of low-latency and high-latency DIMMs in separate memory channels and allocating performance-critical data to low-latency DIMMs to improve performance and energy-efficiency at the same time [29], or the use of highly-reliable DIMMs (protected with ECC) and unreliable DIMMs in separate memory channels and allocating error-vulnerable data to highly-reliable DIMMs to maximize server availability while minimizing server memory cost [122]. We believe these approaches are quite promising for scaling the DRAM technology into the future by specializing DRAM chips for different purposes. These approaches that exploit heterogeneity *do* increase system complexity but that complexity may be warranted if it is lower than the complexity of scaling DRAM chips using the same optimization techniques the DRAM industry has been using so far.

5.2. Making Non-volatile Main Memory Reliable and Secure

Non-volatility of main memory opens up new opportunities that can be exploited by higher levels of the system stack to improve performance and reliability/consistency (see, for example, [38, 48]). Researching how to adapt applications and system software to utilize fast, byte-addressable non-volatile main memory is an important research direction to pursue [127].

On the flip side, the same non-volatility can lead to potentially unforeseen security and privacy issues: critical and private data can persist long after the system is powered down [33], and an attacker can take advantage of this fact. Wearout issues of emerging technology can also cause attacks that can intentionally degrade memory capacity in the system [158, 171]. Securing non-volatile main memory is therefore an important systems challenge.

5.3. Merging of Memory and Storage

One promising opportunity fast, byte-addressable, non-volatile emerging memory technologies open up is the design of a system and applications that can manipulate *persistent data directly in memory* instead of going through a slow storage interface. This can enable not only much more efficient systems but also new and more robust applications. We discuss this opportunity in more detail below.

Traditional computer systems have a two-level storage model: they access and manipulate 1) volatile data in main memory (DRAM, today) with a fast load/store interface, 2) persistent data in storage media (flash and hard disks, today) with a slower file system interface. Unfortunately, such a decoupled memory/storage model managed via vastly different techniques (fast, hardware-accelerated memory management units on one hand, and slow operating/file system (OS/FS) software on the other) suffers from large inefficiencies in locating data, moving data, and translating data between the different formats of these two levels of storage that are accessed via two vastly different interfaces, leading to potentially large amounts of wasted work and energy [127, 170]. The two different interfaces arose largely due to the large discrepancy in the access latencies of conventional technologies used to construct volatile memory (DRAM) and persistent storage (hard disks and flash memory).

Today, new non-volatile memory technologies (NVM), e.g, PCM, STT-MRAM, RRAM, show the promise of storage capacity and endurance similar to or better than flash memory at latencies comparable to DRAM. This makes them prime candidates for providing applications a persistent *single-level store* with a single load/store-like interface to access all system data (including volatile and persistent data). In fact, if we keep the traditional two-level memory/storage model in the presence of these fast NVM devices as part of storage, the operating system and file system code for locating, moving, and translating persistent data from the non-volatile NVM devices to volatile DRAM for manipulation purposes becomes a great bottleneck, causing most of the memory energy consumption and degrading performance by an order of magnitude in some data-intensive workloads, as we showed in recent work [127]. With energy as a key constraint, and in light of modern high-density NVM devices, a promising research direction is to unify and coordinate the management of volatile memory and persistent storage in a single level, to eliminate wasted energy and performance, and to simplify the programming model at the same time.

To this end, Meza et al. [127] describe the vision and research challenges of a persistent memory manager (PMM), a hardware acceleration unit that coordinates and unifies memory/storage

management in a single address space that spans potentially multiple different memory technologies (DRAM, NVM, flash) via hardware/software cooperation. Figure 7 depicts an example PMM, programmed using a load/store interface (with persistent objects) and managing an array of heterogeneous devices.

```
1 int main(void) {
2   // data in file.dat is persistent
3   FILE myData = "file.dat";
4   myData = new int[64];
5 }
6 void updateValue(int n, int value) {
7   FILE myData = "file.dat";
8   myData[n] = value; // value is persistent
9 }
```

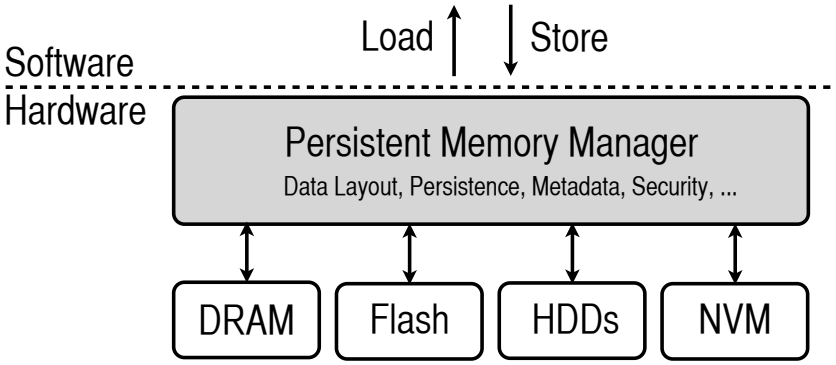


Figure 7. An example Persistent Memory Manager (PMM). Reproduced from [127]

The spirit of the PMM unit is much like the virtual memory management unit of a modern virtual memory system used for managing working memory, but it is fundamentally different in that it redesigns/rethinks the virtual memory and storage abstractions and unifies them in a different interface supported by scalable hardware mechanisms. The PMM: 1) exposes a load/store interface to access persistent data, 2) manages data placement, location, persistence semantics, and protection (across multiple memory devices) using both dynamic access information and hints from the application and system software, 3) manages metadata storage and retrieval, needed to support efficient location and movement of persistent data, and 4) exposes hooks and interfaces for applications and system software to enable intelligent data placement and persistence management. Our preliminary evaluations show that the use of such a unit, if scalable and efficient, can greatly reduce the energy inefficiency and performance overheads of the two-level storage model, improving both performance and energy-efficiency of the overall system, especially for data-intensive workloads [127].

We believe there are challenges to be overcome in the design, use, and adoption of such a unit that unifies working memory and persistent storage. These challenges include:

1) How to devise efficient and scalable data mapping, placement, and location mechanisms (which need to be hardware/software cooperative).
2) How to ensure that the consistency and protection requirements of different types of data are adequately, correctly, and reliably satisfied (One example recent work tackled the problem of providing storage consistency at high performance [121]). How to enable the reliable and effective coexistence and manipulation of volatile and persistent data.

3) How to redesign applications such that they can take advantage of the unified memory/storage interface and make the best use of it by providing appropriate hints for data allocation and placement to the persistent memory manager.
4) How to provide efficient and high-performance backward compatibility mechanisms for enabling and enhancing existing memory and storage interfaces in a single-level store. These techniques can seamlessly enable applications targeting traditional two-level storage systems to take advantage of the performance and energy-efficiency benefits of systems employing single-level stores. We believe such techniques are needed to ease the software transition to a radically different storage interface.
5) How to design system resources such that they can concurrently handle applications/access-patterns that manipulate persistent data as well as those that manipulate non-persistent data. (One example recent work [204] tackled the problem of designing effective memory scheduling policies in the presence of these two different types of applications/access-patterns.)

