A Modern Primer on Processing in Memory

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

SAFARI Research Group

[a] *ETH Zürich*
[b] *Carnegie Mellon University*
[c] *University of Illinois at Urbana-Champaign*
[d] *King Mongkut's University of Technology North Bangkok*

Abstract

Modern computing systems are overwhelmingly designed to move data to computation. This design choice goes directly against at least three key trends in computing that cause performance, scalability and energy bottlenecks: (1) data access is a key bottleneck as many important applications are increasingly data-intensive, and memory bandwidth and energy do not scale well, (2) energy consumption is a key limiter in almost all computing platforms, especially server and mobile systems, (3) data movement, especially off-chip to on-chip, 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 technology 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, the adoption of error correcting codes inside the latest DRAM chips, proliferation of different main memory standards and chips, specialized for different purposes (e.g., graphics, low-power, high bandwidth, low latency), and the necessity of designing new solutions to serious reliability and security issues, such as the RowHammer phenomenon, are an evidence of this trend.

This chapter discusses recent research that aims to practically enable computation close to data, an approach we call *processing-in-memory* (PIM). PIM places computation mechanisms in or near where the data is stored (i.e., inside the memory chips, in the logic layer of 3D-stacked memory, or in the memory controllers), so that data movement between the computation units and memory is reduced or eliminated. While the general idea of PIM is not new, we discuss motivating trends in applications as well as memory circuits/technology that greatly exacerbate the need for enabling it in modern computing systems. We examine at least two promising new approaches to designing PIM systems to accelerate important data-intensive applications: (1) *processing using memory* by exploiting analog operational properties of DRAM chips to perform massively-parallel operations in memory, with low-cost changes, (2) *processing near memory* by exploiting 3D-stacked memory technology design to provide high memory bandwidth and low memory latency to in-memory logic. 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. We conclude by discussing work on solving key challenges to the practical adoption of PIM.

Keywords: memory systems, data movement, main memory, processing-in-memory, near-data processing, computation-in-memory, processing using memory, processing near memory, 3D-stacked memory, non-volatile memory, energy efficiency, high-performance computing, computer architecture, computing paradigm, emerging technologies, memory scaling, technology scaling, dependable systems, robust systems, hardware security, system security, latency, low-latency computing

Contents

1. Introduction

Main memory, built using the Dynamic Random Access Memory (DRAM) technology, is a major component in nearly all computing systems, including servers, cloud platforms, mobile/embedded devices, and sensor systems. Across all of these systems,the data working set sizes of modern applications are rapidly growing, while the need for fast analysis of such data is increasing. Thus, main memory is becoming an increasingly significant bottleneck across a wide variety of computing systems and applications [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. Alleviating the main memory bottleneck requires the memory capacity, energy, cost, and performance to all scale in an efficient manner across technology generations. Unfortunately, it has become increasingly difficult in recent years, especially the past decade, to scale all of these dimensions [1, 2, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49], and thus the main memory bottleneck has been worsening.

A major reason for the main memory bottleneck is the high energy and latency cost associated with *data movement*. In modern computers, to perform any operation on data that resides in main memory, the processor must retrieve the data from main memory. This requires the memory controller to issue commands to a DRAM module across a relatively slow and power-hungry off-chip bus (known as the *memory channel*). The DRAM module sends the requested data across the memory channel, after which the data is placed in the caches and registers. The CPU can perform computation on the data once the data is in its registers. Data movement from the DRAM to the CPU incurs long latency and consumes a significant amount of energy [7, 50, 51, 52, 53, 54]. These costs are often exacerbated by the fact that much of the data brought into the caches is *not reused* by the CPU [52, 53, 55, 56], 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. The CPU is considered to be the master in the system, and computation is performed only in the processor (and accelerators). In contrast, data storage and communication units, including the main memory, are treated as unintelligent workers that are incapable of computation. As a result of this processor-centric design paradigm, data moves a lot in the system between the computation units and communication/ storage units so that computation can be done on it. With the increasingly *data-centric* nature of contemporary and emerging appli-

cations, the processor-centric design paradigm leads to great inefficiency in performance, energy and cost. For example, most of the real estate within a single compute node is already dedicated to handling data movement and storage (e.g., large caches, memory controllers, interconnects, and main memory) [57, 58, 59, 60, 61], and our recent work shows that more than 62% of the entire system energy of a mobile device is spent on data movement between the processor and the memory hierarchy for widely-used mobile workloads [62].

The large overhead of data movement in modern systems along with technology advances that enable better integration of memory and logic have recently prompted the re-examination of an old idea that we will generally call Processing in Memory (PIM). The key idea is to place computation mechanisms in or near where the data is stored (i.e., inside the memory chips, in the logic layer of 3D-stacked memory, in the memory controllers, or inside large caches), so that data movement between where the computation is done and where the data is stored is reduced or eliminated, compared to contemporary processor-centric systems. Processing-in-memory is also known as *near-data processing* (NDP), enables the ability to perform operations and execute software tasks either using (1) the memory itself, or (2) some form of processing logic (e.g., accelerators, simple cores, reconfigurable logic) inside the memory subsystem.

The idea of PIM has been around for at least five decades [63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80]. However, past efforts were *not* widely adopted for various reasons, including 1) the difficulty of integrating processing elements with DRAM, 2) the lack of critical memory-related scaling challenges that current technology and applications face today, and 3) that the data movement bottleneck was not as critical to system cost, energy and performance 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, 13, 50, 51, 52, 53, 62, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123]). We believe it is crucial to re-examine PIM today with a fresh perspective (i.e., with novel approaches and ideas), by exploiting new memory technologies, with realistic workloads and systems, and with a mindset to ease adoption and feasibility.

In this chapter, we explore two new approaches to enabling processing-in-memory in modern systems. The first approach *minimally changes memory chips* to perform simple yet powerful common operations that the chip is inherently efficient at or could be made efficient at performing [11, 40, 97, 104, 105, 106, 107, 108, 109, 110, 111, 112, 116, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144]. We call this approach *processing using memory* [11, 120, 124, 145]. Some 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, using the analog operational principles of DRAM [108, 109, 111, 112, 120, 124, 125, 145]. Other solutions take advantage of the analog operational principles of emerging non-volatile memory technologies to perform similar bulk operations [104] or other specialized computations like convolutions and matrix multiplications [107, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 146].

The second approach enables PIM in a more general-purpose manner by taking advantage of computation capability in conventional memory controllers [50, 51] or the logic layer(s) of the relatively new *3D-stacked memory technologies* [7, 13, 52, 53, 62, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 98, 99, 100, 101, 103, 113, 114, 115, 117, 118, 119, 147]. We call this general approach *processing near memory* [11]. This approach is especially catalyzed by recent advancements in 3D-stacked memory technologies that include a logic processing layer underneath memory layers [148, 149]. 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 [149]. The TSVs provide much greater *internal* bandwidth within the 3D stack layers than is available externally on the memory channel. Several such 3D-stacked memory architectures, such as the Hybrid Memory Cube [150, 151] and High-Bandwidth Memory [149, 152], include a *logic layer*, where designers can add some processing logic (e.g., accelerators, simple cores, reconfigurable logic) to take advantage of this high internal bandwidth. Future die-stacking technologies, like *monolithic 3D* [153, 154, 155, 156, 157, 158, 159], can amplify the benefits of this approach by greatly improving internal bandwidth and the number of logic layers between memory layers.

