Processing Data Where It Makes Sense: Enabling In-Memory Computation

Onur Mutlu[a,b], Saugata Ghose[b], Juan Gómez-Luna[a], Rachata Ausavarungnirun[b,c]

[a] *ETH Zürich*
[b] *Carnegie Mellon University*
[c] *King Mongkut's University of Technology North Bangkok*

Abstract

Today's systems are overwhelmingly designed to move data to computation. This design choice goes directly against at least three key trends in systems that cause performance, scalability and energy bottlenecks: (1) data access from memory is already a key bottleneck as applications become more data-intensive and memory bandwidth and energy do not scale well, (2) energy consumption is a key constraint in especially mobile and server systems, (3) data movement is very expensive in terms of bandwidth, energy and latency, much more so than computation. These trends are especially severely-felt in the data-intensive server and energy-constrained mobile systems of today.

At the same time, conventional memory technology is facing many scaling challenges in terms of reliability, energy, and performance. As a result, memory system architects are open to organizing memory in different ways and making it more intelligent, at the expense of higher cost. The emergence of 3D-stacked memory plus logic as well as the adoption of error correcting codes inside DRAM chips, and the necessity for designing new solutions to serious reliability and security issues, such as the RowHammer phenomenon, are an evidence of this trend.

In this work, we discuss some recent research that aims to practically enable computation close to data. After motivating trends in applications as well as technology, we discuss at least two promising directions for *processing-in-memory* (PIM): (1) performing massively-parallel bulk operations in memory by exploiting the analog operational properties of DRAM, with low-cost changes, (2) exploiting the logic layer in 3D-stacked memory technology to accelerate important data-intensive applications. In both approaches, we describe and tackle relevant cross-layer research, design, and adoption challenges in devices, architecture, systems, and programming models. Our focus is on the development of in-memory processing designs that can be adopted in real computing platforms at low cost.

Keywords: data movement, main memory, processing-in-memory, 3D-stacked memory, near-data processing

1. Introduction

Main memory, which is built using the Dynamic Random Access Memory (DRAM) technology, is a major component in nearly all computing systems. Across all of these systems, including servers, cloud platforms, and mobile/embedded devices, the data working set sizes of modern applications are rapidly growing, causing the main memory to be a significant bottleneck for these applications [1, 2, 3, 4, 5, 6, 7]. Alleviating the main memory bottleneck requires the memory capacity, energy, cost, and performance to all scale in an efficient manner. Unfortunately, it has become increasingly difficult in recent years to scale all of these dimensions [1, 2, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31], and the main memory bottleneck has instead been worsening.

A major reason for the main memory bottleneck is the high cost associated with *data movement*. In today's computers, to perform any operation on data that resides in main memory, the memory controller must first issue a series of commands to the DRAM modules across an off-chip bus (known as the *memory channel*). The DRAM module responds by sending the data to the memory controller across the memory channel, after which the data is placed within a cache or registers. The CPU can only perform the operation on the data once the data is in the cache. This process of moving data from the DRAM to the CPU incurs a long latency, and consumes a significant amount of energy [7, 32, 33, 34, 35]. These costs are often exacerbated by the fact that much of the data brought into the caches is *not reused* by the CPU [36, 37], providing little benefit in return for the high latency and energy cost.

The cost of data movement is a fundamental issue with the *processor-centric* nature of contemporary computer systems, where the CPU is considered to be the master of the system and has been optimized heavily. In contrast, data storage units such as main memory are treated as unintelligent workers, and, thus, are largely not optimized. With the increasingly *data-centric* nature of contemporary and emerging applications, the processor-centric design approach leads to many inefficiencies. For example, within a single compute node, most of the node real estate is dedicated to handle the storage and movement of data (e.g., large on-chip caches, shared interconnect, memory controllers, off-chip interconnects, main memory) [38].

Recent advances in memory design and memory architecture have enabled the opportunity for a paradigm shift towards performing *processing-in-memory* (PIM), where we can redesign the computer to no longer be processor-centric and avoid unnecessary data movement. Processing-in-memory, also known as *near-data processing* (NDP), enables the ability to perform operations either using (1) the memory itself, or (2) some form of processing logic (e.g., accelerators, simple cores, reconfigurable logic) inside the DRAM subsystem. Processing-in-memory has been proposed for at least four decades [39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53]. However, these past efforts were *not* adopted at large scale due to various reasons, including the difficulty of integrating processing elements with DRAM and the fact that memory technology was not facing as critical scaling challenges as it is today. As a result of advances in modern memory architectures, e.g., the integration of logic and memory in a 3D-stacked manner, various recent works explore a range of PIM architectures for multiple different purposes (e.g., [7, 32, 33, 34, 35, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91]).

In this paper, we explore two approaches to enabling processing-in-memory in modern systems. The first approach examines a form of PIM that only *minimally changes memory chips* to perform simple yet powerful common operations that the chip could be made inherently very good at performing [31, 71, 82, 83, 84, 85, 86, 90, 92, 93, 94, 95, 96]. Solutions that fall under this approach take advantage of the existing DRAM design to cleverly and efficiently perform *bulk operations* (i.e., operations on an entire row of DRAM cells), such as bulk copy, data initialization, and bitwise operations. The second approach takes advantage of the design of emerging *3D-stacked memory technologies* to enable PIM in a more general-purpose manner [7, 34, 35, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 70, 72, 73, 74, 75, 77, 87, 88, 89, 91]. In order to stack multiple layers of memory, 3D-stacked chips use vertical *through-silicon vias* (TSVs) to connect the layers to each other, and to the I/O drivers of the chip [97]. The TSVs provide much greater *internal* bandwidth than is available externally on the memory channel. Several such 3D-stacked memory architectures, such as the Hybrid Memory Cube [98, 99] and High-Bandwidth Memory [97, 100], include a *logic layer*, where designers can add some simple processing logic to take advantage of the high internal bandwidth.

For both approaches to PIM, there are a number of new challenges that system architects and programmers must address to enable the widespread adoption of PIM across the computing landscape and in different domains of workloads. In addition to describing work along the two key approaches, we also discuss these challenges in this paper, along with existing work that addresses these challenges.

2. Major Trends Affecting Main Memory

The main memory is a major, critical component of all computing systems, including cloud and server platforms, desktop computers, mobile and embedded devices, and sensors. It is one of the two main pillars of any computing platform, together with the processing elements, namely CPU cores, GPU cores, or reconfigurable devices.

Due to its relatively low cost and low latency, DRAM is the predominant technology to build main memory. Because of the growing data working set sizes of modern applications [1, 2, 3, 4, 5, 6, 7], there is an ever-increasing demand for higher DRAM capacity and performance. Unfortunately, DRAM technology scaling is becoming more and more challenging in terms of increasing the DRAM capacity and maintaining the DRAM energy efficiency and reliability [1, 11, 15, 101, 102]. Thus, fulfilling the increasing memory needs from applications is becoming more and more costly and difficult [2, 3, 4, 8, 9, 10, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 34, 35, 59, 103, 104, 105].

If CMOS technology scaling is coming to an end [106], the projections are significantly worse for DRAM technology scaling [107]. DRAM technology scaling affects all major characteristics of DRAM, including capacity, bandwidth, latency, energy and cost. We next describe the key issues and trends in DRAM technology scaling and discuss how these trends motivate the need for intelligent memory controllers, which can be used as a substrate for processing in memory.

The first key concern is the difficulty of scaling DRAM capacity (i.e., density, or cost per bit), bandwidth and latency *at the same time*. While the processing core count doubles every two years, the DRAM capacity doubles only every three years [20]. This causes the *memory capacity per core* to drop by approximately 30% every two years [20]. The trend is even worse for *memory bandwidth per core* – in the last 20 years, DRAM chip capacity (for the most common DDRx chip of the time) has improved around 128× while DRAM bandwidth has increased only around 20× [22, 23, 31]. In the same period of twenty years, DRAM latency (as measured by the row cycling time) has remained almost constant (i.e., reduced by only 30%), making it a significant performance bottleneck for many modern workloads, including in-memory databases [108, 109, 110, 111], graph processing [34, 112, 113], data analytics [110, 114, 115, 116], datacenter workloads [4], and consumer workloads [7]. As low-latency computing is becoming ever more important [1], e.g., due to the ever-increasing need to process large amounts of data at real time, and predictable performance continues to be a critical concern in the design of modern computing systems [2, 16, 117, 118, 119, 120, 121, 122, 123], it is increasingly critical to design low-latency main memory chips.

The second key concern is that DRAM technology scaling to smaller nodes adversely affects DRAM reliability. A DRAM cell stores each bit in the form of charge in a capacitor, which is accessed via an access transistor and peripheral circuitry. For a DRAM cell to operate correctly, both the capacitor and the access transistor (as well as the peripheral circuitry) need to operate reliably. As the size of the DRAM cell reduces, both the capacitor and the access transistor become less reliable. As a result, reducing the size of the DRAM cell increases the difficulty of correctly storing and detecting the desired original value in the DRAM cell [1, 11, 15, 101]. Hence, memory scaling causes memory errors to appear more frequently. For example, a study of Facebook's entire production datacenter servers showed that memory errors, and thus the server failure rate, increase proportionally with the density of the chips employed in the servers [124]. Thus, it is critical to make the main memory system more reliable to build reliable computing systems on top of it.

The third key issue is that the reliability problems caused by aggressive DRAM technology scaling can leads to new security vulnerabilities. The RowHammer phenomenon [11, 15] shows that it is possible to predictably induce errors (bit flips) in most modern DRAM chips. Repeatedly reading the same row in DRAM can corrupt data in physically-adjacent rows. Specifically, when a DRAM row is opened (i.e., activated) and closed (i.e., precharged) repeatedly (i.e., hammered), enough times within a DRAM refresh interval, one or more bits in physically-adjacent DRAM rows can be flipped to the wrong value. A very simple user-level program [125] can reliably and consistently induce RowHammer errors in vulnerable DRAM modules. The seminal paper that introduced RowHammer [11] showed that more than 85% of the chips tested, built by three major vendors between 2010 and 2014, were vulnerable to RowHammer-induced errors. In particular, *all* DRAM modules from 2012 and 2013 are vulnerable.

The RowHammer phenomenon entails a real reliability, and perhaps even more importantly, a real and prevalent security issue. It breaks physical memory isolation between two addresses, one of the fundamental building blocks of memory, on top of which system security principles are built. With RowHammer, accesses to one row (e.g., an application page) can modify data stored in another memory row (e.g., an OS page). This was confirmed by researchers from Google Project Zero, who developed a user-level attack that uses RowHammer to gain kernel privileges [126, 127]. Other researchers have shown how RowHammer vulnerabilities can be exploited in various ways to gain privileged access to various systems: in a remote server RowHammer can be used to remotely take over the server via the use of JavaScript [128]; a virtual machine can take over another virtual machine by inducing errors in the victim virtual machine's memory space [129]; a malicious application without permissions can take control of an Android mobile device [130]; or an attacker can gain arbitrary read/write access in a web browser on a Microsoft Windows 10 system [131]. For a more detailed treatment of the RowHammer problem and its consequences, we refer the reader to [11, 15, 132].