6. New Research Challenge 3: Predictable Performance

Since memory is a shared resource between multiple cores (or, agents, threads, or applications and virtual machines), as shown in Figure 8, different applications contend for bandwidth and capacity at the different components of the memory system, such as memory controllers, interconnects and caches. As such, memory contention, or memory interference, between different cores critically affects both the overall system performance and each application's performance. Our past work (e.g., [129, 137, 138, 142]) showed that application-unaware design of memory controllers, and in particular, memory scheduling algorithms, leads to uncontrolled interference of applications in the memory system. Such uncontrolled interference can lead to denial of service to some applications [129], low system performance [137, 138, 142], unfair and unpredictable application slowdowns [53, 56, 137, 176]. For instance, Figure 9 shows the application slowdowns when two applications are run together on a simulated two-core system where the two cores share the main memory (including the memory controllers). The application leslie3d (from the SPEC CPU2006 suite) slows down significantly due to interference from the co-running application. Furthermore, leslie3d's slowdown depends heavily on the co-running application. It slows down by 2x when run with gcc, whereas it slows down by more than 5x when run with mcf, an application that exercises the memory significantly. Our past works have shown that similarly unpredictable and uncontrollable slowdowns happen in both existing systems (e.g., [88, 129]) and simulated systems (e.g., [53, 56, 137, 138, 142, 176]), across a wide variety of workloads

Building QoS and application awareness into the different components of the memory system such as memory controllers, caches, interconnects is important to control interference at these different components and mitigate/eliminate unfairness and unpredictability. Towards this end, previous works have explored two different solution directions: 1) to mitigate interference, thereby reducing application slowdowns and improving overall system performance, 2) to precisely quantify and control the impact of interference on application slowdowns, thereby providing performance guarantees to applications that need such guarantees. We discuss these two different approaches and associated research problems next.

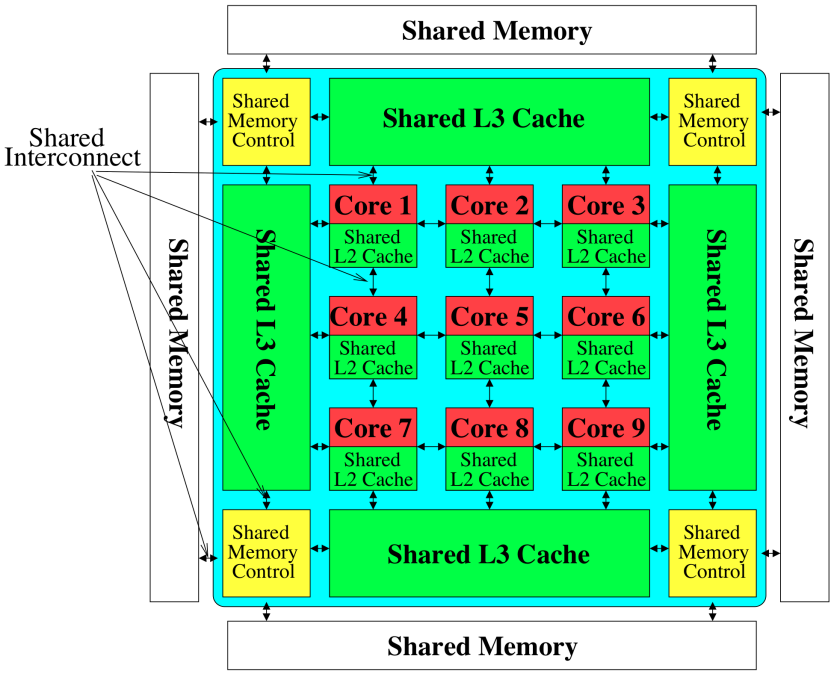
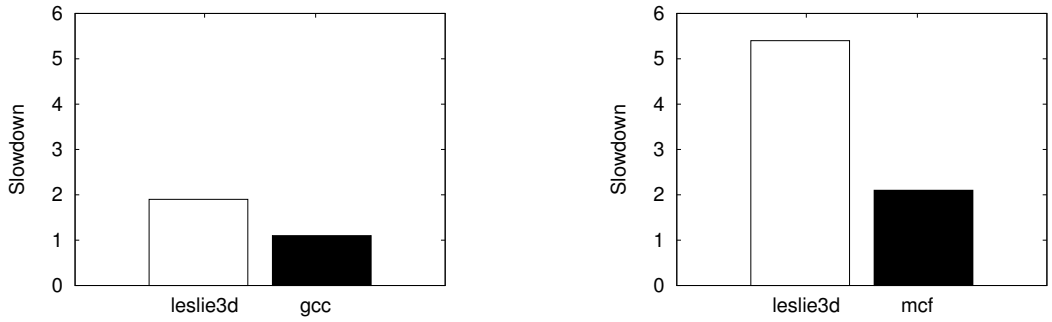


Figure 8. A typical multicore system. Reproduced from [133]



a) leslie3d co-running with gcc b) leslie3d co-running with mcf

Figure 9. High and unpredictable application slowdowns

6.1. Mitigating Interference

In order to mitigate interference at the different components of memory system, two kinds of approaches have been explored. The resources could be either *smart* (i.e., aware of threads' or applications' interference in memory) or *dumb* (i.e., unaware of threads' or applications' interference in memory) as we describe below.[3]

6.1.1. Smart Resources

The *smart resources* approach equips resources (such as memory controllers, interconnects and caches) with the intelligence to 1) be aware of interference characteristics between applications and 2) prevent unfair application slowdowns and performance degradation. Several of our past works have taken the *smart resources* approach and designed QoS-aware memory controllers [11, 54, 88, 93–95, 105, 130, 137, 138, 142, 176, 177] and inter-

[3]For the rest of this article, without loss of generality, we use the terms *thread* and *application* interchangeably. From the point of view of a *smart* resource, the resource needs to be communicated at least the *hardware context identifier* of the application/thread that is generating requests to be serviced.

connects [27, 41–43, 64, 66, 67, 128, 147, 148]. Our and other previous works have explored smart shared cache management policies that 1) allocate shared cache capacity to applications in a manner that is aware of their cache utility [161, 195], 2) modify the cache replacement and insertion policies to be aware of the data reuse and memory access behavior of applications [75, 76, 87, 160, 166, 169]. All these QoS-aware schemes enable resources (such as memory controllers, interconnects and caches) to detect interference between applications by means of monitoring their access characteristics and allocate resources such as memory bandwidth, interconnect link bandwidth and cache capacity to applications so that interference between applications is mitigated.

We provide several brief examples of the *smart resources* approach by focusing on QoS-aware memory controllers [11, 54, 88, 93–95, 105, 130, 137, 138, 142, 176, 177].

Mutlu and Moscibroda [129, 137] devised some of the first fair memory controllers. Their memory scheduler dynamically estimates the slowdown of each application and prioritizes applications' requests in a way that balances the slowdowns. In a later work, Mutlu and Moscibroda [138, 142] show that uncontrolled inter-thread interference in the memory controller can destroy the memory-level parallelism (MLP) [35, 51, 63, 140, 141, 143, 144, 150] and serialize requests of individual threads, leading to significant degradation in both single-thread and system performance in multi-core/multi-threaded systems. Hence, many techniques devised by computer architects to parallelize a thread's memory requests to tolerate memory latency by exploiting MLP, such as out-of-order execution [151, 152, 185], non-blocking caches [99], runahead execution [30, 35, 51, 136, 139–141, 143, 144] and other techniques [34, 160, 205], can become ineffective if the memory controller is not aware of threads. To overcome this and ensure the memory controller can serve each thread's requests in parallel, they introduce the idea of *thread ranking*, where a memory controller forms a rank order among threads and services threads in that order. To provide high fairness and starvation freedom, their controller employs *batching* of requests, where the memory controller groups oldest requests from each thread into a batch and services that batch before all other requests. This work, and the associated memory scheduler PAR-BS (Parallelism-Aware Batch Scheduler) has formed the basis of many future thread-aware memory scheduling policies by providing relatively simple and effective mechanisms for *both* performance and fairness in the memory controller.

Kim et al. [93] observe that applications that have received the least service from the memory controllers in the past, would benefit from quick request service (in the future) and hence, seek to prioritize such applications' requests at the memory controller, with the goal of improving system performance. They propose ATLAS [93], a QoS-aware memory controller design that monitors and ranks applications based on the amount of service each application receives at the memory controllers and prioritizes requests of applications with low *attained memory service*. ATLAS provides significant performance improvement by prioritizing applications that can benefit the most from memory service. In a later work, Kim et al. [94, 95], observe that prioritizing between *all* applications *solely based on one access characteristic*, e.g., the attained memory service in ATLAS, could unfairly slow down some applications: applications that are very memory intensive can be slowed down disproportionately. To solve this problem, they propose the thread cluster memory scheduler (TCM) [94, 95], which employs heterogeneous request scheduling policies for different types of applications: latency-sensitive vs. bandwidth-sensitive, as shown in Figure 10. TCM classifies applications into two clusters, the low- and high-memory-intensity (or, latency-sensitive and bandwidth-sensitive) clusters, prioritizes the latency-sensitive cluster, and

employs a different policy to rank and prioritize among applications within each cluster, with the goal of optimizing for both performance and fairness. Results show that TCM can achieve both high performance and fairness compared to the best schedulers of the time.