Regardless of the approach taken to PIM, there are key practical adoption 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 work paper, along with existing work that addresses these challenges.

Before we describe in detail the two modern approaches to PIM in Section 5, we first describe major trends affecting main memory (Section 2), then demonstrate many reasons why we need to have intelligent memory controllers to enhance memory scaling into the future (Section 3), followed by an analysis of the major shortcomings of the processor-centric computing paradigm which PIM intends to augment, disrupt, and perhaps in some cases displace (Section 4).

2. Major Trends Affecting Main Memory

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 constitutes one of the main pillars of any computing platform, together with 1) the processing elements (or computational elements), which can include CPU cores, GPU cores, accelerators, or reconfigurable devices, and 2) the communication elements, which can include interconnects, network interfaces, and network processing units.

Due to its relatively low cost and low latency, Dynamic Random Access Memory (DRAM) [160] is the predominant data storage technology that is used to build main memory. The growing data working set sizes of modern applications [1, 2, 3, 4, 5, 6, 7, 161, 162, 163, 164, 165] impose an ever-increasing demand for higher DRAM capacity and performance. Unfortunately, DRAM technology scaling is becoming increasingly challenging: it is increasingly difficult to enlarge DRAM chip capacity at low cost while also maintaining maintain DRAM performance, energy efficiency, and reliability [1, 2, 20, 24, 43, 45, 46, 166, 167] Thus, fulfilling the increasing memory needs of modern workloads is becoming increasingly costly and difficult [2, 3, 4, 14, 17, 18, 19, 21, 22, 23, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 42, 44, 45, 46, 47, 49, 52, 53, 86, 121, 168, 169, 170, 171, 172].

If CMOS technology scaling is coming to an end [173], the projections are significantly worse for DRAM technology scaling [174]. DRAM technology scaling affects all major characteristics of DRAM, including capacity, bandwidth, latency, reliability, 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*, i.e., controllers that have intelligence and computation capability to enable better scaling of main memory in terms of all metrics of interest. Such intelligent memory controllers can also more easily pave the way to and be used as a starting 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, as shown by [29], and the latter is slowing down. This trend causes the *memory capacity per core* to drop by approximately 30% every two years [29]. The trend is even worse for *memory bandwidth per core* – in the approximately two decades between 1999 and 2017, DRAM chip storage capacity (for the most commonly-used DDRx chip of the time) has improved approximately 128× while DRAM bandwidth has improved only approximately 20× [31, 32, 40], as shown in Figure 1. In the same period of about two decades, DRAM latency (as measured by the row cycling time) has remained almost constant (i.e., reduced by only 30%, as shown in Figure 1), making it a significant performance bottleneck for many modern workloads, including in-memory databases [62, 97, 112, 175, 176, 177, 178, 179], graph processing [15, 52, 53, 180, 181], data analytics [177, 182, 183, 184], datacenter workloads [4], neural networks [7, 14, 93, 185, 186, 187, 188], and consumer workloads [7]. As low-latency computing is becoming ever more important [1, 2, 3, 12, 13, 189, 190, 191, 192], 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, 25, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202], 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 one bit of data 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, more leaky, and generally more vulnerable to electrical noise and disturbance. 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, as shown in various recent works that study DRAM reliability by analyzing data retention and other reliability issues of modern DRAM chips cell [1, 20, 23, 24, 38, 41, 42, 45, 46, 166, 167, 170, 171, 206, 207, 208, 209, 210, 211]. Hence, mem-

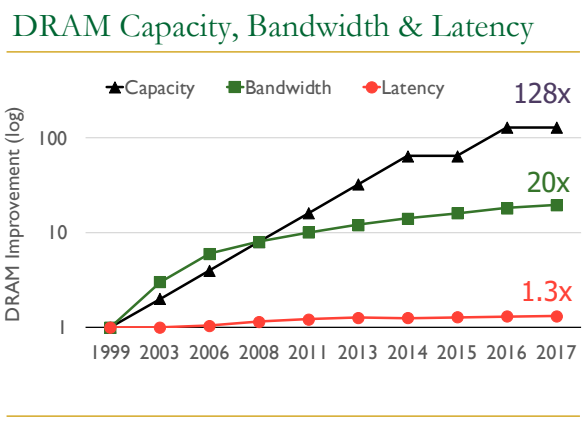

Figure 1: Scaling of DRAM capacity, bandwidth and latency between 1999 and 2017, normalized to the value in 2017. Data depicted for the most common type of DDRx chip of each year. Reproduced from [203]. Originally presented in [31, 204, 205].

ory technology 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, are strongly positively correlated with the density of the chips employed in the servers [212]: the higher the density of the chip used in a server, the more likely the server is to experience a memory error and server failure. 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 lead to new security vulnerabilities. The RowHammer phenomenon [20, 24, 45, 46] 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 [213] can reliably and consistently induce RowHammer errors in vulnerable DRAM modules. The seminal paper that introduced RowHammer [20] 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, as shown by the Figure 2 which depicts the observed RowHammer error vulnerability of DRAM modules manufactured between 2008 and 2014 by all three major DRAM manufacturers A, B, C [20]. A recent technology scaling study [45] of 1580 DRAM chips belonging to three different DRAM types and various different technology node sizes experimentally demonstrated that the RowHammer vulnerability is getting much worse at the circuit level: fewer number of activates to a row can cause bit flips in the most recent chips and recent chips experience higher bit flip rates due to RowHammer. The same work [45] also showed that existing RowHammer mitigation mechanisms will not be effective in future DRAM chips that will be much more vulnerable to RowHammer, and thus RowHammer remains to be an open vulnerability to securely protect against.

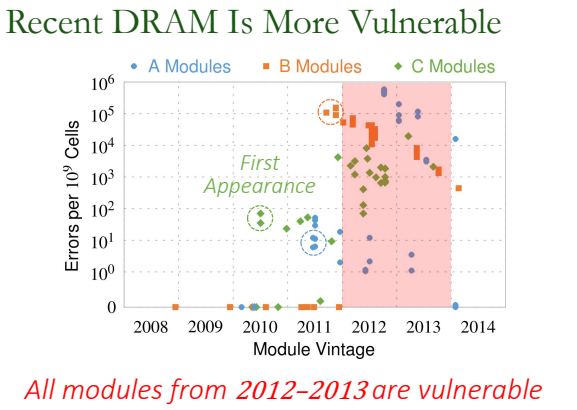

Figure 2: RowHammer vulnerability for DRAM modules manufactured between 2008 and 2014. Reproduced from [214]. Originally presented in [20, 215].