The fourth key issue is the power and energy consumption of main memory. DRAM is inherently a power and energy hog, as it consumes energy even when it is not used (e.g., it requires periodic memory refresh [14]), due to its charge-based nature. And, energy consumption of main memory is becoming worse due to three major reasons. First, its capacity and complexity are both increasing. Second, main memory has remained off the main processing chip, even though many other platform components have been integrated into the processing chip and have benefited from the aggressive energy scaling and low-energy communication substrate on-chip. Third, the difficulties in DRAM technology scaling are making energy reduction very difficult with technology generations. For example, Lefurgy et al. [133] showed, in 2003 that, in large commercial servers designed by

IBM, the off-chip memory hierarchy (including, at that time, DRAM, interconnects, memory controller, and off-chip caches) consumed between 40% and 50% of the total system energy. The trend has become even worse over the course of the one-to-two decades. In recent computing systems with CPUs or GPUs, only DRAM itself is shown to account for more than 40% of the total system power [134, 135]. Hence, the power and energy consumption of main memory is increasing relative to that of other components in computing platform. As energy efficiency and sustainability are critical necessities in computing platforms today, it is critical to reduce the energy and power consumption of main memory.

3. The Need for Intelligent Memory Controllers to Enhance Memory Scaling

A key promising approach to solving the four major issues above is to design *intelligent memory controllers* that can manage main memory better. If the memory controller is designed to be more intelligent and more programmable, it can, for example, incorporate flexible mechanisms to overcome various types of reliability issues (including RowHammer), manage latencies and power consumption better based on a deep understanding of the DRAM and application characteristics, provide enough support for programmability to prevent security and reliability vulnerabilities that are discovered in the field, and manage various types of memory technologies that are put together as a hybrid main memory to enhance the scaling of the main memory system. We provide a few examples of how an intelligent memory controller can help overcome circuit- and device-level issues we are facing at the main memory level. We believe having intelligent memory controllers can greatly alleviate the scaling issues encountered with main memory today, as we have described in an earlier position paper [1]. This is a direction that is also supported by industry today, as described in an informative paper written collaboratively by Intel and Samsung engineers on DRAM technology scaling issues [8].

First, the RowHammer vulnerability can be prevented by probabilistically refreshing rows that are adjacent to an activated row, with a very low probability. This solution, called PARA (Probabilistic Adjacent Row Activation) [11] was shown to provide strong, programmable guarantees against RowHammer, with very little power, performance and chip area overhead [11]. It requires a slightly more intelligent memory controller that knows (or that can figure out) the physical adjacency of rows in a DRAM chip and that is programmable enough to adjust the probability of adjacent row activation and issue refresh requests to adjacent rows accordingly to the probability supplied by the system. As described by prior work [11, 15, 132], this solution is much lower overhead that increasing the refresh rate across the board for the entire main memory, which is the RowHammer solution employed by existing systems in the field that have simple and rigid memory controllers.

Second, an intelligent memory controller can greatly alleviate the refresh problem in DRAM, and hence its negative consequences on energy, performance, predictability, and technology scaling, by understanding the retention time characteristics of different rows well. It is well known that the retention time of different cells in DRAM are widely different due to process manufacturing variation [14, 101]. Some cells are strong (i.e., they can retain data for hundreds of seconds), whereas some cells are weak (i.e., they can retain data for only 64 ms). Yet, today's memory controllers treat every cell as equal and refresh all rows every 64 ms, which is the worst-case retention time that is allowed. This worst-case refresh rate leads to a large number of unnecessary refreshes, and thus great energy waste and performance loss. Refresh is also shown to be the key technology scaling limiter of DRAM [8], and as such refreshing all DRAM cells at the worst case rates is likely to make DRAM technology scaling difficult. An intelligent memory controller can overcome the refresh problem by identifying the minimum data retention time of each row (during online operation) and refreshing each row at the rate it really requires to be refreshed at or by decommissioning weak rows such that data is not stored in them. As shown by a recent body of work whose aim is to design such an intelligent memory controller that can perform inline profiling of DRAM cell retention times and online adjustment of refresh rate on a per-row basis [14, 101, 136, 137, 138, 139, 140, 141], including the works on RAIDR [14, 101], AVATAR [137] and REAPER [140], such an intelligent memory controller can eliminate more than 75% of all refreshes at very low cost, leading to significant energy reduction, performance improvement, and quality of service benefits, all at the same time. Thus the downsides of DRAM refresh can potentially be overcome with the design of intelligent memory controllers.

Third, an intelligent memory controller can enable performance improvements that can overcome the limitations of memory scaling. As we discuss in Section 2, DRAM latency has remained almost constant over the last twenty years, despite the fact that low-latency computing has become more important during that time. Similar to how intelligent memory controllers handle the refresh problem, the controllers can exploit the fact that

not all cells in DRAM need the same amount of time to be accessed. Manufacturers assign timing parameters that define the amount of time required to perform a memory access. In order to guarantee correct operation, the timing parameters are chosen to ensure that the *worst-case* cell in any DRAM chip that is sold can still be accessed correctly at *worst-case operating temperatures* [22, 24, 26, 105]. However, we find that access latency to cells is very heterogeneous due to variation in both operating conditions (e.g., across different temperatures and operating voltage), manufacturing process (e.g., across different chips and different parts of a chip), and access patterns (e.g., whether or not the cell was recently accessed). We give six examples of how an intelligent memory controller can exploit the various different types of heterogeneity.

(1) At low temperature, DRAM cells contain more charge, and as a result, can be accessed much faster than at high temperatures. We find that, averaged across 115 real DRAM modules from three major manufacturers, read and write latencies of DRAM can be reduced by 33% and 55%, respectively, when operating at relatively low temperature (55 °C) compared to operating at worst-case temperature (85 °C) [24, 142]. Thus, a slightly intelligent memory controller can greatly reduce memory latency by adapting the access latency to operating temperature.

(2) Due to manufacturing process variation, we find that the majority of cells in DRAM (across different chips or within the same chip) can be accessed much faster than the manufacturer-provided timing parameters [22, 24, 26, 31, 105, 142]. An intelligent memory controller can profile the DRAM chip and identify which cells can be accessed reliably at low latency, and use this information to reduce access latencies by as much as 57% [22, 26, 105].

(3) In a similar fashion, an intelligent memory controller can use similar properties of manufacturing process variation to reduce the energy consumption of a computer system, by exploiting the minimum voltage required for safe operation of different parts of a DRAM chip [25, 31]. The key idea is to reduce the operating voltage of a DRAM chip from the standard specification and tolerate the resulting errors by increasing access latency on a per-bank basis, while keeping performance degradation in check.

(4) Bank conflict latencies can be dramatically reduced by making modifications in the DRAM chip such that different subarrays in a bank can be accessed mostly independently, and designing an intelligent memory controller that can take advantage of requests that require data from different subarrays (i.e., exploit subarray-level parallelism) [12, 13].

(5) Access latency to a portion of the DRAM bank can be greatly reduced by partitioning the DRAM array such that a subset of rows can be accessed much faster than the other rows and having an intelligent memory controller that decides what data should be placed in fast rows versus slow rows [23, 142].

(6) We find that a recently-accessed or recently-refreshed memory row can be accessed much more quickly than the standard latency if it needs to be accessed again soon, since the recent access and refresh to the row has replenished the charge of the cells in the row. An intelligent memory controller can thus keep track of the charge level of recently-accessed/refreshed rows and use the appropriate access latency that corresponds to the charge level [30, 103, 104], leading to significant reductions in both access and refresh latencies. Thus, the poor scaling of DRAM latency and energy can potentially be overcome with the design of intelligent memory controllers that can facilitate a large number of effective latency and energy reduction techniques.

Intelligent controllers are already in widespread use in another key part of a modern computing system. In solid-state drives (SSDs) consisting of NAND flash memory, the flash controllers that manage the SSDs are designed to incorporate a significant level of intelligence in order to improve both performance and reliability [143, 144, 145, 146, 147]. Modern flash controllers need to take into account a wide variety of issues such as remapping data, performing wear leveling to mitigate the limited lifetime of NAND flash memory devices, refreshing data based on the current wearout of each NAND flash cell, optimizing voltage levels to maximize memory lifetime, and enforcing fairness across different applications accessing the SSD. Much of the complexity in flash controllers is a result of mitigating issues related to the scaling of NAND flash memory [143, 144, 145, 148, 149]. We argue that in order to overcome scaling issues in DRAM, the time has come for DRAM memory controllers to also incorporate significant intelligence.

As we describe above, introducing intelligence into the memory controller can help us overcome a number of key challenges in memory scaling. In particular, a significant body of works have demonstrated that the key reliability, refresh, and latency/energy issues in memory can be mitigated effectively with an intelligent memory controller. As we discuss in Section 4, this intelligence can go even further, by enabling the memory controllers (and the broader memory system) to perform application

computation in order to overcome the significant data movement bottleneck in existing computing systems.

4. Perils of Processor-Centric Design

A major bottleneck against improving the overall system performance and the energy efficiency of today's computing systems is the high cost of *data movement*. This is a natural consequence of the von Neumann model [150], which separates computation and storage in two different system components (i.e., the computing unit versus the memory/storage unit) that are connected by an off-chip bus. With this model, processing is done only in one place, while data is stored in another, separate place. Thus, data needs to move back and forth between the memory/storage unit and the computing unit (e.g., CPU cores or accelerators).

In order to perform an operation on data that is stored within memory, a costly process is invoked. First, the CPU (or an accelerator) must issue a request to the memory controller, which in turn sends a series of commands across the off-chip bus to the DRAM module. Second, the data is read from the DRAM module and returned to the memory controller. Third, the data is placed in the CPU cache and registers, where it is accessible by the CPU cores. Finally, the CPU can operate (i.e., perform computation) on the data. All these steps consume substantial time and energy in order to bring the data into the CPU chip [4, 7, 151, 152].

In current computing systems, the CPU is the only system component that is able to perform computation on data. The rest of system components are devoted to only data storage (memory, caches, disks) and data movement (interconnects); they are incapable of performing computation. As a result, current computing systems are *grossly imbalanced*, leading to large amounts of energy inefficiency and low performance. As empirical evidence to the gross imbalance caused by the processor-memory dichotomy in the design of computing systems today, we have recently observed that more than 62% of the entire system energy consumed by four major commonly-used mobile consumer workloads (including the Chrome browser, TensorFlow machine learning inference engine, and the VP9 video encoder and decoder) [7]. Thus, the fact that current systems can perform computation only in the computing unit (CPU cores and hardware accelerators) is causing significant waste in terms of energy by necessitating data movement across the entire system.