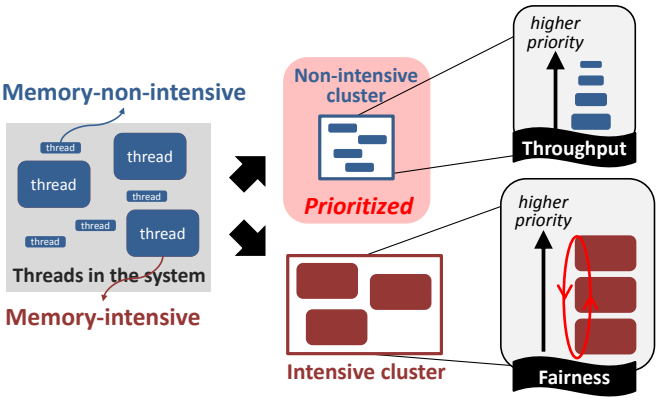


Figure 10. Operation of Thread Cluster Memory Scheduler (TCM). Reproduced from [92]

Most recently, Subramanian et al. [177] propose the blacklisting memory scheduler (BLISS) based on the observation that it is *not* necessary to employ a *full ordered ranking* across all applications, like PAR-BS, ATLAS and TCM do, because of two key reasons. First, such a full ordered ranking scheme incurs *high hardware complexity*. Second, a full ordered ranking of applications prioritizes some high-memory-intensity applications over other high-memory-intensity applications, resulting in *unfair application slowdowns*. Hence, they propose to separate applications into *only two groups* (instead of employing a full ranking of threads), one containing interference-causing applications and the other containing vulnerable-to-interference applications and prioritize the vulnerable-to-interference group over the interference-causing group. These groups are formed by monitoring the number of consecutive requests from an applications and classifying applications that generate more than a certain number of consecutive requests as interference-causing (this is called *blacklisting*). Such a scheme not only greatly reduces the hardware complexity and critical path latency of the memory scheduler (as it *does not require full ranking of all applications*), but also prevents applications from slowing down unfairly, thereby improving system performance.

It is worth noting that inter-thread interference in the memory controller among threads of the *same* application can greatly reduce that application's performance as well. Ebrahimi et al. [54] quantify the performance loss due to such interference and propose a new memory controller that dynamically estimates *critical threads* (or, *limiter threads*) in an application, which limit performance, and prioritizes such threads over others. Such an approach that identifies the most important threads, potentially using various other mechanisms [14, 18, 50, 78, 79, 178–181], and prioritizes/accelerates them in not only the memory controllers but also other resources is likely to be promising to improve parallel program performance and efficiency.

The design of memory controllers remains equally, if not more, important in the presence of persistent memory systems that store and access persistent data through the memory interface (as we discussed in Section 5.3). In fact, in such systems *memory writes* can become very frequent as persistent data needs to be flushed to main memory in a strict order determined by the storage consistency model employed in modern systems [204]. Zhao et al. [204] identified this problem and showed that existing memory controllers cannot appropriately handle interference between applications that access persistent data and applications that access volatile data

because those that write to persistent data can greatly reduce system performance and fairness due to scheduling policies that do not take into account write requests. They develop a new memory scheduling algorithm that provides a solution to this problem by more fairly handling read and write requests of different applications [204].

Finally, it is critically important to appropriately handle the interference caused by *prefetch requests* generated by hardware and software prefetchers employed in almost all modern high performance systems [69, 174, 183]. Lee et al. [105, 107, 108] show that making the memory controller dynamically decide between providing equal or lower priority to prefetch requests compared to demand requests, based on the accuracy of prefetches, can greatly improve system performance and fairness. Ebrahimi et al. [55, 57, 58] showed that interference caused by aggressive prefetchers, even if they are accurate, can cause slowdowns to applications and reduce performance. They devise mechanisms to appropriately throttle prefetchers to reduce the negative effects of the interference caused.

All these past works on QoS-aware and interference-aware memory scheduling have shown that significant performance and fairness gains are possible by designing the memory controller to be aware of different threads/applications and different request/access characteristics and appropriately prioritizing among them. We believe ample opportunity exists for future designs that can effectively navigate the complex tradeoff space of performance, fairness, hardware complexity/cost, scheduling latency and energy efficiency. Designs that optimize for three or more of these metrics at the same time, e.g., in the spirit of BLISS [177], will especially be desirable in the future and are a promising direction for future research.

A challenge with the *smart resources* approach is the coordination of resource allocation decisions across different resources, such as main memory, interconnects and shared caches. To provide QoS across the entire system, the actions of different resources need to be coordinated. Several works examined such coordination mechanisms. One example of such a scheme is a coordinated resource management scheme proposed by Bitirgen et al. [15] that employs machine learning, specifically, an artificial neural network, to predict each application's performance for different possible resource allocations. Resources are then allocated appropriately to different applications so that a global system performance metric is optimized. Another example of such a scheme is a recent work by Wang and Martinez [191] that employs a market-dynamics-inspired mechanism to coordinate allocation decisions across resources. Approaches to coordinate resource allocation and scheduling decisions across multiple resources in the memory system, whether they use machine learning, game theory, or feedback based control, are a promising research topic that offers ample scope for future exploration.

6.1.2. Dumb Resources

The *dumb resources* approach, rather than modifying the resources themselves to be QoS- and application-aware, controls the resources and their allocation at different points in the system (e.g., at the cores/sources or at the system software) so that unfair slowdowns and performance degradation are mitigated. For instance, Ebrahimi et al. propose Fairness via Source Throttling (FST) [53, 55, 56], which throttles applications at the source (processor core) to regulate the number of requests that are sent to the shared caches and main memory from the processor core. Cheng et al. [32] propose to break down threads into compute and memory tasks and restrict the number of concurrent memory tasks. Kayiran et al. [85] throttle the thread-level parallelism of the GPU to mitigate memory contention related slowdowns in heterogeneous

architectures consisting of both CPUs and GPUs. Das et al. [40] propose to map applications to cores (by modifying the application scheduler in the operating system) in a manner that is aware of the applications' interconnect and memory access characteristics. Muralidhara et al. [131] propose to partition memory channels among applications such the data of applications that interfere significantly with each other are mapped to different memory channels. Other works [77, 88, 116, 194] build upon [131] and partition banks between applications at a fine-grained level to prevent different applications' request streams from interfering at the same banks. Zhuravlev et al. [206] and Tang et al. [182] propose to mitigate interference by co-scheduling threads that interact well and interfere less at the shared resources. Kaseridis et al. [84] propose to interleave data across channels and banks such that threads benefit from locality in the DRAM row-buffer, while not causing significant interference to each other.

On the interconnect side, there has been a growing body of work on *source throttling* mechanisms [12, 27, 61, 65, 147, 148, 184] that detect congestion or slowdowns within the network and throttle the injection rates of applications in an application-aware manner to maximize both system performance and fairness. In heterogeneous architectures consisting of CPUs and GPUs, similar *source throttling* approaches can greatly reduce memory congestion and improve both QoS provided to the CPUs and overall system performance [85]. Kayiran et al. [85] provide a promising mechanism that can be configured to achieve multiple different performance goals in such architectures by adapting the number of threads scheduled in the GPU based on memory congestion and latency tolerance characteristics in the heterogeneous system.