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 in 2015 by researchers from Google Project Zero, who developed a user-level attack that uses RowHammer to gain kernel privileges [216, 217]. 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 [218]; a virtual machine can take over another virtual machine by inducing errors in the victim virtual machine's memory space [219]; a malicious application without permissions can take control of an Android mobile device [220]; or an attacker can gain arbitrary read/write access in a web browser on a Microsoft Windows 10 system [221]. Over the past six

years, many security attacks were developed to exploit RowHammer [216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239]. Very recently, the TRRespass attack [167] showed that existing DRAM chips that are advertised to be RowHammer-resistant, as described by various DRAM vendors [240, 241], are actually vulnerable because these mitigation mechanisms can be circumvented with a new type of RowHammer attack called *many-sided hammering*. For a more detailed treatment of the RowHammer problem and its consequences, as well as its root causes, modeling, and analyses, we refer the reader to [20, 24, 45, 46, 167, 211, 242].

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 [23]), due to its charge-based nature. And, energy consumption of main memory is becoming worse due to three major reasons. First, main memory's capacity, bandwidth, parallelism, and complexity are all increasing, causing energy consumption to naturally increase due to higher amount of dynamic activity and higher overall static power consumption. Second, main memory has remained off the main processing chip and thus did not benefit from many energy reduction mechanisms that come with better integration, even though many other platform components have been integrated into the processing chip and have benefited from the aggressive energy/voltage scaling mechanisms and the low-energy communication substrate on-chip. Third, the difficulties in DRAM technology scaling are making DRAM energy reduction very difficult with technology generations. In fact, some of the mechanisms that are added to DRAM chips to compensate for reliability problems in smaller technology generations, e.g., in-DRAM error correcting codes [17, 41, 170, 171, 243, 244, 245] and higher refresh rates [171, 246, 247, 248], directly increase energy consumption. As a result of these three major issues that make main memory a larger energy bottleneck, the fraction of the entire system power consumed by main memory is increasing over the last two decades. In 2003, Lefurgy et al. [249] showed 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 [34, 43, 250, 251]. 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 [34, 43, 49, 252, 253, 254, 255].

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 energy/power consumption better based on a deep understanding of the DRAM chip 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 modern computing systems 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 key hardware manufacturers in computing industry today, as described in an informative paper written collaboratively by Intel and Samsung engineers on DRAM technology scaling issues [17].

In this section, we give several examples of how an intelligent memory controller can help overcome major scaling challenges of modern DRAM. First, a slightly more intelligent memory controller than today's controllers can prevent the RowHammer vulnerability by probabilistically refreshing rows that are physically adjacent to an activated row, with a very low probability. This solution, called PARA (Probabilistic Adjacent Row Activation) [20] was shown to provide strong, programmable, robust guarantees against RowHammer, with very little power, performance and chip area overheads [20]. It requires a slightly more intelligent memory controller that 1) knows (or that can figure out) the physical adjacency of rows in a DRAM chip, 2) is programmable enough to adjust the probability of adjacent row activation depending on the vulnerability of a chip, and 3) can issue refresh requests to physically-adjacent rows accordingly to the

probability supplied by the system or discovered online. As described by prior work [20, 24, 45, 46, 242], this solution has much lower performance and energy overheads than 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 [46, 248].

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 [23, 166]. A very large fraction of all DRAM cells are strong (i.e., they can retain data for hundreds of seconds), whereas only a small fraction of DRAM cells are weak (i.e., they can retain data for only 64 ms), as demonstrated in Figure 3 [23, 166]. 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 [17], 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 1) identifying the minimum data retention time of each row (during online operation) and 2) refreshing each row at the rate it really requires to be refreshed at or 3) 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 online profiling of DRAM cell retention times and online adjustment of refresh rate on a per-row basis [23, 41, 166, 206, 207, 208, 209, 210], including the works on RAIDR [23, 166], AVATAR [210] and REAPER [41], 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, at the system level. 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 decades, despite the fact that low-latency computing has become even more important during that time.

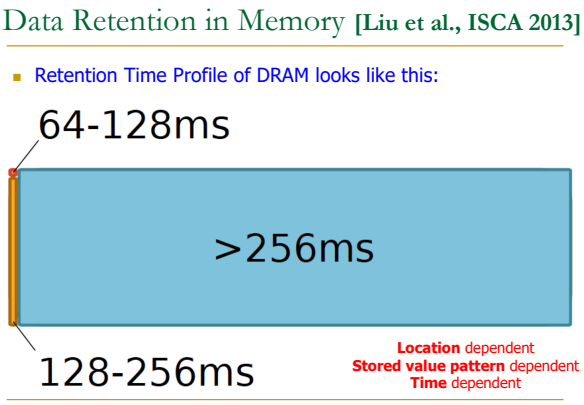

Figure 3: Data retention times of different DRAM cells, represented as a cartoon based on experimental data obtained from real DRAM chips [256]. Reproduced from [214]. Originally presented in [257, 258].

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. Modern DRAM specifications require worst-case 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 acceptable (to satisfy a yield rate) can still be accessed correctly at *worst-case operating temperatures* [31, 33, 35, 42, 44, 47, 169]. However, we find that access latency to cells is very heterogeneous due to variation in operating conditions (e.g., across different temperatures and operating voltage levels), manufacturing process (e.g., across different chips and different parts of a chip), and access patterns (e.g., based on whether or not the cell was recently accessed). We give eight examples of how an intelligent memory controller can exploit the various different types of heterogeneity in access latency.

(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) [33, 259]. 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 [31, 33, 35, 40, 44, 259]. 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% [31, 35, 44].

(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 [34, 40]. 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) [21, 22]. A similar approach is also shown to reduce the performance impact of refresh by enabling parallelization of refresh and access operations to a bank [260].

(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 [32, 42, 110, 121, 169, 259].

(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 stored charge level [39, 47, 168], 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.

(7) Two recent works [172, 261] observe that the latency-reliability tradeoff in modern DRAM devices can be exploited by an intelligent memory controller to 1) generate true random numbers at low latency and high throughput [172], and 2) to evaluate physical unclonable functions quickly using a DRAM device [261]. These works exploit the heterogeneity in the latency-reliability tradeoff of different cells: some cells fail truly randomly and some cells fail very consistently, when accessed with a low latency that violates the timing parameters. The former type of cells are used as true random number generator cells and the latter type of cells can be used as part of the challenge-response space of a DRAM-based physical unclonable function (PUF). An intelligent controller would determine the different types of cells using profiling mechanisms and enable the generation of true random numbers or PUF responses.

(8) An intelligent controller can use application and data characteristics to carefully map data across hybrid memories that consist of multiple different types of memories with different characteristics to maximize the benefits obtained from each memory type while avoiding the downsides of each memory type. Figure 4 depicts an example of such hybrid main memory composed of DRAM and PCM memories, as described by several works [27, 262, 263, 264].

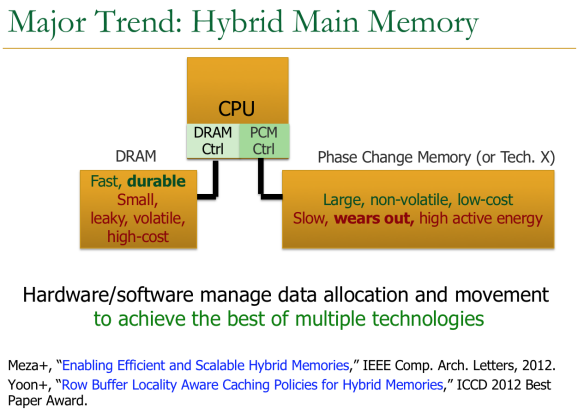

Figure 4: Hybrid main memory. Reproduced from [203]. Originally presented in [27].