At least five factors contribute to the performance loss and the energy waste associated with retrieving data from main memory, which we briefly describe next.

First, the width of the off-chip bus between the memory controller and the main memory is narrow, due to pin count and cost constraints, leading to relatively low bandwidth to/from main memory. This makes it difficult to send a large number of requests to memory in parallel.

Second, current computing systems deploy complex multi-level cache hierarchies and latency tolerance/hiding mechanisms (e.g., sophisticated caching algorithms at many different caching levels, multiple complex prefetching techniques, high amounts of multithreading, complex out-of-order execution) to tolerate the data access from memory. These components, while sometimes effective at improving performance, are costly in terms of both die area and energy consumption, as well as the additional latency required to access/manage them. These components also increase the complexity of the system significantly. Hence, the architectural techniques used in modern systems to tolerate the consequences of the dichotomy between processing unit and main memory, lead to significant energy waste and additional complexity.

Third, the caches are not always properly leveraged. Much of the data brought into the caches is *not* reused by the CPU [36, 37], e.g., in streaming or random access applications. This renders the caches either very inefficient or unnecessary for a wide variety of modern workloads.

Fourth, many modern applications, such as graph processing [34, 35], produce random memory access patterns. In such cases, not only the caches but also the off-chip bus and the DRAM memory itself become very inefficient, since only a little part of each cache line retrieved is actually used by the CPU. Such accesses are also not easy to prefetch and often either confuse the prefetchers or render them ineffective. Modern memory hierarchies are not designed to work well for random access patterns.

Fifth, the computing unit and the memory unit are connected through long, power-hungry interconnects. These interconnects impose significant additional latency to every data access and represent a significant fraction of the energy spent on moving data to/from the DRAM memory. In fact, off-chip interconnect latency and energy consumption is a key limiter of performance and energy in modern systems [16, 23, 71, 82] as it greatly exacerbates the cost of data movement.

The increasing disparity between processing technology and memory/communication technology has resulted in systems in which communication (data movement) costs dominate computation costs in terms of energy consumption. The energy consumption of a main memory access is between two to three orders of magnitude the energy consumption of a complex addition

operation today. For example, [152] reports that the energy consumption of a memory access is $\sim$ 115$\times$ the energy consumption of an addition operation. As a result, data movement accounts for 40% [151], 35% [152], and 62% [7] of the total system energy in scientific, mobile, and consumer applications, respectively. This energy waste due to data movement is a huge burden that greatly limits the efficiency and performance of all modern computing platforms, from datacenters with a restricted power budget to mobile devices with limited battery life.

Overcoming all the reasons that cause low performance and large energy inefficiency (as well as high system design complexity) in current computing systems requires a paradigm shift. We believe that future computing architectures should become more *data-centric*: they should (1) perform computation with minimal data movement, and (2) compute where it makes sense (i.e., where the data resides), as opposed to computing solely in the CPU or accelerators. Thus, the traditional rigid dichotomy between the computing units and the memory/communication units needs to be broken and a new paradigm enabling computation where the data resides needs to be invented and enabled.

5. Processing-in-Memory (PIM)

Large amounts of data movement is a major result of the predominant processor-centric design paradigm of modern computers. Eliminating unnecessary data movement between memory unit and compute unit is essential to make future computing architectures higher performance, more energy efficient and sustainable. To this end, *processing-in-memory* (PIM) equips the memory subsystem with the ability to perform computation.

In this section, we describe two promising approaches to implementing PIM in modern architectures. The first approach exploits the existing DRAM architecture and the operational principles of the DRAM circuitry to enable bulk processing operations within the main memory with minimal changes. This minimalist approach can especially be powerful in performing specialized computation in main memory by taking advantage of what the main memory substrate is extremely good at performing with minimal changes to the existing memory chips. The second approach exploits the ability to implement a wide variety of general-purpose processing logic in the logic layer of 3D-stacked memory and thus the high internal bandwidth and low latency available between the logic layer and the memory layers of 3D-stacked memory. This is a more general approach where the logic implemented in the logic layer can be general purpose and thus can benefit a wide variety of applications.

5.1. Approach I: Minimally Changing Memory Chips

One approach to implementing processing-in-memory modifies existing DRAM architectures minimally to extend their functionality with computing capability. This approach takes advantage of the existing interconnects in and analog operational behavior of conventional DRAM architectures (e.g., DDRx, LPDDRx, HBM), without the need for a dedicated logic layer or logic processing elements, and usually with very low overheads. Mechanisms that use this approach take advantage of the high internal bandwidth available within each DRAM cell array. There are a number of example PIM architectures that make use of this approach [31, 82, 83, 84, 85, 86, 92, 93]. In this section, we first focus on two such designs: RowClone, which enables in-DRAM bulk data movement operations [82] and Ambit, which enables in-DRAM bulk bitwise operations [83, 85, 86]. Then, we describe a low-cost substrate that performs data reorganization for non-unit strided access patterns [71].

5.1.1. RowClone

Two important classes of bandwidth-intensive memory operations are (1) *bulk data copy*, where a large quantity of data is copied from one location in physical memory to another; and (2) *bulk data initialization*, where a large quantity of data is initialized to a specific value. We refer to these two operations as *bulk data movement operations*. Prior research [4, 153, 154] has shown that operating systems and data center workloads spend a significant portion of their time performing bulk data movement operations. Therefore, accelerating these operations will likely improve system performance and energy efficiency.

We have developed a mechanism called *RowClone* [82], which takes advantage of the fact that bulk data movement operations do *not* require any computation on the part of the processor. RowClone exploits the internal organization and operation of DRAM to perform bulk data copy/initialization quickly and efficiently inside a DRAM chip. A DRAM chip contains multiple banks, where the banks are connected together and to I/O circuitry by a shared internal bus, each of which is divided into multiple *subarrays* [12, 82, 155]. Each subarray contains many rows of DRAM cells, where each column of DRAM cells is connected together across the multiple rows using *bitlines*.

RowClone consists of two mechanisms that take advantage of the existing DRAM structure. The first mechanism, Fast Parallel Mode, copies the data of a row inside a subarray to another row inside the same DRAM subarray by issuing back-to-back activate (i.e., row open) commands to the source and the destination row. The second mechanism, Pipelined Serial Mode, can transfer an arbitrary number of bytes between two banks using the shared internal bus among banks in a DRAM chip.

RowClone significantly reduces the raw latency and energy consumption of bulk data copy and initialization, leading to 11.6× latency reduction and 74.4× energy reduction for a 4kB bulk page copy (using the Fast Parallel Mode), at very low cost (only 0.01% DRAM chip area overhead) [82]. This reduction directly translates to improvement in performance and energy efficiency of systems running copy or initialization-intensive workloads. Our MICRO 2013 paper [82] shows that the performance of six copy/initialization-intensive benchmarks (including the fork system call, Memcached [156] and a MySQL [157] database) improves between 4% and 66%. For the same six benchmarks, RowClone reduces the energy consumption between 15% and 69%.

5.1.2. Ambit

In addition to bulk data movement, many applications trigger *bulk bitwise operations*, i.e., bitwise operations on large bit vectors [158, 159]. Examples of such applications include bitmap indices [160, 161, 162, 163] and bitwise scan acceleration [164] for databases, accelerated document filtering for web search [165], DNA sequence alignment [166, 167, 168], encryption algorithms [169, 170, 171], graph processing [78], and networking [159]. Accelerating bulk bitwise operations can thus significantly boost the performance and energy efficiency of a wide range applications.

In order to avoid data movement bottlenecks when the system performs these bulk bitwise operations, we have recently proposed a new **A**ccelerator-in-**M**emory for bulk **Bit**wise operations (Ambit) [83, 85, 86]. Unlike prior approaches, Ambit uses the analog operation of existing DRAM technology to perform bulk bitwise operations. Ambit consists of two components. The first component, Ambit–AND–OR, implements a new operation called *triple-row activation*, where the memory controller simultaneously activates three rows. Triple-row activation performs a bitwise majority function across the cells in the three rows, due to the charge sharing principles that govern the operation of the DRAM array. By controlling the initial value of one of the three rows, we can use triple-row activation to perform a bitwise AND or OR of the other two rows. The second component, Ambit–NOT, takes advantage of the two inverters that are connected to each sense amplifier in a DRAM subarray. Ambit–NOT exploits the fact that, at the end of the sense amplification process, the voltage level of one of the inverters represents the negated logical value of the cell. The Ambit design adds a special row to the DRAM array, which is used to capture the negated value that is present in the sense amplifiers. One possible implementation of the special row [86] is a row of *dual-contact cells* (a 2-transistor 1-capacitor cell [172, 173]) that connects to both inverters inside the sense amplifier. With the ability to perform AND, OR, and NOT operations, Ambit is functionally complete: It can reliably perform *any* bulk bitwise operation completely using DRAM technology, even in the presence of significant process variation (see [86] for details).

Averaged across seven commonly-used bitwise operations, Ambit with 8 DRAM banks improves bulk bitwise operation throughput by 44× compared to an Intel Skylake processor [174], and 32× compared to the NVIDIA GTX 745 GPU [175]. Compared to the DDR3 standard, Ambit reduces energy consumption of these operations by 35× on average. Compared to HMC 2.0 [99], Ambit improves bulk bitwise operation throughput by 2.4×. When integrated directly into the HMC 2.0 device, Ambit improves throughput by 9.7× compared to processing in the logic layer of HMC 2.0.

A number of Ambit-like bitwise operation substrates have been proposed in recent years, making use of emerging resistive memory technologies, e.g., phase-change memory (PCM) [17, 19, 176, 177, 178, 179], SRAM, or specialized computational DRAM. These substrates can perform bulk bitwise operations in a special DRAM array augmented with computational circuitry [90] and in PCM [78]. Similar substrates can perform simple arithmetic operations in SRAM [79, 80] and arithmetic and logical operations in memristors [81, 180, 181, 182, 183]. We believe it is extremely important to continue exploring such low-cost Ambit-like substrates, as well as more sophisticated computational substrates, for all types of memory technologies, old and new. Resistive memory technologies are fundamentally non-volatile and amenable to in-place updates, and as such, can lead to even less data movement compared to DRAM, which fundamentally requires some data movement to access the data. Thus, we believe it is very promising to examine the design of emerging resistive memory chips that can incorporate Ambit-like bitwise operations and other types of suitable computation capability.

5.1.3. Gather-Scatter DRAM

Many applications access data structures with different access patterns at different points in time. Depending on the layout of the data structures in the physical memory, some access patterns require non-unit strides. As current memory systems are optimized to access sequential cache lines, non-unit strided accesses exhibit low spatial locality, leading to memory bandwidth waste and cache space waste.