One common characteristic among all these *dumb resources* approaches is that they regulate the amount of contention at the caches, interconnects and main memory by controlling their operation/allocation from a different agent such as the cores, the operating system, or the memory allocator, while *not* modifying the resources themselves to be QoS- or application-aware. This has the advantage of keeping the resources themselves simple, while also potentially enabling better coordination of allocation decisions across multiple resources. On the other hand, the disadvantages of the *dumb resources* approach are that 1) each resource may not be best utilized to provide the highest performance because it cannot make interference-aware decisions, 2) monitoring and control mechanisms to understand and decide how to best control operation/allocation from a different agent than the resources themselves to achieve a particular goal increase complexity.

6.1.3. Integrated Approaches to QoS and Resource Management

Even though both the *smart resources* and *dumb resources* approaches can greatly mitigate interference and provide QoS in the memory system when employed individually, they each have advantages and disadvantages that are unique, as we discussed above. Therefore, integrating the *smart resources* and *dumb resources* approaches can enable better interference mitigation than employing either approach alone by exploiting the advantages of each approach. An early example of such an integrated resource management approach is the Integrated Memory Partitioning and Scheduling (IMPS) scheme [131], which combines memory channel partitioning in the operating system (a *dumb resource* approach) along with simple application-aware memory request scheduling in the memory controller (a *smart resource* approach), leading to higher performance than when either approach is employed alone. The key idea of IMPS is to partition memory channels to mitigate interference between memory-intensive applications while prioritizing compute-intensive applications in the memory scheduler. Subramanian et al. [131] show

that this combined technique improves performance more than either memory channel partitioning or application-aware memory scheduling alone. We believe such combined approaches will become even more important in the future as memory interference becomes an even more severe problem than today due to limited memory bandwidth and data-intensive workloads. Combining different approaches to memory QoS and resource management, both *smart* and *dumb*, with the goal of more effective interference mitigation is therefore a promising area for future research and exploration.

6.2. Quantifying and Controlling Interference

While several previous works have focused on mitigating interference at the different components of a memory system, with the goals of improving performance and preventing unfair application slowdowns, few previous works have focused on *precisely* quantifying and controlling the impact of interference on application slowdowns, with the goal of providing *soft or hard performance guarantees*. An application's susceptibility to interference and, consequently, its performance, depends on which other applications it is sharing resources with: an application can sometimes have very high performance and at other times very low performance on the same system, solely depending on its co-runners (as we have already discussed in Section 6 and shown an example in Figure 9). Therfore a critical research challenge is how to design the memory system (including all shared resources such as main memory, caches, and interconnects) so that 1) the performance of *each application* is predictable and controllable, and performance requirements of each application are satisfied, while 2) the performance and efficiency of the *entire system* are as high as needed or possible.

A promising solution direction to address this *predictable performance* challenge is to devise mechanisms that are effective and accurate at 1) estimating and predicting application performance in the presence of inter-application interference in a dynamic system with continuously incoming and outgoing applications, and 2) enforcing *end-to-end performance guarantees* within the entire shared memory system. We discuss briefly several prior works that took some of the first steps to achieve these goals and conclude with some promising future directions to pursue.

Stall Time Fair Memory Scheduling (STFM) [137] is one of the first works on estimating the impact of inter-application memory interference on application performance. STFM estimates the impact of memory interference at the main memory alone on application slowdowns. It does so by estimating the impact of delaying every individual request of an application on its slowdown. Fairness via Source Throttling (FST) [53, 56] and Per-Thread Cycle Accounting [49] employ a scheme similar to STFM to estimate the impact of main memory interference, while also taking into account interference at the shared caches. While these approaches are good starting points towards addressing the challenge of estimating and predicting application slowdowns in the presence of memory interference, our recent work [176] observed that the slowdown estimates from STFM, FST and Per-Thread Cycle Accounting are relatively inaccurate since they estimate interference at an *individual request granularity*. As a result, such schemes may not only be able to provide strict performance guarantees but also become complex due to the hardware needed to track interference on an individual request granularity.

We have recently designed a simple method, called MISE (Memory-interference Induced Slowdown Estimation) [176], for estimating application slowdowns accurately in the presence of main memory interference. We observe that an application's aggregate memory request service rate is a good proxy for its performance, as depicted in Figure 11, which shows the measured

performance versus memory request service rate for three applications on a real system [176]. As such, an application's slowdown can be accurately estimated by estimating its *uninterfered request service rate*, which can be done by prioritizing that application's requests in the memory system during some execution intervals. Results show that average error in slowdown estimation with this relatively simple technique is approximately 8% across a wide variety of workloads. Figure 12 shows the actual versus predicted slowdowns over time, for astar, a representative application from among the many applications examined, when it is run alongside three other applications on a simulated 4-core system. As we can see, MISE's slowdown estimates track the actual measured slowdown closely.

We believe these works on accurately estimating application slowdowns and providing predictable performance in the presence of memory interference have just scratched the surface of a critically important research direction. Designing predictable computing systems is a Grand Research Challenge, as identified by the Computing Research Association [37]. Many future ideas in this direction seem promising. We discuss some of them briefly. First, extending such simple performance estimation techniques like MISE [176] to the entire memory and storage system is a promising area of future research in both homogeneous and heterogeneous systems. Second, estimating and bounding memory slowdowns for hard real-time performance guarantees, as recently discussed by Kim et al. [88], is similarly promising. Third, devising memory devices, architectures and interfaces that can support better predictability and QoS also appears promising. Some key exciting research questions include, but are by no means limited to, the following: How sensitive are different applications to memory, interconnect and storage bandwidth? How does this sensitivity vary with the memory/storage/interconnect technology? How sensitive are applications to memory/cache/storage capacity? How do we take into account sensitivity to different resources such as cache/memory/storage capacity and bandwidth in estimating application slowdowns? How can we estimate slowdowns in heterogeneous systems consisting of CPUs, GPUs and hardware accelerators?

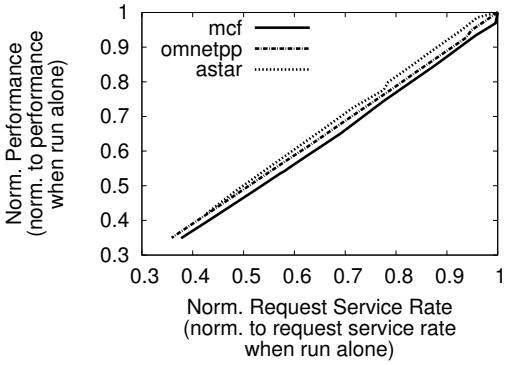


Figure 11. Request service rate vs. performance. Reproduced from [176]

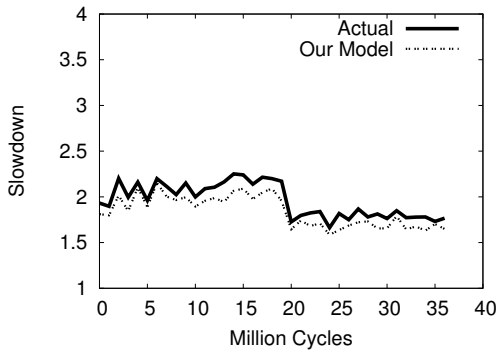


Figure 12. Actual vs. predicted slowdowns with MISE. Reproduced from [176]

Once accurate slowdown estimates are available, they can be leveraged in multiple possible ways. One possible use case is to leverage them in the hardware to allocate *just enough resources* to an application so that its performance requirements are met. We demonstrate such a scheme for memory bandwidth allocation showing that applications' performance/slowdown requirements can be effectively met by leveraging slowdown estimates from the MISE model [176]. There are several other ways in which slowdown estimates can be leveraged in both the hardware and the software to achieve various system-level goals. For instance, accurate slowdown

estimates can be used to drive fair pricing schemes based on slowdowns, rather than just resource allocation, in a cloud computing setting [1, 2]. Slowdown estimates can also be used to consolidate virtual machines onto physical hosts so that applications are not unfairly slowed down, through virtual machine migration and admission control schemes [68, 117, 182]. These and other potential schemes to leverage accurate slowdown estimates are promising directions to explore.