Many proposals exist for such intelligent controllers that manage hybrid memories, e.g., [27, 36, 262, 263, 264, 265, 266, 267, 268], indicating that such an intelligent controller can enhance memory scaling by enabling the best of multiple technologies. For example, the idea of *Heterogeneous Reliability Memory* [36] uses an intelligent memory controller that can communicate with both applications and memory devices to map each data element to different types of memories depending on the error vulnerability characteristics of the data element, thereby reducing memory cost. Similarly, EDEN [14] uses a memory controller that can communicate with a neural network application to map different neural net-

work layers to different DRAM partitions with different access latency and voltage parameters, depending on the error tolerance characteristics of each layer, thereby greatly improving energy efficiency and performance of neural network inference tasks. With increasing reliance on hybrid memories as well as increasing heterogeneity within each memory type to solve key memory scaling issues, it has become necessary to have intelligent controllers to manage data allocation, migration, and movement across the different heterogeneous parts.

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 memory controllers that manage the SSDs are designed to incorporate a significant level of intelligence in order to improve both performance and reliability [36, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289]. Figure 5 shows one of our real experimental infrastructures (from [277]) used for the design and evaluation of intelligent flash memory controllers.

Aside: Intelligent Controller for NAND Flash

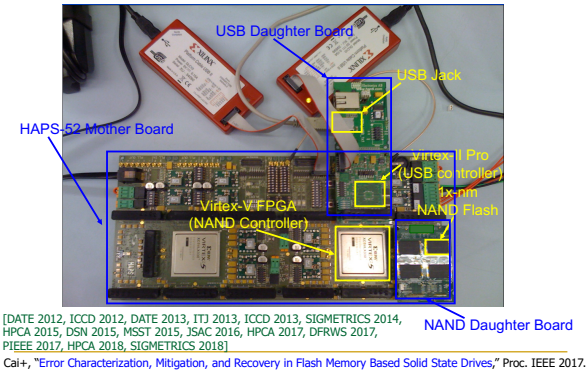

Figure 5: Example of an intelligent flash memory controllers. The figure depicts a picture of one of our real experimental infrastructures (from [277]) used for the design and evaluation of intelligent flash memory controllers. Reproduced from [203].

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, employing sophisticated error correction and recovery techniques to maximize lifetime and minimize error rates, 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 [36, 269, 270, 271, 272, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289]. A comprehensive review of scaling issues of NAND flash memory and related mitigation techniques can be found in [275] (Figure 6) and [269, 270]. We argue that in order to overcome scaling issues in main memory (DRAM), the time has come for main memory controllers to also incorporate significant intelligence. Yet, incorporating sophisticated intelligence in the DRAM controller is more challenging than doing so in a flash controller due to the much lower access latency and much higher access bandwidth of modern DRAM devices.

Aside: Intelligent Controller for NAND Flash

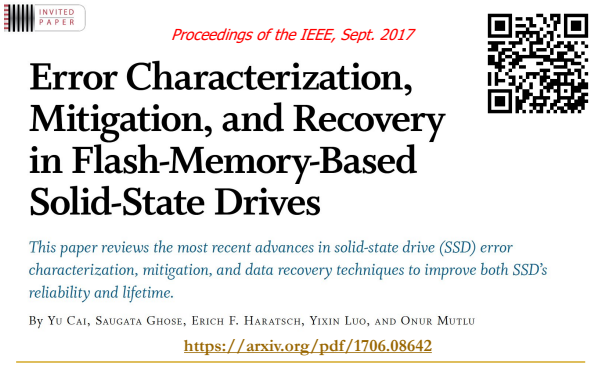

Figure 6: A comprehensive review article on scaling issues of NAND flash memory and related mitigation techniques [275] Reproduced from [203].

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 work has demonstrated that the key reliability, refresh, and latency/energy issues in memory can be mitigated effectively with an intelligent memory controller that intelligently and meticulously manages the many different characteristics of underlying memory chips, which may consist of different types of memory technology. As we discuss in Section 4, such 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 modern and future computing systems.

4. Perils of Processor-Centric Design

As described earlier, a major reason for performance and energy degradation in modern computing systems is the large amount of *data movement* present in the

systems. Such data movement is a natural consequence of the *processor-centric* execution model and design paradigm [290], which creates a dichotomy between computation and memory/storage. The processor-centric design paradigm separates computation capability and memory/storage capability into two completely-separate system components (i.e., the computing unit versus the memory/storage unit) that are connected by long and energy-hungry interconnects: processing is done only in the computing unit, while data is stored in a completely separate place. As a result, data has to continuously move back and forth between the memory/storage unit and the computing unit (e.g., CPU cores or accelerators), for any computation to be performed.

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, 291, 292].

In current computing systems, the CPU (or any accelerator) 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*, which leads to large amounts of energy inefficiency and lost 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 data movement between processor and memory. We briefly describe these next, to demonstrate the sweeping negative impact of data movement in modern computing systems.

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 and high latency to/from main memory. This makes it difficult to send a large number of requests to memory in parallel to enable higher levels of parallelism and to tolerate the long main memory latency. As a result, systems that require higher levels of concurrency and lower latency require much higher cost because they require wider processor-memory interconnects or more processor-memory channels, both of which lead to higher power consumption and higher hardware area overheads [2, 43, 49].

Second, current computing systems employ many sophisticated mechanisms to tolerate the data access from main memory. These mechanisms include complex multi-level cache hierarchies with sophisticated insertion/promotion/eviction policies and sophisticated latency tolerance/hiding mechanisms (e.g., sophisticated caching algorithms at many different caching levels, multiple complex prefetching techniques, high amounts of multithreading, complex and power-hungry out-of-order execution mechanisms). 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. When these components are not effective at improving performance, they result in a net energy waste and latency overheads that hurt the very performance that they are designed to improve. These components significantly increase the complexity of the system. Hence, the architectural and microarchitectural 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 system complexity. As such, we are in a vicious cycle in system design due to the processor-centric design paradigm: 1) data movement between the processor and memory already causes significant energy waste and latency; 2) to tolerate the latency of such data movement, existing systems employ many complex mechanisms whose effectiveness varies depending on the workloads; 3) these complex mechanisms in turn cause additional energy waste and latency overheads. The fundamental cause of this vicious cycle is the processor-centric execution model and design paradigm, and hence breaking out of this vicious cycle requires tackling this fundamental cause by changing the paradigm (to a data-centric one).

Third, the many caches employed in computing systems are not always effective or efficient. Much of the data brought into the caches is *not* reused by the CPU [52, 53, 55, 56, 293], resulting in a large waste of

hardware area and memory bandwidth. For example, 1) random access to memory leads to poor locality, rendering caches almost completely ineffective, 2) strided access to memory where stride is greater than a cache block also renders caches ineffective, 3) even streaming access to memory where all elements in a cache block are used in a consecutive manner is inefficient to handle with large caches because the block is not reused again. There are many such access patterns in a wide variety of modern workloads [49, 52, 53, 55, 56, 199, 293, 294, 295, 296] that render caches either very inefficient or unnecessary, exacerbating the energy waste due to data movement in processor-centric systems.