Gather-Scatter DRAM (GS-DRAM) [71] is a low-cost substrate that addresses this problem. It performs in-DRAM data structure reorganization by accessing multiple values that belong to a strided access pattern using a single read/write command in the memory controller. GS-DRAM uses two key new mechanisms. First, GS-DRAM remaps the data of each cache line to different chips such that multiple values of a strided access pattern are mapped to different chips. This enables the possibility of gathering different parts of the strided access pattern concurrently from different chips. Second, instead of sending separate requests to each chip, the GS-DRAM memory controller communicates a pattern ID to the memory module. With the pattern ID, each chip computes the address to be accessed independently. This way, the returned cache line contains different values of the strided pattern gathered from different chips.

GS-DRAM achieves near-ideal memory bandwidth and cache utilization in real-world workloads, such as in-memory databases and matrix multiplication. For in-memory databases, GS-DRAM outperforms a transactional workload with column store layout by 3× and an analytics workload with row store layout by 2×, thereby getting the best performance for both transactional and analytical queries on databases (which in general benefit from different types of data layouts). For matrix multiplication, GS-DRAM is 10% faster than the best-performing tiled implementation of the matrix multiplication algorithm.

5.2. Approach II: PIM using 3D-Stacked Memory

Several works propose to place some form of processing logic (typically accelerators, simple cores, or reconfigurable logic) inside the logic layer of 3D-stacked memory [97]. This *PIM processing logic*, which we also refer to as *PIM cores* or *PIM engines*, interchangeably, can execute portions of applications (from individual instructions to functions) or entire threads and applications, depending on the design of the architecture. Such PIM engines have high-bandwidth and low-latency access to the memory stacks that are on top of them, since the logic layer and the memory layers are connected via high-bandwidth vertical connections [97], e.g., through-silicon vias. In this section, we discuss how systems can make use of relatively simple PIM engines within the logic layer to avoid data movement and thus obtain significant performance and energy improvements on a wide variety of application domains.

5.2.1. Tesseract: Graph Processing

A popular modern application is large-scale graph processing [87, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193]. Graph processing has broad applicability and use in many domains, from social networks to machine learning, from data analytics to bioinformatics. Graph analysis workloads are known to put significant pressure on memory bandwidth due to (1) large amounts of random memory accesses across large memory regions (leading to very limited cache efficiency and very large amounts of unnecessary data transfer on the memory bus) and (2) very small amounts of computation per each data item fetched from memory (leading to very limited ability to hide long memory latencies and exacerbating the energy bottleneck by exercising the huge energy disparity between memory access and computation). These two characteristics make it very challenging to scale up such workloads despite their inherent parallelism, especially with conventional architectures based on large on-chip caches and relatively scarce off-chip memory bandwidth for random access.

We can exploit the high bandwidth as well as the potential computation capability available within the logic layer of 3D-stacked memory to overcome the limitations of conventional architectures for graph processing. To this end, we design a programmable PIM accelerator for large-scale graph processing, called Tesseract [34]. Tesseract consists of (1) a new hardware architecture that effectively utilizes the available memory bandwidth in 3D-stacked memory by placing simple in-order processing cores in the logic layer and enabling each core to manipulate data only on the memory partition it is assigned to control, (2) an efficient method of communication between different in-order cores within a 3D-stacked memory to enable each core to request computation on data elements that reside in the memory partition controlled by another core, and (3) a message-passing based programming interface, similar to how modern distributed systems are programmed, which enables remote function calls on data that resides in each memory partition. The Tesseract design moves functions to data rather than moving data elements across different memory partitions and cores. It also includes two hardware prefetchers specialized for memory access patterns of graph processing, which operate based on the hints pro-

vided by our programming model. Our comprehensive evaluations using five state-of-the-art graph processing workloads with large real-world graphs show that the proposed Tesseract PIM architecture improves average system performance by 13.8× and achieves 87% average energy reduction over conventional systems.

5.2.2. Consumer Workloads

A very popular domain of computing today consists of consumer devices, which include smartphones, tablets, web-based computers such as Chromebooks, and wearable devices. In consumer devices, energy efficiency is a first-class concern due to the limited battery capacity and the stringent thermal power budget. We find that *data movement* is a major contributor to the total system energy and execution time in modern consumer devices. Across all of the popular modern applications we study (described in the next paragraph), we find that a massive 62.7% of the total system energy, on average, is spent on data movement across the memory hierarchy [7].

We comprehensively analyze the energy and performance impact of data movement for several widely-used Google consumer workloads [7], which account for a significant portion of the applications executed on consumer devices. These workloads include (1) the Chrome web browser [194], which is a very popular browser used in mobile devices and laptops; (2) TensorFlow Mobile [195], Google's machine learning framework, which is used in services such as Google Translate, Google Now, and Google Photos; (3) the VP9 video playback engine [196], and (4) the VP9 video capture engine [196], both of which are used in many video services such as YouTube and Google Hangouts. We find that offloading key functions to the logic layer can greatly reduce data movement in all of these workloads. However, there are challenges to introducing PIM in consumer devices, as consumer devices are extremely stringent in terms of the area and energy budget they can accommodate for any new hardware enhancement. As a result, we need to identify what kind of in-memory logic can both (1) *maximize energy efficiency* and (2) be implemented at *minimum possible cost, in terms of both area overhead and complexity*.

We find that many of target functions for PIM in consumer workloads are comprised of simple operations such as *memcopy*, *memset*, basic arithmetic and bitwise operations, and simple data shuffling and reorganization routines. Therefore, we can relatively easily implement these PIM target functions in the logic layer of 3D-stacked memory using either (1) a small low-power general-purpose embedded core or (2) a group of small fixed-function accelerators. Our analysis shows that the area of a PIM core and a PIM accelerator take up no more than 9.4% and 35.4%, respectively, of the area available for PIM logic in an HMC-like [197] 3D-stacked memory architecture. Both the PIM core and PIM accelerator eliminate a large amount of data movement, and thereby significantly reduce total system energy (by an average of 55.4% across all the workloads) and execution time (by an average of 54.2%).

5.2.3. GPU Applications

In the last decade, Graphics Processing Units (GPUs) have become the accelerator of choice for a wide variety of data-parallel applications. They deploy thousands of in-order, SIMT (Single Instruction Multiple Thread) cores that run lightweight threads. Their multithreaded architecture is devised to hide the long latency of memory accesses by interleaving threads that execute arithmetic and logic operations. Despite that, many GPU applications are still very memory-bound [198, 199, 200, 201, 202, 203, 204, 205, 206, 207], because the limited off-chip pin bandwidth cannot supply enough data to the running threads.

3D-stacked memory architectures present a promising opportunity to alleviate the memory bottleneck in GPU systems. GPU cores placed in the logic layer of a 3D-stacked memory can be directly connected to the DRAM layers with high bandwidth (and low latency) connections. In order to leverage the potential performance benefits of such systems, it is necessary to enable computation offloading and data mapping to multiple such compute-capable 3D-stacked memories, such that GPU applications can benefit from processing-in-memory capabilities in the logic layers of such memories.

TOM (Transparent Offloading and Mapping) [59] proposes two mechanisms to address computation offloading and data mapping in such a system in a programmer-transparent manner. First, it introduces new compiler analysis techniques to identify code sections in GPU kernels that can benefit from PIM offloading. The compiler estimates the potential memory bandwidth savings for each code block. To this end, the compiler compares the bandwidth consumption of the code block, when executed on the regular GPU cores, to the bandwidth cost of transmitting/receiving input/output registers, when offloading to the GPU cores in the logic layers. At runtime, a final offloading decision is made based on system conditions, such as contention for processing resources in the logic layer. Second, a software/hardware cooperative mechanism predicts the memory pages that will be accessed by offloaded code, and places such pages in the same 3D-stacked memory cube where the code will be executed. The goal is to make PIM effective by

ensuring that the data needed by the PIM cores is in the same memory stack. Both mechanisms are completely transparent to the programmer, who only needs to write regular GPU code without any explicit PIM instructions or any other modification to the code. TOM improves the average performance of a variety of GPGPU workloads by 30% and reduces the average energy consumption by 11% with respect to a baseline GPU system without PIM offloading capabilities.

A related work [60] identifies GPU kernels that are suitable for PIM offloading by using a regression-based affinity prediction model. A concurrent kernel management mechanism uses the affinity prediction model and determines which kernels should be scheduled concurrently to maximize performance. This way, the proposed mechanism enables the simultaneous exploitation of the regular GPU cores and the in-memory GPU cores. This scheduling technique improves performance and energy efficiency by an average of 42% and 27%, respectively.

5.2.4. PEI: PIM-Enabled Instructions

PIM-Enabled Instructions (PEI) [35] aims to provide the minimal processing-in-memory support to take advantage of PIM using 3D-stacked memory, in a way that can achieve significant performance and energy benefits without changing the computing system significantly. To this end, PEI proposes a collection of simple instructions, which introduce negligible changes to the computing system and no changes to the programming model or the virtual memory system, in a system with 3D-stacked memory. These instructions, inserted by the compiler/programmer to code written in a regular program, are operations that can be executed either in a traditional host CPU (that fetches and decodes them) or the PIM engine in 3D-stacked memory.

PIM-Enabled Instructions are based on two key ideas. First, a PEI is a cache-coherent, virtually-addressed host processor instruction that operates on only a single cache block. It requires no changes to the sequential execution and programming model, no changes to virtual memory, minimal changes to cache coherence, and no need for special data mapping to take advantage of PIM (because each PEI is restricted to a single memory module due to the single cache block restriction it has). Second, a Locality-Aware Execution runtime mechanism decides dynamically where to execute a PEI (i.e., either the host processor or the PIM logic) based on simple locality characteristics and simple hardware predictors. This runtime mechanism executes the PEI at the location that maximizes performance. In summary, PIM-Enabled Instructions provide the illusion that PIM operations are executed as if they were host instructions.

Examples of PEIs are integer increment, integer minimum, floating-point addition, hash table probing, histogram bin index, Euclidean distance, and dot product [35]. Data-intensive workloads such as graph processing, in-memory data analytics, machine learning, and data mining can significantly benefit from these PEIs. Across 10 key data-intensive workloads, we observe that the use of PEIs in these workloads, in combination with the Locality-Aware Execution runtime mechanism, leads to an average performance improvement of 47% and an average energy reduction of 25% over a baseline CPU.

6. Enabling the Adoption of PIM

Pushing some or all of the computation for a program from the CPU to memory introduces new challenges for system architects and programmers to overcome. These challenges must be addressed in order for PIM to be adopted as a mainstream architecture in a wide variety of systems and workloads, and in a seamless manner that does not place heavy burden on the vast majority of programmers. In this section, we discuss several of these system-level and programming-level challenges, and highlight a number of our works that have addressed these challenges for a wide range of PIM architectures.

6.1. Programming Model and Code Generation

Two open research questions to enable the adoption of PIM are 1) what should the programming models be, and 2) how can compilers and libraries alleviate the programming burden.