7. Flash Memory Scaling Challenges

We discuss briefly the challenges of scaling the major component of the storage subsystem, flash memory. Flash memory is arguably the most successful charge-based memory (like DRAM) that has been employed as part of the storage system. Its benefits over hard disks are clear: greatly lower latencies, greatly higher bandwidth and much higher reliability due to the lack of mechanical components. These have lead to the successful adoption of flash memory in modern systems across the board, to the point of replacing hard disk drives completely in space-constrained environments, e.g., laptop computers.

Our discussion in this section will be limited to some key challenges in improving the reliability and lifetime of flash memory, which we have been exploring in our recent research. Note that many other challenges exist, including but not limited to the following: 1) making the storage stack much higher performance to take advantage of the low latency and high parallelism of raw flash devices [189] (similarly to what we discussed in Section 5.3 with respect to the Persistent Memory Manager), 2) providing transactional support in the flash translation layer for better system performance and flexibility [120], 3) taking advantage of application- and system-level information to manage flash memory in a way that improves performance, efficiency, lifetime and cost. These are great research directions to explore, but, for brevity, we will not discuss them in further detail.

In part of our research, we aim to develop new techniques that overcome reliability and endurance challenges of flash memory to enable its scaling beyond the 20nm technology generations. To this end, we experimentally measure, characterize, analyze, and model error patterns that occur in existing flash chips, using an experimental flash memory testing and characterization platform [19]. Based on the understanding we develop from our experiments, we aim to develop error management techniques that mitigate the fundamental types of errors that are likely to increase as flash memory scales.

We have recently experimentally characterized complex flash errors that occur at 30-40nm flash technologies [20], categorizing them into four types: retention errors, program interference errors, read errors, and erase errors. Our characterization shows the relationship between various types of errors and demonstrates empirically using real 3x-nm flash chips that retention errors are the most dominant error type. Our results demonstrate that different flash errors have distinct patterns: retention errors and program interference errors are program/erase-(P/E)-cycle-dependent, memory-location-dependent, and data-value-dependent. Since the observed error patterns are due to fundamental circuit and device behavior inherent in flash memory, we expect our observations and error patterns to also hold in flash memories beyond 30-nm technology.

Based on our experimental characterization results that show that the retention errors are the most dominant errors, we have developed a suite of techniques to mitigate the effects of such errors, called Flash Correct-and-Refresh (FCR) [21]. The key idea is to periodically read each

page in flash memory, correct its errors using simple error correcting codes (ECC), and either remap (copy/move) the page to a different location or reprogram it in its original location by recharging the floating gates, before the page accumulates more errors than can be corrected with simple ECC. Our simulation experiments using real I/O workload traces from a variety of file system, database, and search applications show that FCR can provide 46x flash memory lifetime improvement at only 1.5% energy overhead, with no additional hardware cost.

We have also experimentally investigated and characterized the threshold voltage distribution of different logical states in MLC NAND flash memory [24]. We have developed new models that can predict the shifts in the threshold voltage distribution based on the number of P/E cycles endured by flash memory cells. Our data shows that the threshold voltage distribution of flash cells that store the same value can be approximated, with reasonable accuracy, as a Gaussian distribution. The threshold voltage distribution of flash cells that store the same value gets distorted as the number of P/E cycles increases, causing threshold voltages of cells storing different values to overlap with each other, which can lead to the incorrect reading of values of some cells as flash cells accumulate P/E cycles. We find that this distortion can be accurately modeled and predicted as an exponential function of the P/E cycles, with more than 95% accuracy. Such predictive models can aid the design of more sophisticated error correction methods, such as LDPC codes [62], which are likely needed for reliable operation of future flash memories.

We are currently investigating another increasingly significant obstacle to MLC NAND flash scaling, which is the increasing cell-to-cell program interference due to increasing parasitic capacitances between the cells' floating gates. Accurate characterization and modeling of this phenomenon are needed to find effective techniques to combat program interference. In recent work [23], we leverage the *read retry* mechanism found in some flash designs to obtain measured threshold voltage distributions from state-of-the-art 2Y-nm (i.e., 24-20 nm) MLC NAND flash chips. These results are then used to characterize the cell-to-cell program interference under various programming conditions. We show that program interference can be accurately modeled as additive noise following Gaussian-mixture distributions, which can be predicted with 96.8% accuracy using linear regression models. We use these models to develop and evaluate a read reference voltage prediction technique that reduces the raw flash bit error rate by 64% and increases the flash lifetime by 30%. More details can be found in Cai et al. [23].

To improve flash memory lifetime, we have developed a mechanism called Neighbor-Cell Assisted Correction (NAC) [25], which uses the value information of cells in a neighboring page to correct errors found on a page when reading. This mechanism takes advantage of the new empirical observation that identifying the value stored in the immediate-neighbor cell makes it easier to determine the data value stored in the cell that is being read. The key idea is to re-read a flash memory page that fails error correction codes (ECC) with the set of read reference voltage values corresponding to the conditional threshold voltage distribution assuming a neighbor cell value and use the re-read values to correct the cells that have neighbors with that value. Our simulations show that NAC effectively improves flash memory lifetime by 33% while having no (at nominal lifetime) or very modest (less than 5% at extended lifetime) performance overhead.

Most recently, we have provided a rigorous characterization of retention errors in NAND flash memory devices and new techniques to take advantage of this characterization to improve flash memory lifetime [26]. We have extensively characterized how the threshold voltage distribution of flash memory changes under different retention age, i.e., the length of time since a flash cell is programmed. We observe from our characterization results that 1) the optimal read reference

voltage of a flash cell, at which the data can be read with the lowest raw bit error rate (RBER), systematically changes with the cell's retention age and 2) different regions of flash memory can have different retention ages, and hence different optimal read reference voltages. Based on these observations, we propose a new technique to learn and apply the optimal read reference voltage online (called retention optimized reading). Our evaluations show that our technique can extend flash memory lifetime by 64% and reduce average error correction latency by 7% with only a 768 KB storage overhead in flash memory for a 512 GB flash-based SSD. We also propose a technique to recover data with uncorrectable errors by identifying and *probabilistically correcting* flash cells with retention errors. Our evaluation shows that this technique can effectively recover data from uncorrectable flash errors and reduce RBER by 50%. More detail can be found in Cai et al. [26].

These works, to our knowledge, are the first open-literature works that 1) characterize various aspects of real state-of-the-art flash memory chips, focusing on reliability and scaling challenges, and 2) exploit the insights developed from these characterizations to develop new mechanisms that can improve flash memory reliability and lifetime. We believe such an experimental- and characterization-based approach (which we also employ for DRAM [86, 97, 110, 115], as we discussed in Section 4) to developing novel techniques for both existing and emerging memory technologies is critically important as it 1) provides a solid basis (i.e., real data from modern devices) on which future analyses and new techniques can be based, 2) reveals new scaling trends in modern devices, pointing to important challenges in the field, and 3) equips the research community with reliable models and analyses that can be openly used to innovate in areas where experimental information is usually scarce in the open literature due to heavy competition within industry, thereby enhancing research and investigation in areas that are previously deemed to be difficult to research.

Going forward, we believe more accurate and detailed characterization of flash memory error mechanisms is needed to devise models that can aid the design of even more efficient and effective mechanisms to tolerate errors found in sub-20nm flash memories. A promising direction is the design of predictive models that the system (e.g., the flash controller or system software) can use to proactively estimate the occurrence of errors and take action to prevent the error before it happens. Flash-correct-and-refresh [21], read reference voltage prediction [23], and retention optimized reading [26] mechanisms, described earlier, are early forms of such predictive error tolerance mechanisms. Methods for exploiting application and memory access characteristics to optimize flash performance, energy, lifetime and cost are also very promising to explore. We believe there is a bright future ahead for more aggressive and effective application- and data-characteristic-aware management mechanisms for flash memory (just like for DRAM and emerging memory technologies). Such techniques will likely aid effective scaling of flash memory technology into the future.