Fourth, many modern applications, such as graph processing [52, 53] and workloads that operate on sparse data structures, such as sparse linear algebra [15, 297] and sparse neural networks [298, 299, 300], produce random memory access patterns. Figure 7 shows the example of PageRank [301], a graph processing algorithm with frequent random memory accesses and little amount of computation. With such random access patterns, not only the caches but also the main memory bus and the main memory itself are very inefficient, since only a small part of each memory row and cache line retrieved all the way from main memory is actually used by the CPU. Such random accesses are fundamentally difficult to prefetch, rendering prefetchers ineffective. This example demonstrates that modern memory hierarchies are not designed to work well for random memory access patterns that are found in many modern workloads.

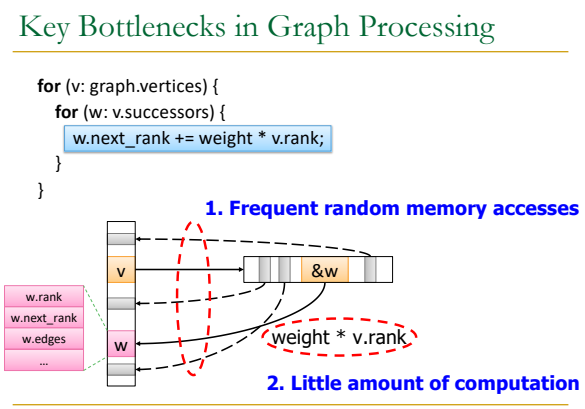

Figure 7: Random memory accesses in the PageRank graph processing algorithm [301]. Reproduced from [214]. Originally depicted in [52, 302].

Fifth, the processor (as well as accelerators) and the main memory are connected to each other via 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 [7, 25, 32, 52, 97, 108], as it greatly exacerbates the cost of data movement. Unfortunately, off-chip interconnect latency and energy are not scaling (i.e., reducing) well with the scaling of technology node generations, which mainly benefits logic [303].

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 an addition operation today. For example, [292] reports that the energy consumption of a memory access is $\sim$ 115$\times$ the energy consumption of an addition operation. Similarly, Figure 8 shows that compares the energy consumed by a DRAM access is $\sim$ 800$\times$ the energy consumption of a double precision addition operation, based on data reported by [304]. As a result, data movement is empirically shown to account for 40% [291], 35% [292], 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.

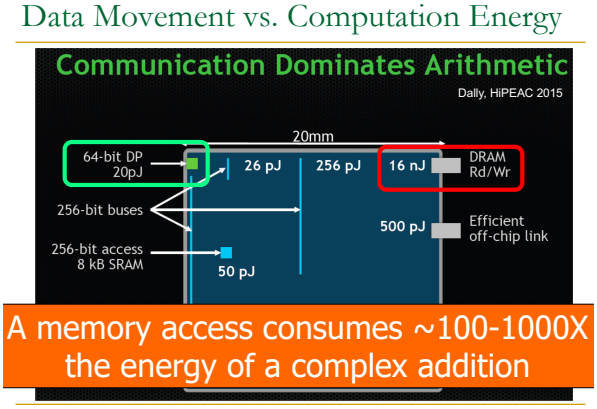

Figure 8: Data movement versus computation energy. The figure depicts the absolute amount of energy spent on various arithmetic and data movement operations, including a double-precision floating point addition and a single DRAM access. Reproduced from [214], based on data reported in [304].

Overcoming all the reasons that cause low performance and large energy inefficiency (as well as high system design complexity) in current computing systems first requires the realization that all of these reasons are caused by the processor-centric design paradigm employed by existing computing systems. As such, a fundamental solution to all of these reasons at the same time requires a paradigm shift [305]. We believe that future computing architectures should become *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 processor (i.e., 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. We refer to this general data-centric execution model and design paradigm as *Processing-in-Memory (PIM)*.

5. Processing-in-Memory (PIM): Technology Enablers and Two Approaches

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 and the processor is essential to make future computing architectures high performance, 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 first describe two new technology enablers for PIM: 1) the emergence of 3D-stacked memories, and 2) the use of byte-addressable memories. These two relatively new main memory technologies provide new opportunities that can make it easier for modern computing systems to introduce and adopt PIM.

Second, we introduce two promising approaches to implementing PIM in modern architectures. The first approach, *processing using memory*, 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, *processing near memory*, 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.

Below, we provide a more detailed general overview of the two approaches, to show that the approaches are more general than what we will describe in more detail. It is important for the reader to keep in mind that the two approaches can be applied to many different types of memory technologies, even though our major focus will be on DRAM, the predominant main memory technology for several decades, in most of this section.

5.1. New Technology Enablers: 3D-Stacked Memory and Non-Volatile Memory

Memory manufacturers are actively developing new approaches for main memory system design, due to the DRAM technology scaling issues we described in detail in Section 2. Two promising technologies are 3D-stacked memory and byte-addressable Non-volatile Memory (NVM), both of which can be exploited to overcome prior barriers to introducing and implementing PIM architectures.

5.1.1. 3D-Stacked Memory Architectures

The first major new approach to main memory design is 3D-stacked memory [52, 149, 150, 151, 152, 306]. In a 3D-stacked memory, multiple layers of memory (typically DRAM in already-existing systems) are stacked on top of each other, as shown in Figure 9. These layers are connected together using vertical through-silicon vias (TSVs) [149, 306]. Using current manufacturing process technologies, thousands of TSVs can be placed within a single 3D-stacked memory chip. As such, the TSVs provide much greater internal memory bandwidth than the narrow memory channel. Examples of 3D-stacked DRAM available commercially include High-Bandwidth Memory (HBM) [149, 152], Wide I/O [307], Wide I/O 2 [308], and the Hybrid Memory Cube (HMC) [151]. Detailed analysis of such 3D-stacked memories and their effects on modern workloads can be found in [49, 148, 149].

In addition to the multiple layers of DRAM, a number of prominent 3D-stacked DRAM architectures, including HBM and HMC, incorporate a logic layer inside the chip [149, 151, 152]. The logic layer is typically the bottommost layer of the chip, and is connected to the same TSVs as the memory layers. The logic layer provides area inside the DRAM chip where architects can implement functionality that interacts with both the processor and the DRAM cells. Currently, manufacturers make

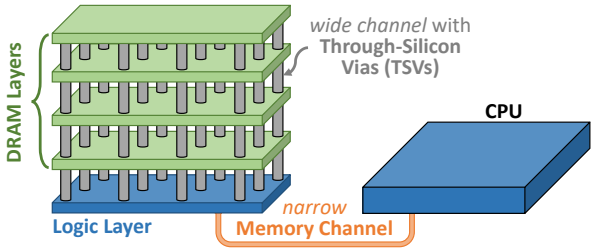

Figure 9: High-level overview of a 3D-stacked DRAM architecture. Reproduced from [309].

limited use of the logic layer and there is significant amount of area the logic layer can provide. This presents a promising opportunity for architects to implement new PIM logic in the available area of the logic layer. We can potentially add a wide range of computational logic (e.g., general-purpose cores, accelerators, reconfigurable architectures, or a combination of all three types of logic) in the logic layer, as long as the added logic meets area, energy, and thermal dissipation constraints, which are important and potentially limiting constraints in 3D-stacked systems.