While PIM-Enabled Instructions [35] work well for offloading small amounts of computation to memory, they can potentially introduce overheads while taking advantage of PIM for large tasks, due to the need to frequently exchange information between the PIM processing logic and the CPU. Hence, there is a need for researchers to investigate how to integrate PIM instructions with other compiler-based methods or library calls that can support PIM integration, and how these approaches can ease the burden on the programmer, by enabling seamless offloading of instructions or function/library calls.

Such solutions can often be platform-dependent. One of our recent works [59] examines compiler-based mechanisms to decide what portions of code should be offloaded to PIM processing logic in a GPU-based system in a manner that is transparent to the GPU programmer. Another recent work [60] examines system-level techniques that decide which GPU application kernels are suitable for PIM execution.

Determining effective programming interfaces and the necessary compiler/library support to perform PIM remain open research and design questions, which are important for future works to tackle.

6.2. PIM Runtime: Scheduling and Data Mapping

We identify four key runtime issues in PIM: (1) what code to execute near data, (2) when to schedule execution on PIM (i.e., when is it worth offloading computation to the PIM cores), (3) how to map data to multiple memory modules such that PIM execution is viable and effective, and (4) how to effectively share/partition PIM mechanisms/accelerators at runtime across multiple threads/cores to maximize performance and energy efficiency. We have already proposed several approaches to solve these four issues.

Our recent works in PIM processing identify suitable *PIM offloading candidates* with different granularities. PIM-Enabled Instructions [35] propose various operations that can benefit from execution near or inside memory, such as integer increment, integer minimum, floating-point addition, hash table probing, histogram bin index, Euclidean distance, and dot product. In [7], we find simple functions with intensive data movement that are suitable for PIM in consumer workloads (e.g., Chrome web browser, TensorFlow Mobile, video playback, and video capture), as described in Section 5.2.2. Bulk memory operations (copy, initialization) and bulk bitwise operations are good candidates for in-DRAM processing [82, 83, 86, 93]. GPU applications also contain several parts that are suitable for offloading to PIM engines [59, 60].

In several of our research works, we propose runtime mechanisms for *dynamic scheduling* of PIM offloading candidates, i.e., mechanisms that decide whether or not to actually offload code that is marked to be potentially offloaded to PIM engines. In [35], we develop a locality-aware scheduling mechanism for PIM-enabled instructions. For GPU-based systems [59, 60], we explore the combination of compile-time and runtime mechanisms for identification and dynamic scheduling of PIM offloading candidates.

The best *mapping of data and code* that enables the maximal benefits from PIM depends on the applications and the computing system configuration. For instance, in [59], we present a software/hardware mechanism to map data and code to several 3D-stacked memory cubes in regular GPU applications with relatively regular memory access patterns. This work also deals with effectively *sharing PIM engines across multiple threads*, as GPU code sections are offloaded from different GPU cores. Developing new approaches to data/code mapping and scheduling for a large variety of applications and possible core and memory configurations is still necessary.

In summary, there are still several key research questions that should be investigated in runtime systems for PIM, which perform scheduling and data/code mapping:

- What are simple mechanisms to enable and disable PIM execution? How can PIM execution be throttled for highest performance gains? How should data locations and access patterns affect where/whether PIM execution should occur?
- Which parts of a given application's code should be executed on PIM? What are simple mechanisms to identify when those parts of the application code can benefit from PIM?
- What are scheduling mechanisms to share PIM engines between multiple requesting cores to maximize benefits obtained from PIM?
- What are simple mechanisms to manage access to a memory that serves both CPU requests and PIM requests?

6.3. Memory Coherence

In a traditional multithreaded execution model that makes use of shared memory, writes to memory must be coordinated between multiple CPU cores, to ensure that threads do not operate on stale data values. Since CPUs include per-core private caches, when one core writes data to a memory address, cached copies of the data held within the caches of other cores must be updated or invalidated, using a mechanism known as *cache coherence*. Within a modern chip multiprocessor, the per-core caches perform coherence actions over a shared interconnect, with hardware coherence protocols.

Cache coherence is a major system challenge for enabling PIM architectures as general-purpose execution engines, as PIM processing logic can modify the data it processes, and this data may also be needed by CPU cores. If PIM processing logic is coherent with the processor, the PIM programming model is relatively simple, as it remains similar to conventional shared memory multithreaded programming, which makes PIM architectures easier to adopt in general-purpose systems. Thus, allowing PIM processing logic to maintain such a simple and traditional shared memory programming model can facilitate the widespread adoption of PIM. However, employing traditional fine-grained cache coherence (e.g., a cache-block based MESI protocol [208]) for PIM forces a large number of coherence messages to traverse the narrow processor-memory bus, potentially undoing the

benefits of high-bandwidth and low-latency PIM execution. Unfortunately, solutions for coherence proposed by prior PIM works [35, 59] either place some restrictions on the programming model (by eliminating coherence and requiring message passing based programming) or limit the performance and energy gains achievable by a PIM architecture.

We have developed a new coherence protocol, LazyPIM [70, 209], that maintains cache coherence between PIM processing logic and CPU cores *without* sending coherence requests for every memory access. Instead, LazyPIM efficiently provides coherence by having PIM processing logic *speculatively* acquire coherence permissions, and then later sends compressed *batched* coherence lookups to the CPU to determine whether or not its speculative permission acquisition violated the coherence semantics. As a result of this "lazy" checking of coherence violations, LazyPIM approaches near-ideal coherence behavior: the performance and energy consumption of a PIM architecture with LazyPIM are, respectively, within 5.5% and 4.4% the performance and energy consumption of a system where coherence is performed at zero latency and energy cost.

Despite the leap that LazyPIM [70, 209] represents for memory coherence in computing systems with PIM support, we believe that it is still necessary to explore other solutions for memory coherence that can efficiently deal with all types of workloads and PIM offloading granularities.

6.4. Virtual Memory Support

When an application needs to access its data inside the main memory, the CPU core must first perform an *address translation*, which converts the data's virtual address into a *physical* address within main memory. If the translation metadata is not available in the CPU's translation lookaside buffer (TLB), the CPU must invoke the page table walker in order to perform a long-latency page table walk that involves multiple *sequential* reads to the main memory and lowers the application's performance. In modern systems, the virtual memory system also provides access protection mechanisms.

A naive solution to reducing the overhead of page walks is to utilize PIM engines to perform page table walks. This can be done by duplicating the content of the TLB and moving the page walker to the PIM processing logic in main memory. Unfortunately, this is either difficult or expensive for three reasons. First, coherence has to be maintained between the CPU's TLBs and the memory-side TLBs. This introduces extra complexity and off-chip requests. Second, duplicating the TLBs increases the storage and complexity overheads on the memory side, which should be carefully contained. Third, if main memory is shared across CPUs with different types of architectures, page table structures and the implementation of address translations can be different across the different architectures. Ensuring compatibility between the in-memory TLB/page walker and all possible types of virtual memory architecture designs can be complicated and often not even practically feasible.

To address these concerns and reduce the overhead of virtual memory, we explore a tractable solution for PIM address translation as part of our in-memory pointer chasing accelerator, IMPICA [62]. IMPICA exploits the high bandwidth available within 3D-stacked memory to traverse a chain of virtual memory pointers within DRAM, *without* having to look up virtual-to-physical address translations in the CPU translation lookaside buffer (TLB) and without using the page walkers within the CPU. IMPICA's key ideas are 1) to use a region-based page table, which is optimized for PIM acceleration, and 2) to decouple address calculation and memory access with two specialized engines. IMPICA improves the performance of pointer chasing operations in three commonly-used linked data structures (linked lists, hash tables, and B-trees) by 92%, 29%, and 18%, respectively. On a real database application, DBx1000, IMPICA improves transaction throughput and response time by 16% and 13%, respectively. IMPICA also reduces overall system energy consumption (by 41%, 23%, and 10% for the three commonly-used data structures, and by 6% for DBx1000).

Beyond pointer chasing operations that are tackled by IMPICA [62], providing efficient mechanisms for PIM-based virtual-to-physical address translation (as well as access protection) remains a challenge for the generality of applications, especially those that access large amounts of virtual memory [210, 211, 212]. We believe it is important to explore new ideas to address this PIM challenge in a scalable and efficient manner.

6.5. Data Structures for PIM

Current systems with many cores run applications with concurrent data structures to achieve high performance and scalability, with significant benefits over sequential data structures. Such concurrent data structures are often used in heavily-optimized server systems today, where high performance is critical. To enable the adoption of PIM in such many-core systems, it is necessary to develop concurrent data structures that are specifically tailored to take advantage of PIM.

Pointer chasing data structures and *contended data structures* require careful analysis and design to leverage the high bandwidth and low latency of 3D-stacked

memories [72]. First, pointer chasing data structures, such as linked-lists and skip-lists, have a high degree of inherent parallelism and low contention, but a naive implementation in PIM cores is burdened by hard-to-predict memory access patterns. By combining and partitioning the data across 3D-stacked memory vaults, it is possible to fully exploit the inherent parallelism of these data structures. Second, contended data structures, such as FIFO queues, are a good fit for CPU caches because they expose high locality. However, they suffer from high contention when many threads access them concurrently. Their performance on traditional CPU systems can be improved using a new PIM-based FIFO queue [72]. The proposed PIM-based FIFO queue uses a PIM core to perform enqueue and dequeue operations requested by CPU cores. The PIM core can pipeline requests from different CPU cores for improved performance.

As recent work [72] shows, PIM-managed concurrent data structures can outperform state-of-the-art concurrent data structures that are designed for and executed on multiple cores. We believe and hope that future work will enable other types of data structures (e.g., hash tables, search trees, priority queues) to benefit from PIM-managed designs.

6.6. Benchmarks and Simulation Infrastructures

To ease the adoption of PIM, it is critical that we accurately assess the benefits and shortcomings of PIM. Accurate assessment of PIM requires (1) a preferably large set of real-world memory-intensive applications that have the potential to benefit significantly when executed near memory, (2) a rigorous methodology to (automatically) identify PIM offloading candidates, and (3) simulation/evaluation infrastructures that allow architects and system designers to accurately analyze the benefits and overheads of adding PIM processing logic to memory and executing code on this processing logic.

In order to explore what processing logic should be introduced near memory, and to know what properties are ideal for PIM kernels, we believe it is important to begin by developing a *real-world benchmark suite* of a wide variety of applications that can potentially benefit from PIM. While many data-intensive applications, such as pointer chasing and bulk memory copy, can potentially benefit from PIM, it is crucial to examine important candidate applications for PIM execution, and for researchers to agree on a common set of these candidate applications to focus the efforts of the community as well as to enable reproducibility of results, which is important to assess the relative benefits of different ideas developed by different researchers. We believe that these applications should come from a number of popular and emerging domains. Examples of potential domains include data-parallel applications, neural networks, machine learning, graph processing, data analytics, search/filtering, mobile workloads, bioinformatics, Hadoop/Spark programs, security/cryptography, and in-memory data stores. Many of these applications have large data sets and can benefit from high memory bandwidth and low memory latency benefits provided by computation near memory. In our prior work, we have started identifying several applications that can benefit from PIM in graph processing frameworks [34, 35], pointer chasing [33, 62], databases [62, 70, 71, 209], consumer workloads [7], machine learning [7], and GPGPU workloads [59, 60]. However, there is significant room for methodical development of a large-scale PIM benchmark suite, which we hope future work provides.