8. Conclusion

We have described several research directions and ideas to enhance memory scaling via system and architecture-level approaches. We believe there are three key *fundamental principles* that are essential for memory scaling: 1) better cooperation between devices, system, and software, i.e., the efficient exposure of richer information up and down the layers of the system stack with the development of more flexible yet abstract interfaces that can scale well into the future , 2) better-than-worst-case design, i.e., design of the memory system such that it is optimized for

the common case instead of the worst case, 3) heterogeneity in design, i.e., the use of heterogeneity at all levels in memory system design to enable the optimization of multiple metrics at the same time. We believe these three principles are related and sometimes coupled. For example, to exploit heterogeneity in the memory system, we may need to enable better cooperation between the devices and the system, e.g., as in the case of heterogeneous DRAM refresh rates [114], tiered-latency DRAM [109], heterogeneous-reliability memory [122], locality-aware management of hybrid memory systems [201] and the persistent memory manager for a heterogeneous array of memory/storage devices [127], five of the many ideas we have discussed in this paper.

We have shown that a promising approach to designing scalable memory systems is the co-design of memory and other system components to enable better system optimization. Enabling better cooperation across multiple levels of the computing stack, including software, microarchitecture, and devices can help scale the memory system by exposing more of the memory device characteristics to higher levels of the system stack such that the latter can tolerate and exploit such characteristics. Finally, heterogeneity in the design of the memory system can help overcome the memory scaling challenges at the device level by enabling better specialization of the memory system and its dynamic adaptation to different demands of various applications. We believe such system-level and integrated approaches will become increasingly important and effective as the underlying memory technology nears its scaling limits at the physical level and envision a near future full of innovation in main memory architecture, enabled by the co-design of the system and main memory.

Acknowledgments

The source code and data sets of some of the works we have discussed or alluded to (e.g., [9, 86, 97, 111, 153, 166, 196]) are available under open source software license at our research group, SAFARI's, tools website [3].

We would like to thank Rachata Ausavarungnirun for logistic help in preparing this article and earlier versions of it. Many thanks to all students in the SAFARI research group and collaborators at Carnegie Mellon as well as other universities, whom have contributed to the various works outlined in this article. Thanks also go to our research group's industrial sponsors over the past six years, including AMD, HP Labs, Huawei, IBM, Intel, Microsoft, Nvidia, Oracle, Qualcomm, Samsung. Some of the research reported here was also partially supported by GSRC, Intel URO Memory Hierarchy Program, Intel Science and Technology Center on Cloud Computing, NIH, NSF, and SRC. In particular, NSF grants 0953246, 1065112, 1147397, 1212962, 1320531, 1409723, 1423172 have supported parts of the research we have described.

This article is a significantly extended and revised version of an invited paper that appeared at the 5th International Memory Workshop [133], which was also presented at MemCon 2013 [134]. Part of the structure of this article is based on an evolving set of talks Onur Mutlu has delivered at various venues on *Scaling the Memory System in the Many-Core Era* and *Rethinking Memory System Design for Data-Intensive Computing* between 2010-2015, including invited talks at the 2011 International Symposium on Memory Management and ACM SIGPLAN Workshop on Memory System Performance and Correctness [135] and the 2012 DAC More Than Moore Technologies Workshop. Section 7 of this article is a significantly extended and revised version of the introduction of an invited article that appeared in a special issue of the Intel Technology Journal, titled *Error Analysis and Retention-Aware Error Management for NAND Flash Memory* [22].

References

1. *Amazon EC2*, http://aws.amazon.com/ec2/pricing/.
2. *Microsoft Azure*, http://azure.microsoft.com/en-us/pricing/details/virtual-machines/.
3. "SAFARI tools," https://www.ece.cmu.edu/~safari/tools.html.
4. "International technology roadmap for semiconductors (ITRS)," 2011.
5. *Hybrid Memory Consortium*, 2012, http://www.hybridmemorycube.org.
6. *Top 500*, 2013, http://www.top500.org/featured/systems/tianhe-2/.
7. J.-H. Ahn *et al.*, "Adaptive self refresh scheme for battery operated high-density mobile DRAM applications," in *ASSCC*, 2006.
8. A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *ISCA*, 2004.
9. C. Alkan *et al.*, "Personalized copy-number and segmental duplication maps using next-generation sequencing," in *Nature Genetics*, 2009.
10. G. Atwood, "Current and emerging memory technology landscape," in *Flash Memory Summit*, 2011.
11. R. Ausavarungnirun *et al.*, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012.
12. R. Ausavarungnirun *et al.*, "Design and evaluation of hierarchical rings with deflection routing," in *SBAC-PAD*, 2014.
13. S. Balakrishnan and G. S. Sohi, "Exploiting value locality in physical register files," in *MICRO*, 2003.
14. A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *ISCA*, 2009.
15. R. Bitirgen *et al.*, "Coordinated management of multiple interacting resources in CMPs: A machine learning approach," in *MICRO*, 2008.
16. B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, 1970.
17. R. Bryant, "Data-intensive supercomputing: The case for DISC," *CMU CS Tech. Report 07-128*, 2007.
18. Q. Cai *et al.*, "Meeting points: Using thread criticality to adapt multicore hardware to parallel regions," in *PACT*, 2008.
19. Y. Cai *et al.*, "FPGA-based solid-state drive prototyping platform," in *FCCM*, 2011.
20. Y. Cai *et al.*, "Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis," in *DATE*, 2012.
21. Y. Cai *et al.*, "Flash Correct-and-Refresh: Retention-aware error management for increased flash memory lifetime," in *ICCD*, 2012.

22. Y. Cai *et al.*, "Error analysis and retention-aware error management for NAND flash memory," *Intel technology Journal*, vol. 17, no. 1, May 2013.
23. Y. Cai *et al.*, "Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation," in *ICCD*, 2013.
24. Y. Cai *et al.*, "Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis and modeling," in *DATE*, 2013.
25. Y. Cai *et al.*, "Neighbor-cell assisted error correction for MLC NAND flash memories," in *SIGMETRICS*, 2014.
26. Y. Cai *et al.*, "Data retention in MLC NAND flash memory: Characterization, optimization and recovery," in *HPCA*, 2015.
27. K. Chang *et al.*, "HAT: Heterogeneous adaptive throttling for on-chip networks," in *SBAC-PAD*, 2012.
28. K. Chang *et al.*, "Improving DRAM performance by parallelizing refreshes with accesses," in *HPCA*, 2014.
29. N. Chatterjee *et al.*, "Leveraging heterogeneity in DRAM main memories to accelerate critical word access," in *MICRO*, 2012.
30. S. Chaudhry *et al.*, "High-performance throughput computing," *IEEE Micro*, vol. 25, no. 6, 2005.
31. E. Chen *et al.*, "Advances and future prospects of spin-transfer torque random access memory," *IEEE Transactions on Magnetics*, vol. 46, no. 6, 2010.
32. H.-Y. Cheng *et al.*, "Memory latency reduction via thread throttling," in *MICRO*, 2010.
33. S. Chhabra and Y. Solihin, "i-NVMM: a secure non-volatile main memory system with incremental encryption," in *ISCA*, 2011.
34. Y. Chou *et al.*, "Store memory-level parallelism optimizations for commercial applications," in *MICRO*, 2005.
35. Y. Chou *et al.*, "Microarchitecture optimizations for exploiting memory-level parallelism," in *ISCA*, 2004.
36. E. Chung *et al.*, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs?" in *MICRO*, 2010.
37. *Grand Research Challenges in Information Systems*, Computing Research Association, http://www.cra.org/reports/gc.systems.pdf.
38. J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.
39. K. V. Craeynest *et al.*, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ISCA*, 2012.
40. R. Das *et al.*, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *HPCA*, 2013.
41. R. Das *et al.*, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.
42. R. Das *et al.*, "Aergia: Exploiting packet latency slack in on-chip networks," in *ISCA*, 2010.
43. R. Das *et al.*, "Aergia: A network-on-chip exploiting packet latency slack," *IEEE Micro (TOP PICKS Issue)*, vol. 31, no. 1, 2011.