5.1.2. Non-Volatile Memory (NVM) Architectures

The second major new approach to main memory design is the development of byte-addressable resistive nonvolatile memory (NVM). In order to circumvent DRAM scaling limitations, such as refresh due to charge loss, as much as possible, researchers and manufacturers have been developing new memory devices that can store data at much higher densities than the typical density available in existing DRAM manufacturing process technologies. Manufacturers are exploring at least three types of emerging NVMs to augment or replace DRAM: (1) phase-change memory (PCM) [26, 28, 264, 310, 311, 312, 313]. (2) magnetic RAM (MRAM) [314, 315], and (3) metal-oxide resistive RAM (RRAM) or memristors [316, 317, 318]. All three of these NVM types are expected to provide memory access latencies and energy usage that are competitive with or close enough to DRAM, while enabling much larger capacities per chip and nonvolatility in main memory. Since they are emerging and their designs do not have the long-term ”baggage” other main memories (DRAM) have accumulated, NVMs present architects with an opportunity to redesign how the main memory subsystem operates from the cell and chip levels all the way up to software and algorithms. While it can be relatively difficult to modify the design of DRAM arrays due to the delicacy of DRAM manufacturing process technologies as we approach scaling limitations, NVMs have yet to approach such scaling limitations. As a result, architects can potentially design NVM memory arrays that integrate PIM functionality from the getgo. A promising direction for this functionality is the ability to manipulate NVM cells at the circuit level in order to perform logic operations using the memory cells themselves. A number of recent works have demonstrated that NVM cells can be used to perform a complete family of Boolean logic operations [104, 132, 133, 134, 135, 136], similar to such operations that can be performed in DRAM cells [109, 111, 112, 120, 124]. NVMs have also been shown to perform more sophisticated operations like multiplication [93, 107, 146], which are more difficult to implement in DRAM.

5.2. Two Approaches: Processing Using Memory (PUM) vs. Processing Near Memory (PNM)

Many recent works take advantage of the memory technology innovations that we discuss in Section 5.1 to enable and implement PIM. We find that these works generally take one of two approaches, which are categorized in Table 1: (1) *processing using memory* or (2) *processing near memory*. We briefly describe each approach here. Sections 6 and 7 will provide example approaches and more detail for both.

Table 1: Summary of enabling technologies for the two approaches to PIM used by recent works. Adapted from [309].

Approach	**Enabling Technologies**
Processing Using Memory	SRAM DRAM Phase-change memory (PCM) Magnetic RAM (MRAM) Resistive RAM (RRAM)/memristors
Processing Near Memory	Logic layers in 3D-stacked memory Silicon interposers Logic in memory controllers

Processing using memory (PUM) exploits the existing memory architecture and the operational principles of the memory circuitry to enable operations within main memory with minimal changes. PUM makes use of intrinsic properties and operational principles of the memory cells and cell arrays themselves, by inducing interactions between cells such that the cells and/or cell arrays can perform useful computation. Prior works show that processing using memory is possible using static RAM (SRAM) [105, 106, 130, 131], DRAM [108, 109, 110, 111, 112, 120, 124, 145, 319], PCM [104], MRAM [132, 133, 134], or RRAM/memristive [107, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144] devices. Processing using memory architectures enable a wide range of different functions, such as bulk as well

as finer-grained data copy/initialization [106, 108, 110, 121, 123], bulk bitwise operations (e.g., a complete set of Boolean logic operations) [31, 46, 104, 106, 109, 111, 112, 116, 120, 122, 129, 132, 133, 134, 320], and simple arithmetic operations (e.g., addition, multiplication, implication) [105, 106, 107, 116, 128, 130, 131, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 319].

Processing near memory (PNM) involves adding or integrating PIM logic (e.g., accelerators, simple processing cores, reconfigurable logic) close to or inside the memory (e.g., [7, 13, 50, 51, 52, 53, 62, 81, 82, 83, 84, 86, 87, 88, 91, 92, 93, 94, 95, 96, 98, 99, 100, 102, 103, 113, 115, 117, 118, 119, 178, 179, 321, 322, 323, 324, 325, 326, 327]) Many of these works place PIM logic inside the logic layer of 3D-stacked memories or at the memory controller, but recent advances in silicon interposers (in-package wires that connect directly to the through-silicon vias in a 3D-stacked chip) [118, 147, 152] also allow for separate logic chips to be placed in the same die package as a 3D-stacked memory while still taking advantage of the TSV bandwidth.

Note that more functionality can be potentially integrated into a memory chip using PNM than using PUM, but both approaches can be combined to get even higher benefit from PIM. In Section 6, we provide a detailed overview of PUM within the commodity DRAM technology. In Section 7, we provide a detailed overview of PNM within the 3D-stacked DRAM technology. We note that the described approaches and techniques in Sections 6 and 7 are applicable to other types of technologies as well, with small modifications.

6. Processing Using Memory (PUM)

The PUM approach to 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 the PUM approach [40, 108, 109, 110, 111, 112, 120, 124, 125]. In this section, we first focus on two such designs: RowClone, which enables in-DRAM bulk data movement operations [108] and Ambit, which enables in-DRAM bulk bitwise operations [109, 111, 112, 120]. Then, we describe a low-cost substrate that performs data reorganization for non-unit strided access patterns [97].

6.1. 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, 328, 329] has shown that operating systems and data center workloads spend a significant portion of their time performing bulk data movement operations. For example, a paper by Google shows that close to 5% of the execution time in Google's data center workloads is spent on executing only two data movement function calls, *memset* and *memcopy*. Therefore, accelerating these operations will likely improve system performance and energy efficiency.

We have developed a mechanism called *RowClone* [108], 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 external I/O circuitry by a shared internal bus. Each bank is divided into multiple *subarrays* [21, 108, 260]. 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. Figure 10 illustrates the two steps of RowClone's Fast Parallel Mode. The first step activates source row *A*, which enables the capture of the entire row's data in the row buffer. The second step activates destination row *B*, which enables the copying of the contents of the row buffer into row *B*. Thus, the back-to-back activate in the same subarray enables the copying of source row *A* to destination row *B* by using the row buffer as a temporary buffer for row *A*'s contents. The second mechanism, Pipelined Serial Mode, can transfer an arbitrary number of bytes from a row in one bank to another row in another bank 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,

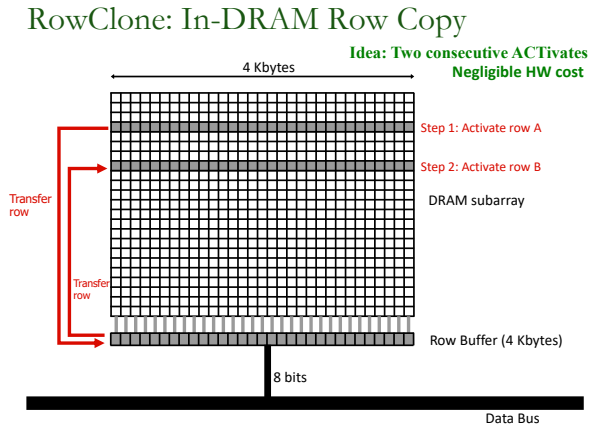

Figure 10: RowClone Fast Parallel Mode. Reproduced from [214].