A systematic *methodology* for (automatically) identifying potential PIM kernels (i.e., code portions that can benefit from PIM) within an application can, among many other benefits, 1) ease the burden of programming PIM architectures by aiding the programmer to identify what should be offloaded, 2) ease the burden of and improve the reproducibility of PIM research, 3) drive the design and implementation of PIM functional units that many types of applications can leverage, 4) inspire the development of tools that programmers and compilers can use to automate the process of offloading portions of existing applications to PIM processing logic, and 5) lead the community towards convergence on PIM designs and offloading candidates.

We also need *simulation infrastructures* to accurately model the performance and energy of PIM hardware structures, available memory bandwidth, and communication overheads when we execute code near or inside memory. Highly-flexible and commonly-used memory simulators (e.g., Ramulator [213, 214], SoftMC [29, 215]) can be combined with full-system simulators (e.g., gem5 [216], zsim [217], gem5-gpu [218], GPGPUSim [219]) to provide a robust environment that can evaluate how various PIM architectures affect the *entire compute stack*, and can allow designers to identify memory, workload, and system characteristics that affect the efficiency of PIM execution. We believe it is critical to support the open source development such simulation and emulation infrastructures for assessing the benefits of a wide variety of PIM designs.

7. Conclusion and Future Outlook

Data movement is a major performance and energy bottleneck plaguing modern computing systems. A large fraction of system energy is spent on moving data across

the memory hierarchy into the processors (and accelerators), the only place where computation is performed in a modern system. Fundamentally, the large amounts of data movement are caused by the processor-centric design of modern computing systems: processing of data is performed only in the processors (and accelerators), which are far away from the data, and as a result, data moves a lot in the system, to facilitate computation on it.

In this work, we argue for a paradigm shift in the design of computing systems toward a data-centric design that enables computation capability in places where data resides and thus performs computation with minimal data movement. *Processing-in-memory* (PIM) is a fundamentally data-centric design approach for computing systems that enables the ability to perform operations in or near memory. Recent advances in modern memory architectures have enabled us to extensively explore two novel approaches to designing PIM architectures. First, with minimal changes to memory chips, we show that we can perform a number of important and widely-used operations (e.g., memory copy, data initialization, bulk bitwise operations, data reorganization) within DRAM. Second, we demonstrate how embedded computation capability in the logic layer of 3D-stacked memory can be used in a variety of ways to provide significant performance improvements and energy savings, across a large range of application domains and computing platforms.

Despite the extensive design space that we have studied so far, a number of key challenges remain to enable the widespread adoption of PIM in future computing systems [94, 95]. Important challenges include developing easy-to-use programming models for PIM (e.g., PIM application interfaces, compilers and libraries designed to abstract away PIM architecture details from programmers), and extensive runtime support for PIM (e.g., scheduling PIM operations, sharing PIM logic among CPU threads, cache coherence, virtual memory support). We hope that providing the community with (1) a large set of memory-intensive benchmarks that can potentially benefit from PIM, (2) a rigorous methodology to identify PIM-suitable parts within an application, and (3) accurate simulation infrastructures for estimating the benefits and overheads of PIM will empower researchers to address remaining challenges for the adoption of PIM.

We firmly believe that it is time to design principled system architectures to solve the data movement problem of modern computing systems, which is caused by the rigid dichotomy and imbalance between the computing unit (CPUs and accelerators) and the memory/storage unit. Fundamentally solving the data movement problem requires a paradigm shift to a more data-centric computing system design, where computation happens in or near memory/storage, with minimal movement of data. Such a paradigm shift can greatly push the boundaries of future computing systems, leading to orders of magnitude improvements in energy and performance (as we demonstrated with some examples in this work), potentially enabling new applications and computing platforms.

Acknowledgments

This work is based on a keynote talk delivered by Onur Mutlu at the 3rd Mobile System Technologies (MST) Workshop in Milan, Italy on 27 October 2017 [220]. The mentioned keynote talk is similar to a series of talks given by Onur Mutlu in a wide variety of venues since 2015 until now. This talk has evolved significantly over time with the accumulationof new works and feedback received from many audiences. A recent version of the talk was delivered as a distinguished lecture at George Washington University [221].

This article and the associated talks are based on research done over the course of the past seven years in the SAFARI Research Group on the topic of processing-in-memory (PIM). We thank all of the members of the SAFARI Research Group, and our collaborators at Carnegie Mellon, ETH Zürich, and other universities, who have contributed to the various works we describe in this paper. Thanks also goes to our research group's industrial sponsors over the past ten years, especially Alibaba, Google, Huawei, Intel, Microsoft, NVIDIA, Samsung, Seagate, and VMware. This work was also partially supported by the Intel Science and Technology Center for Cloud Computing, the Semiconductor Research Corporation, the Data Storage Systems Center at Carnegie Mellon University, various NSF grants, and various awards, including the NSF CAREER Award, the Intel Faculty Honor Program Award, and a number of Google Faculty Research Awards to Onur Mutlu.

References

[1] O. Mutlu, Memory Scaling: A Systems Architecture Perspective, IMW (2013).
[2] O. Mutlu, L. Subramanian, Research Problems and Opportunities in Memory Systems, SUPERFRI (2014).
[3] J. Dean, L. A. Barroso, The Tail at Scale, CACM (2013).
[4] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, D. Brooks, Profiling a Warehouse-Scale Computer, in: ISCA, 2015.
[5] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware, in: ASPLOS, 2012.
[6] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, B. Qiu, BigDataBench: A Big Data Benchmark Suite From Internet Services, in: HPCA, 2014.

[7] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, O. Mutlu, Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks, in: ASPLOS, 2018.

[8] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, J. Choi, Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling, in: The Memory Forum, 2014.

[9] S. A. McKee, Reflections on the Memory Wall, in: CF, 2004.

[10] M. V. Wilkes, The Memory Gap and the Future of High Performance Memories, CAN (2001).

[11] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, O. Mutlu, Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors, in: ISCA, 2014.

[12] Y. Kim, V. Seshadri, D. Lee, J. Liu, O. Mutlu, A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM, in: ISCA, 2012.

[13] Y. Kim, Architectural Techniques to Enhance DRAM Scaling, Ph.D. thesis, Carnegie Mellon University (2015).

[14] J. Liu, B. Jaiyen, R. Veras, O. Mutlu, RAIDR: Retention-Aware Intelligent DRAM Refresh, in: ISCA, 2012.

[15] O. Mutlu, The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser, in: DATE, 2017.

[16] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, O. Mutlu, Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM, in: PACT, 2015.

[17] B. C. Lee, E. Ipek, O. Mutlu, D. Burger, Architecting Phase Change Memory as a Scalable DRAM Alternative, in: ISCA, 2009.

[18] H. Yoon, R. A. J. Meza, R. Harding, O. Mutlu, Row Buffer Locality Aware Caching Policies for Hybrid Memories, in: ICCD, 2012.

[19] H. Yoon, J. Meza, N. Muralimanohar, N. P. Jouppi, O. Mutlu, Efficient Data Mapping and Buffering Techniques for Multilevel Cell Phase-Change Memories, ACM TACO (2014).

[20] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, T. F. Wenisch, Disaggregated Memory for Expansion and Sharing in Blade Servers, in: ISCA, 2009.

[21] W. A. Wulf, S. A. McKee, Hitting the Memory Wall: Implications of the Obvious, CAN (1995).

[22] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, O. Mutlu, Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization, in: SIGMETRICS, 2016.

[23] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, O. Mutlu, Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture, in: HPCA, 2013.

[24] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, O. Mutlu, Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case, in: HPCA, 2015.

[25] K. K. Chang, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, O. Mutlu, Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms, in: SIGMETRICS, 2017.

[26] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, O. Mutlu, Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms, in: SIGMETRICS, 2017.

[27] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, O. Mutlu, Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory, in: DSN, 2014.

[28] Y. Luo, S. Ghose, T. Li, S. Govindan, B. Sharma, B. Kelly, A. Boroumand, O. Mutlu, Using ECC DRAM to Adaptively Increase Memory Capacity, arXiv:1706.08870 [cs:AR] (2017).

[29] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, O. Mutlu, SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies, in: HPCA, 2017.

[30] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, O. Mutlu, ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality, in: HPCA, 2016.

[31] K. K. Chang, Understanding and Improving the Latency of DRAM-Based Memory Systems, Ph.D. thesis, Carnegie Mellon University (2017).

[32] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, Y. N. Patt, Accelerating Dependent Cache Misses with an Enhanced Memory Controller, in: ISCA, 2016.

[33] M. Hashemi, O. Mutlu, Y. N. Patt, Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads, in: MICRO, 2016.

[34] J. Ahn, S. Hong, S. Yoo, O. Mutlu, K. Choi, A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing, in: ISCA, 2015.

[35] J. Ahn, S. Yoo, O. Mutlu, K. Choi, PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture, in: ISCA, 2015.

[36] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., J. Emer, Adaptive Insertion Policies for High-Performance Caching, in: ISCA, 2007.

[37] M. K. Qureshi, M. A. Suleman, Y. N. Patt, Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines, in: HPCA, 2007.

[38] R. Kumar, G. Hinton, A Family of 45nm IA Processors, in: ISSCC, 2009.

[39] H. S. Stone, A Logic-in-Memory Computer, TC (1970).

[40] D. E. Shaw, S. J. Stolfo, H. Ibrahim, B. Hillyer, G. Wiederhold, J. Andrews, The NON-VON Database Machine: A Brief Overview, IEEE Database Eng. Bull. (1981).

[41] D. G. Elliott, W. M. Snelgrove, M. Stumm, Computational RAM: A Memory-SIMD Hybrid and Its Application to DSP, in: CICC, 1992.

[42] P. M. Kogge, EXECUBE–A New Architecture for Scaleable MPPs, in: ICPP, 1994.

[43] M. Gokhale, B. Holmes, K. Iobst, Processing in Memory: The Terasys Massively Parallel PIM Array, IEEE Computer (1995).

[44] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick, A Case for Intelligent RAM, IEEE Micro (1997).

[45] M. Oskin, F. T. Chong, T. Sherwood, Active Pages: A Computation Model for Intelligent Memory, in: ISCA, 1998.

[46] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, J. Torrellas, FlexRAM: Toward an Advanced Intelligent Memory System, in: ICCD, 1999.