44. H. David *et al.*, "Memory power management via dynamic voltage/frequency scaling," in *ICAC*, 2011.
45. J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, 2013.
46. Q. Deng *et al.*, "MemScale: active low-power modes for main memory," in *ASPLOS*, 2011.
47. G. Dhiman *et al.*, "PDRAM: A hybrid PRAM and DRAM main memory system," in *DAC*, 2009.
48. X. Dong *et al.*, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *SC*, 2009.
49. K. Du Bois *et al.*, "Per-thread cycle accounting in multicore processors," *TACO*, 2013.
50. K. Du Bois *et al.*, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *ISCA*, 2013.
51. J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *ICS*, 1997.
52. J. Dusser *et al.*, "Zero-content augmented caches," in *ICS*, 2009.
53. E. Ebrahimi *et al.*, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *ASPLOS*, 2010.
54. E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *MICRO*, 2011.
55. E. Ebrahimi *et al.*, "Prefetch-aware shared-resource management for multi-core systems," in *ISCA*, 2011.
56. E. Ebrahimi *et al.*, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," *TOCS*, 2012.
57. E. Ebrahimi *et al.*, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *HPCA*, 2009.
58. E. Ebrahimi *et al.*, "Coordinated control of multiple prefetchers in multi-core systems," in *MICRO*, 2009.
59. M. Ekman, "A robust main-memory compression scheme," in *ISCA*, 2005.
60. S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *ISCA*, 2010.
61. C. Fallin *et al.*, "CHIPPER: a low-complexity bufferless deflection router," in *HPCA*, 2011.
62. R. Gallager, "Low density parity check codes," 1963, MIT Press.
63. A. Glew, "MLP yes! ILP no!" in *ASPLOS Wild and Crazy Idea Session*, Oct. 1998.
64. B. Grot *et al.*, "Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees," in *ISCA*, 2011.
65. B. Grot *et al.*, "Regional congestion awareness for load balance in networks-on-chip," in *HPCA*, 2008.
66. B. Grot *et al.*, "Preemptive virtual clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip," in *MICRO*, 2009.
67. B. Grot *et al.*, "Topology-aware quality-of-service support in highly integrated chip multiprocessors," in *WIOSCA*, 2010.

68. A. Gulati *et al.*, "VMware distributed resource management: Design, implementation, and lessons learned," *VMware Technical Journal*, 2012.
69. G. Hinton *et al.*, "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, Feb. 2001, Q1 2001 Issue.
70. S. Hong, "Memory technology trend and future challenges," in *IEDM*, 2010.
71. E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
72. C. Isen and L. K. John, "Eskimo: Energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *MICRO*, 2009.
73. R. Iyer, "CQoS: a framework for enabling QoS in shared caches of CMP platforms," in *ICS*, 2004.
74. R. Iyer *et al.*, "QoS policies and architecture for cache/memory in CMP platforms," in *SIGMETRICS*, 2007.
75. A. Jaleel *et al.*, "Adaptive insertion policies for managing shared caches," in *PACT*, 2008.
76. A. Jaleel *et al.*, "High performance cache replacement using re-reference interval prediction," in *ISCA*, 2010.
77. M. K. Jeong *et al.*, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *HPCA*, 2012.
78. J. A. Joao *et al.*, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*, 2012.
79. J. A. Joao *et al.*, "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *ISCA*, 2013.
80. A. Jog *et al.*, "Orchestrated scheduling and prefetching for GPGPUs," in *ISCA*, 2013.
81. A. Jog *et al.*, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," in *ASPLOS*, 2013.
82. T. L. Johnson *et al.*, "Run-time spatial locality detection and optimization," in *MICRO*, 1997.
83. U. Kang *et al.*, "Co-architecting controllers and DRAM to enhance DRAM process scaling," in *The Memory Forum*, 2014.
84. D. Kaseridis *et al.*, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *MICRO*, 2011.
85. O. Kayiran *et al.*, "Managing GPU concurrency in heterogeneous architectures," in *MICRO*, 2014.
86. S. Khan *et al.*, "The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study," in *SIGMETRICS*, 2014.
87. S. Khan *et al.*, "Improving cache performance by exploiting read-write disparity," in *HPCA*, 2014.
88. H. Kim *et al.*, "Bounding memory interference delay in COTS-based multi-core systems," in *RTAS*, 2014.
89. J. Kim and M. C. Papaefthymiou, "Dynamic memory design for low data-retention power," in *PATMOS*, 2000.

90. K. Kim, "Future memory technology: challenges and opportunities," in *VLSI-TSA*, 2008.
91. K. Kim *et al.*, "A new investigation of data retention time in truly nanoscaled DRAMs." *IEEE Electron Device Letters*, vol. 30, no. 8, Aug. 2009.
92. Y. Kim, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Talk at MICRO*, 2010.
93. Y. Kim *et al.*, "ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.
94. Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.
95. Y. Kim *et al.*, "Thread cluster memory scheduling," *IEEE Micro (TOP PICKS Issue)*, vol. 31, no. 1, 2011.
96. Y. Kim *et al.*, "A case for subarray-level parallelism (SALP) in DRAM," in *ISCA*, 2012.
97. Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.
98. Y. Koh, "NAND flash scaling beyond 20nm," in *IMW*, 2009.
99. D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *ISCA*, 1981.
100. E. Kultursay *et al.*, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
101. S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *ISCA*, 1998.
102. B. C. Lee *et al.*, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.
103. B. C. Lee *et al.*, "Phase change memory architecture and the quest for scalability," *Communications of the ACM*, vol. 53, no. 7, 2010.
104. B. C. Lee *et al.*, "Phase change technology and the future of main memory," *IEEE Micro (TOP PICKS Issue)*, vol. 30, no. 1, 2010.
105. C. J. Lee *et al.*, "Prefetch-aware DRAM controllers," in *MICRO*, 2008.
106. C. J. Lee *et al.*, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," HPS, UT-Austin, Tech. Rep. TR-HPS-2010-002, 2010.
107. C. J. Lee *et al.*, "Prefetch-aware memory controllers," *TC*, vol. 60, no. 10, 2011.
108. C. J. Lee *et al.*, "Improving memory bank-level parallelism in the presence of prefetching," in *MICRO*, 2009.
109. D. Lee *et al.*, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *HPCA*, 2013.
110. D. Lee *et al.*, "Adaptive-latency DRAM: Optimizing DRAM timing for the common-case," in *HPCA*, 2015.
111. D. Lee *et al.*, "Fast and accurate mapping of complete genomics reads," in *Methods*, 2014.
112. C. Lefurgy *et al.*, "Energy management for commercial servers," in *IEEE Computer*, 2003.
113. K. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers," in *ISCA*, 2009.
114. J. Liu *et al.*, "RAIDR: Retention-aware intelligent DRAM refresh," in *ISCA*, 2012.

115. J. Liu *et al.*, "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms," in *ISCA*, 2013.

116. L. Liu *et al.*, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012.

117. M. Liu and T. Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," in *ISCA*, 2014.

118. S. Liu *et al.*, "Flikker: saving DRAM refresh-power through critical data partitioning," in *ASPLOS*, 2011.

119. G. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA*, 2008.

120. Y. Lu *et al.*, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *ICCD*, 2013.

121. Y. Lu *et al.*, "Loose-ordering consistency for persistent memory," in *ICCD*, 2014.

122. Y. Luo *et al.*, "Characterizing application memory error vulnerability to optimize data center cost via heterogeneous-reliability memory," in *DSN*, 2014.

123. A. Maislos *et al.*, "A new era in embedded flash memory," in *FMS*, 2011.

124. J. Mandelman *et al.*, "Challenges and future directions for the scaling of dynamic random-access memory (DRAM)," in *IBM JR&D*, vol. 46, 2002.

125. J. Meza *et al.*, "A case for small row buffers in non-volatile main memories," in *ICCD*, 2012.

126. J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *IEEE CAL*, 2012.

127. J. Meza *et al.*, "A case for efficient hardware-software cooperative management of storage and memory," in *WEED*, 2013.

128. A. Mishra *et al.*, "A heterogeneous multiple network-on-chip design: An application-aware approach," in *DAC*, 2013.