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) [108]. This reduction directly translates to improvement in performance and energy efficiency of systems running copy or initialization-intensive workloads. Our MICRO 2013 paper [108] shows that the performance of six copy/initialization-intensive benchmarks (including the fork system call, Memcached [330] and a MySQL [331] database) improves between 4% and 66%. For the same six benchmarks, RowClone reduces the energy consumption between 15% and 69%.

Recent works have improved upon the RowClone approach in various ways. First, Low-cost Interlinked SubArrays (LISA) [110] provides mechanisms to enable the rapid transfer of data between one subarray to and adjacent subarray in the same bank, by enhancing the connectivity of subarrays using isolation transistors. LISA reduces inter-subarray copy latency by 9.2× and DRAM energy by 48×, approaching the *intra-subarray* latency and energy improvements of RowClone's Fast Parallel Mode. Second, FIGARO [121] improves upon LISA by enabling fine-grained (i.e., column granularity) data copy across subarrays within a bank using the shared global I/O structures of the bank as an intermediate location. This work shows significant benefit from FIGARO when its principles and techniques are used to build a highly-effective yet low-cost in-DRAM cache. Third, Network-on-Memory (NoM) [123] improves the parallelism of bank-to-bank copy as well as bank read/write operations by providing more connectivity between different banks and chip I/O structures using the logic layer in 3D-stacked memory. Fifth, the ComputeDRAM work [122] shows that one can mimic the effect of RowClone's back-to-back activation mechanism in off-the-shelf DRAM chips by violating the timing parameters such that two wordlines in a subarray are activated back-to-back as in Rowclone. This work shows that such a version of RowClone can operate reliably in a variety of off-the-shelf DRAM chips tested using the SoftMC infrastructure [38, 332]. Sixth, the PINATUBO work [104] show that RowClone can effectively be performed in emerging resistive memory chips, including Phase Change Memory (PCM) [26, 264].

We believe that RowClone provides very low-cost specialized support for a critical and often-used operation: data copy and initialization. In latency-critical systems, such as virtual machines, modern software is written to, as much as possible, avoid large amounts of data copy exactly because data copy is expensive in modern systems (because it goes through the processor over a bandwidth-bottlenecked memory bus). Eliminating copies as much as possible complicates software design, making it less maintainable and readable. If RowClone is implemented in real chips, perhaps the need for avoiding data copies will greatly diminish due to the more-than-an-order-of-magnitude latency reduction of page copy, leading to easier-to-write and easier-to-maintain software. As such, we believe that an idea as simple as RowClone (and the work that builds on it) can have exciting and forward-looking implications on making both systems and software much faster, more efficient and overall better.

6.2. Ambit

In addition to bulk data movement and initialization, many applications make use of *bulk bitwise operations*, i.e., bitwise operations on large bit vectors [333, 334]. Examples of such applications include bitmap indices [335, 336, 337, 338] used in databases, bitwise scan acceleration [339] in databases, accelerated document filtering for web search [340], DNA sequence alignment [13, 115, 190, 191, 192, 341, 342], encryption algorithms [343, 344, 345], graph processing [104], and networking [334]. 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) [109, 111, 112]. 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, depicted in Figure 11, 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. In the initial state, all three rows are closed ❶. In the example of Figure 11, two cells are in the charged state. When the three wordlines are raised simultaneously ❷, charge sharing results in a positive deviation of the bitline. After sense amplification ❸, the sense amplifier drives the bitline to V_{DD}, and as a result, fully charges the three cells. By controlling the initial value of one of the three rows (e.g., C), we can use triple-row activation to perform a bitwise AND or OR of the other two rows, since the bitwise majority function can be expressed as $C(A + B) + \bar{C}(AB)$. The second component, Ambit–NOT, takes advantage of the two inverters that are part of 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 [112] is a row of *dual-contact cells* (a 2-transistor 1-capacitor cell [346, 347]) that connect 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 [112] for details).

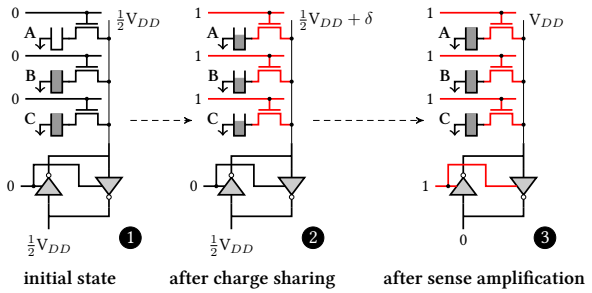

Figure 11: Triple-row activation in Ambit. Reproduced from [112].

Averaged across seven commonly-used bitwise operations (*not, and, or, nand, nor, xor, xnor*), Ambit with 8 DRAM banks improves bulk bitwise operation throughput by 44× compared to an Intel Skylake processor [348], and 32× compared to the NVIDIA GTX 745 GPU [349]. Compared to the DDR3 standard, Ambit reduces energy consumption of these operations by 35× on average. Compared to HMC 2.0 [151], 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.

The Ambit work also shows that porting bitmap-index based databases as well as the BitWeaving database to execute Ambit can greatly improve query latencies. For example, Ambit reduces the end-to-end query latencies by 5.4× to 6.6× for bitmap-based databases, with larger improvements coming from cases where more data needs to be scanned in the database. For the BitWeaving database, which is specifically designed to maximize bitwise operations so that the database can be relatively easily accelerated on modern GPUs, Ambit reduces the end-to-end query latencies by 4× to 12×, again with larger improvements coming from cases where more data needs to be scanned in the database. These results are clearly very promising on two important data-intensive applications.

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) [26, 28, 264, 310, 311, 313], SRAM [105, 106, 130, 131], or specialized computational DRAM [116, 122, 129, 134, 319]. These substrates can perform bulk bitwise operations in a special DRAM array augmented with computational circuitry [116, 128] and in resistive memories [104] like PCM. Substrates similar to Ambit can perform simple arithmetic operations in SRAM [105, 106] and arithmetic and logical operations in memristors [107, 135, 136, 137, 138]. All of these works have shown significant benefits from performing bitwise operations using memory, for a wide variety of applications, including databases, machine learning, graph processing, genome analysis, and using a variety of different memory technologies, including DRAM, SRAM, PCM, memristors.

Recently, the ComputeDRAM work [122] showed that carefully violating timing parameters between activation commands can mimic the triple-row-activation operation of Ambit in some existing off-the-shelf DRAM chips, using the SoftMC infrastructure [38]. Thus, in-DRAM AND and OR operations can be performed in some real off-the-shelf DRAM chips even though clearly such chips are not designed to perform such Ambit operations. This proof-of-concept demonstration shows that the ideas presented in Ambit may not be far from reality: if some existing DRAM chips that are not even designed for in-DRAM bulk bitwise operations can perform such operations, then DRAM chips that are carefully designed for such operations will hopefully be even more capable!