[47] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca, The Architecture of the DIVA Processing-in-Memory Chip, in: SC, 2002.

[48] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, M. Horowitz, Smart Memories: A Modular Reconfigurable Architecture, in: ISCA, 2000.

[49] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, R. McKenzie, Computational RAM: Implementing Processors in Memory, IEEE Design & Test (1999).

[50] E. Riedel, G. Gibson, C. Faloutsos, Active Storage for Large-scale Data Mining and Multimedia Applications, in: VLDB, 1998.

[51] K. Keeton, D. A. Patterson, J. M. Hellerstein, A Case for Intelligent Disks (IDISKs), SIGMOD Rec. (1998).

[52] S. Kaxiras, R. Sugumar, Distributed Vector Architecture: Beyond a Single Vector-IRAM, in: First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, 1997.

[53] A. Acharya, M. Uysal, J. Saltz, Active Disks: Programming Model, Algorithms and Evaluation, in: ASPLOS, 1998.

[54] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, F. Franchetti, Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware, in: HPEC, 2013.

[55] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, F. Li, NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads, in: ISPASS, 2014.

[56] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, M. Ignatowski, TOP-PIM: Throughput-Oriented Programmable Processing in Memory, in: HPDC, 2014.

[57] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, N. S. Kim, NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules, in: HPCA, 2015.

[58] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, M. Ignatowski, A Processing in Memory Taxonomy and a Case for Studying Fixed-Function PIM, in: WoNDP, 2013.

[59] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Conner, N. Vijaykumar, O. Mutlu, S. Keckler, Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems, in: ISCA, 2016.

[60] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, C. R. Das, Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities, in: PACT, 2016.

[61] B. Akin, F. Franchetti, J. C. Hoe, Data Reorganization in Memory Using 3D-Stacked DRAM, in: ISCA, 2015.

[62] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, O. Mutlu, Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation, in: ICCD, 2016.

[63] O. O. Babarinsa, S. Idreos, JAFAR: Near-Data Processing for Databases, in: SIGMOD, 2015.

[64] J. H. Lee, J. Sim, H. Kim, BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models, in: PACT, 2015.

[65] M. Gao, C. Kozyrakis, HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing, in: HPCA, 2016.

[66] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, Y. Xie, PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory, in: ISCA, 2016.

[67] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, D. Chang, Biscuit: A Framework for Near-Data Processing of Big Data Workloads, in: ISCA, 2016.

[68] D. Kim, J. Kung, S. Chai, S. Yalamanchili, S. Mukhopadhyay, Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory, in: ISCA, 2016.

[69] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, N. S. Kim, Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems, in: MICRO, 2016.

[70] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, O. Mutlu, LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory, CAL (2016).

[71] V. Seshadri, T. Mullins, A. Boroumand, O. Mutli, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses, in: MICRO, 2015.

[72] Z. Liu, I. Calciu, M. Herlihy, O. Mutlu, Concurrent Data Structures for Near-Memory Computing, in: SPAA, 2017.

[73] M. Gao, G. Ayers, C. Kozyrakis, Practical Near-Data Processing for In-Memory Analytics Frameworks, in: PACT, 2015.

[74] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, F. Franchetti, 3D-Stacked Memory-Side Acceleration: Accelerator and System Design, in: WoNDP, 2014.

[75] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, R. Nair, Data Access Optimization in a Processing-in-Memory System, in: CF, 2015.

[76] A. Morad, L. Yavits, R. Ginosar, GP-SIMD Processing-in-Memory, ACM TACO (2015).

[77] S. M. Hassan, S. Yalamanchili, S. Mukhopadhyay, Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore, in: MEMSYS, 2015.

[78] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, Y. Xie, Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories, in: DAC, 2016.

[79] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, K. Curewitz, An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM, in: ICASSP, 2014.

[80] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, R. Das, Compute Caches, in: HPCA, 2017.

[81] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, V. Srikumar, ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars, in: ISCA, 2016.

[82] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, M. A. Kozuch, P. B. Gibbons, T. C. Mowry, RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization, in: MICRO, 2013.

[83] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, T. C. Mowry, Fast Bulk Bitwise AND and OR in DRAM, CAL (2015).

[84] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, O. Mutlu, Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM, in: HPCA, 2016.

[85] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, T. C. Mowry, Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM, arXiv:1611.09988 [cs:AR] (2016).

[86] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, T. C. Mowry, Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology, in: MICRO, 2017.

[87] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, H. Kim, GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks, in: HPCA, 2017.

[88] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, O. Mutlu, GRIM-Filter: Fast Seed Filtering in Read Mapping Using Emerging Memory Technologies, arXiv:1708.04329 [q-bio.GN] (2017).

[89] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, O. Mutlu, GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies, BMC Genomics (2018).

[90] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, Y. Xie, DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator, in: MICRO, 2017.

[91] G. Kim, N. Chatterjee, M. O'Connor, K. Hsieh, Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs, in: SC, 2017.

[92] V. Seshadri, O. Mutlu, Simple Operations in Memory to Reduce Data Movement, in: Advances in Computers, Volume 106, 2017.

[93] V. Seshadri, Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems, Ph.D. thesis,

Carnegie Mellon University (2016).
[94] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, O. Mutlu, The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption, in: Beyond-CMOS Technologies for Next Generation Computer Design, 2019.
[95] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, O. Mutlu, Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions, arXiv:1802.00320 [cs:AR] (2018).
[96] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, J. Yang, DrAcc: a DRAM Based Accelerator for Accurate CNN Inference, in: DAC, 2018.
[97] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, O. Mutlu, Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost, TACO (2016).
[98] Hybrid Memory Cube Consortium, HMC Specification 1.1 (2013).
[99] Hybrid Memory Cube Consortium, HMC Specification 2.0 (2014).
[100] JEDEC, High Bandwidth Memory (HBM) DRAM, Standard No. JESD235 (2013).
[101] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, O. Mutlu, An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms, in: ISCA, 2013.
[102] S. Ghose, A. G. Yaglikci, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, M. O'Connor, O. Mutlu, What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study, in: SIGMETRICS, 2018.
[103] Y. Wang, A. Tavakkol, L. Orosa, S. Ghose, N. M. Ghiasi, M. Patel, J. Kim, H. Hassan, M. Sadrosadati, O. Mutlu, Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration, in: MICRO, 2018.
[104] A. Das, H. Hassan, O. Mutlu, VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency, in: DAC, 2018.
[105] J. Kim, M. Patel, H. Hassan, L. Orosa, O. Mutlu, Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines, in: ICCD, 2018.
[106] P. J. Denning, T. G. Lewis, Exponential Laws of Computing Growth, Commun. ACM (Jan. 2017).
[107] International Technology Roadmap for Semiconductors (ITRS) (2009).
[108] A. Ailamaki, D. J. DeWitt, M. D. Hill, D. A. Wood, DBMSs on a Modern Processor: Where Does Time Go?, in: VLDB, 1999.
[109] P. A. Boncz, S. Manegold, M. L. Kersten, Database Architecture Optimized for the New Bottleneck: Memory Access, in: VLDB, 1999.
[110] R. Clapp, M. Dimitrov, K. Kumar, V. Viswanathan, T. Willhalm, Quantifying the Performance Impact of Memory Latency and Bandwidth for Big Data Workloads, in: IISWC, 2015.
[111] S. L. Xi, O. Babarinsa, M. Athanassoulis, S. Idreos, Beyond the Wall: Near-Data Processing for Databases, in: DaMoN, 2015.
[112] Y. Umuroglu, D. Morrison, M. Jahre, Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform, in: FPL, 2015.
[113] Q. Xu, H. Jeon, M. Annavaram, Graph Processing on GPUs: Where Are the Bottlenecks?, in: IISWC, 2014.
[114] A. J. Awan, M. Brorsson, V. Vlassov, E. Ayguade, Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server, in: CCBD, 2015.
[115] A. J. Awan, M. Brorsson, V. Vlassov, E. Ayguade, Micro-Architectural Characterization of Apache Spark on Batch and Stream Processing Workloads, in: BDCloud-SocialCom-SustainCom, 2016.
[116] A. Yasin, Y. Ben-Asher, A. Mendelson, Deep-Dive Analysis of the Data Analytics Workload in CloudSuite, in: IISWC, 2014.
[117] T. Moscibroda, O. Mutlu, Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems, in: USENIX Security, 2007.
[118] O. Mutlu, T. Moscibroda, Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors, in: MICRO, 2007.
[119] O. Mutlu, T. Moscibroda, Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems, in: ISCA, 2008.
[120] L. Subramanian, Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management, Ph.D. thesis, Carnegie Mellon University (2015).
[121] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, O. Mutlu, MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems, in: HPCA, 2013.
[122] H. Usui, L. Subramanian, K. Chang, O. Mutlu, DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators, in: ACM TACO, 2016.
[123] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, O. Mutlu, The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory, in: MICRO, 2015.
[124] J. Meza, Q. Wu, S. Kumar, O. Mutlu, Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field, in: DSN, 2015.
[125] SAFARI Research Group, RowHammer – GitHub Repository, `https://github.com/CMU-SAFARI/rowhammer/`.
[126] M. Seaborn and T. Dullien, Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges, `http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`.
[127] M. Seaborn and T. Dullien, Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges, BlackHat (2016).
[128] D. Gruss, C. Maurice, S. Mangard, Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript, CoRR (2015) abs/1507.06955. arXiv:1507.06955.
URL `http://arxiv.org/abs/1507.06955`
[129] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, H. Bos, Flip Feng Shui: Hammering a Needle in the Software Stack, in: USENIX Security, 2016.
[130] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, C. Giuffrida, Drammer: Deterministic Rowhammer Attacks on Mobile Platforms, in: CCS, 2016.
[131] E. Bosman, K. Razavi, H. Bos, C. Giuffrida, Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector, in: SP, 2016.
[132] O. Mutlu, RowHammer, in: Top Picks in Hardware and Embedded Security, 2018.
[133] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, T. W. Keller, Energy Management for Commercial Servers, Computer (Dec 2003).
[134] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, J. B. Carter, Architecting for Power Management: The IBM® POWER7™ Approach, in: HPCA, 2010.
[135] I. Paul, W. Huang, M. Arora, S. Yalamanchili, Harmonia: Balancing Compute and Memory Power in High-Performance GPUs, in: ISCA, 2015.
[136] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, O. Mutlu, The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study, in: SIGMETRICS, 2014.
[137] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, O. Mutlu, AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems, in: DSN, 2015.
[138] S. Khan, D. Lee, O. Mutlu, PARBOR: An Efficient System-Level Technique to Detect Data Dependent Failures in DRAM, in: DSN, 2016.