129. T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.

130. T. Moscibroda and O. Mutlu, "Distributed order scheduling and its application to multi-core DRAM controllers," in *PODC*, 2008.

131. S. Muralidhara *et al.*, "Reducing memory interference in multi-core systems via application-aware memory channel partitioning," in *MICRO*, 2011.

132. O. Mutlu, "Asymmetry everywhere (with automatic resource management)," in *CRA Workshop on Advanced Computer Architecture Research*, 2010.

133. O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.

134. O. Mutlu, "Memory scaling: A systems architecture perspective," in *MemCon*, 2013.

135. O. Mutlu *et al.*, "Memory systems in the many-core era: Challenges, opportunities, and solution directions," in *ISMM*, 2011, http://users.ece.cmu.edu/~omutlu/pub/onur-ismm-mspc-keynote-june-5-2011-short.pptx.

136. O. Mutlu *et al.*, "Address-value delta (AVD) prediction: A hardware technique for efficiently parallelizing dependent cache misses," *IEEE Transactions on Computers*, vol. 55, no. 12, Dec. 2006.

137. O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

138. O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.

139. O. Mutlu *et al.*, "Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *MICRO*, 2005.

140. O. Mutlu *et al.*, "Techniques for efficient processing in runahead execution engines," in *ISCA*, 2005.

141. O. Mutlu *et al.*, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro (TOP PICKS Issue)*, vol. 26, no. 1, 2006.

142. O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enabling high-performance and fair memory controllers," *IEEE Micro (TOP PICKS Issue)*, vol. 29, no. 1, 2009.

143. O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003.

144. O. Mutlu *et al.*, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro (TOP PICKS Issue)*, vol. 23, no. 6, 2003.

145. P. J. Nair *et al.*, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *ISCA*, 2013.

146. V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *MICRO*, 2011.

147. G. Nychis *et al.*, "Next generation on-chip networks: What kind of congestion control do we need?" in *HotNets*, 2010.

148. G. Nychis *et al.*, "On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects," in *SIGCOMM*, 2012.

149. T. Ohsawa *et al.*, "Optimizing the DRAM refresh count for merged DRAM/logic LSIs," in *ISLPED*, 1998.

150. V. S. Pai and S. Adve, "Code transformations to improve memory parallelism," in *MICRO*, 1999.

151. Y. N. Patt *et al.*, "HPS, a new microarchitecture: Rationale and introduction," in *MICRO*, 1985.

152. Y. N. Patt *et al.*, "Critical issues regarding HPS, a high performance microarchitecture," in *MICRO*, 1985.

153. G. Pekhimenko *et al.*, "Base-delta-immediate compression: A practical data compression mechanism for on-chip caches," in *PACT*, 2012.

154. G. Pekhimenko *et al.*, "Linearly compressed pages: A main memory compression framework with low complexity and low latency." in *MICRO*, 2013.

155. G. Pekhimenko *et al.*, "Exploiting compressed block size as an indicator of future reuse," in *HPCA*, 2015.

156. S. Phadke and S. Narayanasamy, "MLP aware heterogeneous memory system," in *DATE*, 2011.

157. M. K. Qureshi *et al.*, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *HPCA*, 2007.

158. M. K. Qureshi *et al.*, "Enhancing lifetime and security of phase change memories via start-gap wear leveling." in *MICRO*, 2009.

159. M. K. Qureshi *et al.*, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009.

160. M. K. Qureshi *et al.*, "A case for MLP-aware cache replacement," in *ISCA*, 2006.

161. M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006.

162. L. E. Ramos *et al.*, "Page placement in hybrid memory systems," in *ICS*, 2011.

163. S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM JR&D*, vol. 52, Jul/Sep 2008.

164. B. Schroder *et al.*, "DRAM errors in the wild: A large-scale field study," in *SIGMETRICS*, 2009.

165. N. H. Seong *et al.*, "Tri-level-cell phase change memory: Toward an efficient and reliable memory system," in *ISCA*, 2013.

166. V. Seshadri *et al.*, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *PACT*, 2012.

167. V. Seshadri *et al.*, "RowClone: Fast and efficient In-DRAM copy and initialization of bulk data," in *MICRO*, 2013.

168. V. Seshadri *et al.*, "The dirty-block index," in *ISCA*, 2014.

169. V. Seshadri *et al.*, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *TACO*, 2014.

170. F. Soltis, "Inside the AS/400," *29th Street Press*, 1996.

171. N. H. Song *et al.*, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *ISCA*, 2010.

172. V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC*, 2012.

173. V. Sridharan *et al.*, "Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *SC*, 2013.

174. S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA*, 2007.

175. J. Stuecheli *et al.*, "The virtual write queue: Coordinating DRAM and last-level cache policies," in *ISCA*, 2010.

176. L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.

177. L. Subramanian *et al.*, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.

178. M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009.

179. M. A. Suleman *et al.*, "Data marshaling for multi-core architectures," in *ISCA*, 2010.

180. M. A. Suleman *et al.*, "Data marshaling for multi-core systems," *IEEE Micro (TOP PICKS Issue)*, vol. 31, no. 1, 2011.

181. M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," *IEEE Micro (TOP PICKS Issue)*, vol. 30, no. 1, 2010.

182. L. Tang *et al.*, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011.

183. J. Tendler *et al.*, "POWER4 system microarchitecture," *IBM JRD*, Oct. 2001.

184. M. Thottethodi *et al.*, "Exploiting global knowledge to achieve self-tuned congestion control for k-ary n-cube networks," *IEEE TPDS*, vol. 15, no. 3, 2004.

185. R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM JR&D*, vol. 11, Jan. 1967.

186. T. Treangen and S. Salzberg, "Repetitive DNA and next-generation sequencing: computational challenges and solutions," in *Nature Reviews Genetics*, 2012.

187. A. Udipi *et al.*, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *ISCA*, 2010.

188. A. Udipi *et al.*, "Combining memory and a controller with photonics through 3D-stacking to enable scalable and energy-efficient systems," in *ISCA*, 2011.

189. V. Vasudevan *et al.*, "Using vector interfaces to deliver millions of IOPS from a networked key-value storage server," in *SoCC*, 2012.

190. R. K. Venkatesan *et al.*, "Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *HPCA*, 2006.

191. X. Wang and J. Martinez, "XChange: Scalable dynamic multi-resource allocation in multicore architectures," in *HPCA*, 2015.

192. H.-S. P. Wong *et al.*, "Phase change memory," in *Proceedings of the IEEE*, 2010.

193. H.-S. P. Wong *et al.*, "Metal-oxide RRAM," in *Proceedings of the IEEE*, 2012.

194. M. Xie *et al.*, "Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning," in *HPCA*, 2014.

195. Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *ISCA*, 2009.

196. H. Xin *et al.*, "Accelerating read mapping with FastHASH," in *BMC Genomics*, 2013.

197. H. Xin *et al.*, "Shifted Hamming Distance: A Fast and Accurate SIMD-friendly Filter to Accelerate Alignment Verification in Read Mapping," in *Bioinformatics*, 2015.

198. J. Yang *et al.*, "Frequent value compression in data caches," in *MICRO*, 2000.

199. D. Yoon *et al.*, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," in *ISCA*, 2011.

200. D. Yoon *et al.*, "The dynamic granularity memory system," in *ISCA*, 2012.

201. H. Yoon *et al.*, "Row buffer locality aware caching policies for hybrid memories," in *ICCD*, 2012.

202. H. Yoon *et al.*, "Data mapping and buffering in multi-level cell memory for higher performance and energy efficiency." *CMU SAFARI Tech. Report*, 2013.

203. H. Yoon *et al.*, "Efficient data mapping and buffering techniques for multi-level cell phase-change memories," *TACO*, 2014.

204. J. Zhao *et al.*, "FIRM: Fair and high-performance memory control for persistent memory systems," in *MICRO*, 2014.

205. H. Zhou and T. M. Conte, "Enhancing memory level parallelism via recovery-free value prediction," in *ICS*, 2003.

206. S. Zhuravlev *et al.*, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, 2010.