[139] S. Khan, C. Wilkerson, D. Lee, A. R. Alameldeen, O. Mutlu, A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM, CAL (2016).
[140] M. Patel, J. Kim, O. Mutlu, The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions, in: ISCA, 2017.
[141] S. Khan, C. Wilkerson, Z. Wang, A. Alameldeen, D. Lee, O. Mutlu, Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content, in: MICRO, 2017.
[142] D. Lee, Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity, Ph.D. thesis, Carnegie Mellon University (2016).
[143] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, O. Mutlu, Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives, Proc. IEEE (Sep. 2017).
[144] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, O. Mutlu, Reliability Issues in Flash-Memory-Based Solid-State Drives: Experimental Analysis, Mitigation, Recovery, in: Inside Solid State Drives (SSDs), 2018.
[145] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, O. Mutlu, Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery, arXiv:1711.11427 [cs:AR] (2018).
[146] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, O. Mutlu, MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices, in: FAST, 2018.
[147] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. M. Ghiasi, L. Orosa, J. Gómez-Luna, O. Mutlu, FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives, in: ISCA, 2018.
[148] Y. Cai, NAND Flash Memory: Characterization, Analysis, Modeling, and Mechanisms, Ph.D. thesis, Carnegie Mellon University (2013).
[149] Y. Luo, Architectural Techniques for Improving NAND Flash Memory Reliability, Ph.D. thesis, Carnegie Mellon University (2018).
[150] A. W. Burks, H. H. Goldstine, J. von Neumann, Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946).
[151] G. Kestor, R. Gioiosa, D. J. Kerbyson, A. Hoisie, Quantifying the Energy Cost of Data Movement in Scientific Applications, in: IISWC, 2013.
[152] D. Pandiyan, C.-J. Wu, Quantifying the Energy Cost of Data Movement for Emerging Smart Phone Workloads on Mobile Platforms, in: IISWC, 2014.
[153] J. K. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware?, in: USENIX STC, 1990.
[154] M. Rosenblum, et al., The Impact of Architectural Trends on Operating System Performance, in: SOSP, 1995.
[155] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, O. Mutlu, Improving DRAM Performance by Parallelizing Refreshes with Accesses, in: HPCA, 2014.
[156] Memcached: A High Performance, Distributed Memory Object Caching System, `http://memcached.org`.
[157] MySQL: An Open Source Database, `http://www.mysql.com`.
[158] D. E. Knuth, The Art of Computer Programming, Volume 4 Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams (2009).
[159] H. S. Warren, Hacker's Delight, 2nd Edition, Addison-Wesley Professional, 2012.
[160] C.-Y. Chan, Y. E. Ioannidis, Bitmap Index Design and Evaluation, in: SIGMOD, 1998.
[161] E. O'Neil, P. O'Neil, K. Wu, Bitmap Index Design Choices and Their Performance Implications, in: IDEAS, 2007.
[162] FastBit: An Efficient Compressed Bitmap Index Technology, `https://sdm.lbl.gov/fastbit/`.
[163] K. Wu, E. J. Otoo, A. Shoshani, Compressing Bitmap Indexes for Faster Search Operations, in: SSDBM, 2002.
[164] Y. Li, J. M. Patel, BitWeaving: Fast Scans for Main Memory Data Processing, in: SIGMOD, 2013.
[165] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, Y. He, BitFunnel: Revisiting Signatures for Search, in: SIGIR, 2017.
[166] G. Benson, Y. Hernandez, J. Loving, A Bit-Parallel, General Integer-Scoring Sequence Alignment Algorithm, in: CPM, 2013.
[167] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, O. Mutlu, Shifted Hamming Distance: A Fast and Accurate SIMD-Friendly Filter to Accelerate Alignment Verification in Read Mapping, Bioinformatics (2015).
[168] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, C. Alkan, GateKeeper: A New Hardware Architecture for Accelerating Pre-Alignment in DNA Short Read Mapping, Bioinformatics (2017).
[169] P. Tuyls, H. D. L. Hollmann, J. H. V. Lint, L. Tolhuizen, XOR-Based Visual Cryptography Schemes, Designs, Codes and Cryptography.
[170] J.-W. Han, C.-S. Park, D.-H. Ryu, E.-S. Kim, Optical Image Encryption Based on XOR Operations, SPIE OE (1999).
[171] S. A. Manavski, CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography, in: ICSPC, 2007.
[172] H. Kang, S. Hong, One-Transistor Type DRAM, US Patent 7701751 (2009).
[173] S.-L. Lu, Y.-C. Lin, C.-L. Yang, Improving DRAM Latency with Dynamic Asymmetric Subarray, in: MICRO, 2015.
[174] 6th Generation Intel Core Processor Family Datasheet, `http://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html`.
[175] GeForce GTX 745, `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-745-oem/specifications`.
[176] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, D. Burger, Phase-Change Technology and the Future of Main Memory, IEEE Micro (2010).
[177] B. C. Lee, E. Ipek, O. Mutlu, D. Burger, Phase Change Memory Architecture and the Quest for Scalability, CACM (2010).
[178] P. Zhou, B. Zhao, J. Yang, Y. Zhang, A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology, in: ISCA, 2009.
[179] M. K. Qureshi, V. Srinivasan, J. A. Rivers, Scalable High Performance Main Memory System Using Phase-Change Memory Technology, in: ISCA, 2009.
[180] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, U. C. Weiser, MAGIC—Memristor-Aided Logic, IEEE TCAS II: Express Briefs (2014).
[181] S. Kvatinsky, A. Kolodny, U. C. Weiser, E. G. Friedman, Memristor-Based IMPLY Logic Design Procedure, in: ICCD, 2011.
[182] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, U. C. Weiser, Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies, TVLSI (2014).
[183] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, S. Kvatinsky, Logic Operations in Memory Using a Memristive Akers Array, Microelectronics Journal (2014).
[184] S. Salihoglu, J. Widom, GPS: A Graph Processing System, in: SSDBM, 2013.
[185] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, J. McPherson, From "Think Like a Vertex" to "Think Like a Graph", VLDB Endowment (2013).
[186] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein, Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud, VLDB Endowment (2012).

[187] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, Green-Marl: A DSL for Easy and Efficient Graph Analysis, in: ASPLOS, 2012.
[188] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A System for Large-Scale Graph Processing, in: SIGMOD, 2010.
[189] Harshvardhan, et al., KLA: A New Algorithmic Paradigm for Parallel Graph Computation, in: PACT, 2014.
[190] J. E. Gonzalez, et al., PowerGraph: Distributed Graph-Parallel Computation on Natural Graph, in: OSDI, 2012.
[191] J. Shun, G. E. Blelloch, Ligra: A Lightweight Graph Processing Framework for Shared Memory, in: PPoPP, 2013.
[192] J. Xue, Z. Yang, Z. Qu, S. Hou, Y. Dai, Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing, in: HPDC, 2014.
[193] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. M. Hellerstein, GraphLab: A New Framework for Parallel Machine Learning, arXiv:1006.4990 [cs:LG] (2010).
[194] Google LLC, Chrome Browser, `https://www.google.com/chrome/browser/`.
[195] Google LLC, TensorFlow: Mobile, `https://www.tensorflow.org/mobile/`.
[196] A. Grange, P. de Rivaz, J. Hunt, VP9 Bitstream & Decoding Process Specification, `http://storage.googleapis.com/downloads.webmproject.org/docs/vp9/vp9-bitstream-specification-v0.6-20160331-draft.pdf`.
[197] Hybrid Memory Cube Specification 2.0 (2014).
[198] V. Narasiman, C. J. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, Y. N. Patt, Improving GPU Performance via Large Warps and Two-Level Warp Scheduling, in: MICRO, 2011.
[199] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, C. R. Das, OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, in: ASPLOS, 2013.
[200] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, C. R. Das, Orchestrated Scheduling and Prefetching for GPGPUs, in: ISCA, 2013.
[201] R. Ausavarungnirun, S. Ghose, O. Kayıran, G. H. Loh, C. R. Das, M. T. Kandemir, O. Mutlu, Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance, in: PACT, 2015.
[202] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, O. Mutlu, A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps, in: ISCA, 2015.
[203] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, O. Mutlu, Zorua: A Holistic Approach to Resource Virtualization in GPUs, in: MICRO, 2016.
[204] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, C. R. Das, Exploiting Core Criticality for Enhanced GPU Performance, in: SIGMETRICS, 2016.
[205] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, O. Mutlu, MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency, in: ASPLOS, 2018.
[206] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, O. Mutlu, Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes, in: MICRO, 2017.
[207] R. Ausavarungnirun, Techniques for Shared Resource Management in Systems with Throughput Processors, Ph.D. thesis, Carnegie Mellon University (2017).
[208] M. S. Papamarcos, J. H. Patel, A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories, in: ISCA, 1984.
[209] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, N. Hajinazar, K. Hsieh, K. T. Malladi, H. Zheng, O. Mutlu, LazyPIM: Efficient Support for Cache Coherence in Processing-in-Memory Architectures, arXiv:1706.03162 [cs:AR] (2017).
[210] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. Rossbach, O. Mutlu, MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency, in: ASPLOS, 2018.
[211] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, O. Mutlu, Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes, in: MICRO, 2017.
[212] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, O. Mutlu, Mosaic: Enabling Application-Transparent Support for Multiple Page Sizes in Throughput Processors, SIGOPS Oper. Syst. Rev. (Aug. 2018).
[213] Y. Kim, W. Yang, O. Mutlu, Ramulator: A Fast and Extensible DRAM Simulator, CAL (2015).
[214] SAFARI Research Group, Ramulator: A DRAM Simulator – GitHub Repository, `https://github.com/CMU-SAFARI/ramulator/`.
[215] SAFARI Research Group, SoftMC v1.0 – GitHub Repository, `https://github.com/CMU-SAFARI/SoftMC/`.
[216] N. Binkert, B. Beckman, A. Saidi, G. Black, A. Basu, The gem5 Simulator, CAN (2011).
[217] D. Sanchez, C. Kozyrakis, ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems, in: ISCA, 2013.
[218] J. Power, J. Hestness, M. S. Orr, M. D. Hill, D. A. Wood, gem5-gpu: A Heterogeneous CPU-GPU Simulator, IEEE CAL (Jan 2015).
[219] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, T. M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, in: ISPASS, 2009.
[220] O. Mutlu, Processing Data Where It Makes Sense: Enabling In-Memory Computation, `https://people.inf.ethz.ch/omutlu/pub/onur-MST-Keynote-EnablingInMemoryComputation-October-27-2017-unrolled-FINAL.pptx`, keynote talk at MST (2017).
[221] O. Mutlu, Processing Data Where It Makes Sense in Modern Computing Systems: Enabling In-Memory Computation, `https://people.inf.ethz.ch/omutlu/pub/onur-GWU-EnablingInMemoryComputation-February-15-2019-unrolled-FINAL.pptx`, video available at `https://www.youtube.com/watch?v=oHqsNbxgdzM`, distinguished lecture at George Washington University (2019).