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 sense, amplify and restore the data. Thus, we believe it is very promising to examine the design of both charge-based conventional and emerging resistive memory chips that can incorporate Ambit-like bitwise operations and other types of suitable computation capability. Going forward, it is also critical to research frameworks that can enable ease-of-programming of such substrates such that many algorithms can take advantage of the massive bit-level parallelism offered by Ambit-like PUM substrates.

6.3. 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) [97] 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 DRAM 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 DRAM chips. Figure 12 show an example mapping on four DRAM chips. Adjacent values and/or adjacent pairs of values are swapped. Second, instead of sending separate requests to each chip, the GS-DRAM memory controller communicates a pattern ID to the memory module, as Figure 12 shows. With the pattern ID, each DRAM chip computes the address to be accessed independently via a custom *column translation logic (CTL)* hardware that is part of the DRAM module. This way, the returned cache line contains different values of the strided pattern gathered from different DRAM 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 providing 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. We note that the idea of GS-DRAM is completely independent of memory technology, and thus GS-DRAM can be used in any type of memory module, including DRAM, SRAM, PCM, memristors, STT-MRAM, RRAM.

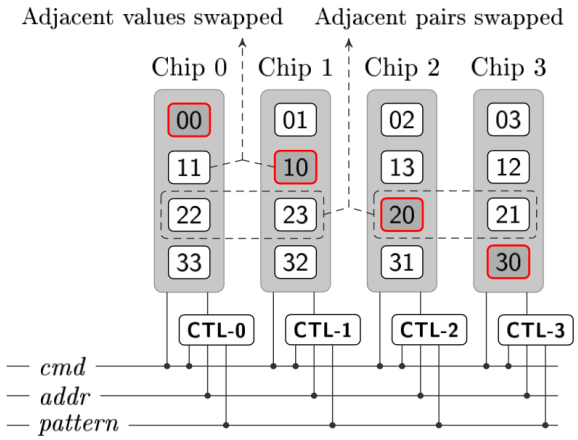

Figure 12: GS-DRAM (Gather-Scatter DRAM) data mapping and chip control overview. CTL refers to Column Translation Logic hardware in the DRAM module. Reproduced from [97].

6.4. In-DRAM Security Primitives

Secure computation is of critical importance in modern computing systems. Therefore, it is important for a PIM system to support fundamental security primitives that enable secure computation and security functions. Doing so would enable PIM systems to execute a wider range of workloads and do so securely. To this end, recent work shows that Processing Using Memory can provide two basic security primitives: by carefully violating DRAM access timing parameters and taking advantage of the resulting characteristics of different DRAM cells (i.e., whether they always/never fail or fail randomly), it is possible to use DRAM to generate Physical Unclonable Functions (PUFs) [261] and true random numbers (TRNs) [172].

PUFs are commonly used in cryptography to identify devices based on the uniqueness of their physical microstructures. DRAM-based PUFs have two key advantages: (1) DRAM is present in many modern computing systems, and (2) DRAM has high capacity and thus can provide many unique identifiers. However, traditional

DRAM PUFs exhibit unacceptably high latencies and are not runtime-accessible. Our recent work, the DRAM Latency PUF [261], proposes a new class of fast, reliable DRAM PUFs that are runtime-accessible, i.e., that can be used during online operation with low latency. The key idea is to reduce DRAM read access latency below the reliable datasheet specifications using software-only system calls. Doing so results in error patterns that reflect the compound effects of manufacturing variations in various DRAM structures (e.g., capacitors, wires, sense amplifiers). Some DRAM cells fail always or not at all, and a combination of a set of such cells can be used to generate a unique identifier for the device. Figure 13 illustrates the key idea of using the pattern of predictable access latency failures in a DRAM subarray to generate a unique DRAM device identifier. An experimental characterization of 223 LPDDR4 DRAM chips from all three major manufacturers shows that these error patterns (1) satisfy runtime-accesible PUF requirements, and (2) are quickly generated (i.e., at 88.2ms) irrespective of operating temperature. The DRAM latency PUF does not require any modification to existing DRAM chips – it only requires an intelligent memory controller that can change timing parameters and identify DRAM regions and cells that can be reliably used as PUFs.

DRAM Latency PUF Key Idea

- A cell's latency failure probability is inherently related to **random process variation** from manufacturing
- We can provide **repeatable and unique device signatures** using latency error patterns

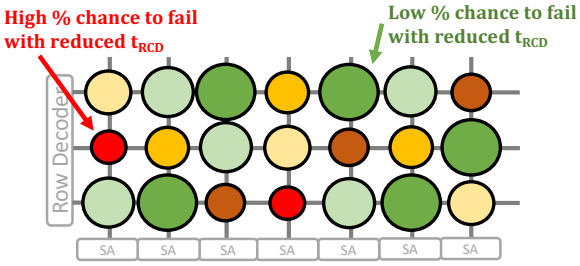

Figure 13: Key idea of DRAM latency PUF. Reproduced from [350].

Intentionally violating DRAM access timing parameters can also be used to generate true random numbers inside DRAM. The technique we propose in [172] decreases the DRAM row activation latency (timing parameter tRCD) below the datasheet specifications to induce read errors, or activation failures. As a result, some DRAM cells, called TRNG (True Random Number Generator) cells, fail truly randomly. By aggregating the resulting data from multiple such TRNG cells, our technique, called D-RaNGe, provides a high-throughput and low-latency TRNG. Figure 14 illustrates the key idea of D-RaNGe: finding and using the TRNG cells in a DRAM subarray to generate true random values.

We demonstrate the effectiveness of D-RaNGe in 282 LPDDR4 devices from the three major manufacturers, and observe that the produced random data remains robust over both time and temperature variation. D-RaNGe (1) successfully passes all NIST statistical tests for randomness, and (2) generates true random numbers with over two orders of magnitude higher throughput than the state-of-the-art DRAM-based TRNG. D-RaNGe does not require any modification to existing DRAM chips – it only requires an intelligent memory controller that can change timing parameters and identify DRAM cells that can be reliably used as TRNG cells.

D-RaNGe Key Idea

- A cell's latency failure probability is inherently related to **random process variation** from manufacturing
- We can extract **random values** by observing DRAM cells' latency failure probabilities

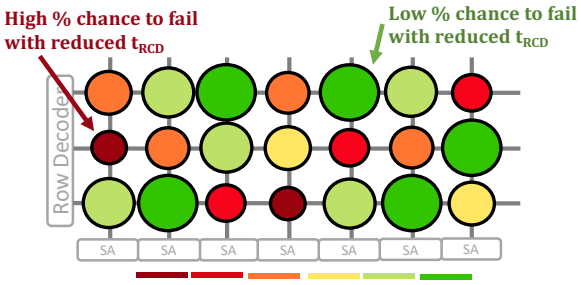

Figure 14: Key idea of D-RaNGe. Reproduced from [351].

D-RaNGe and the DRAM Latency PUF show that commodity DRAM devices can be reliably used to generate true random numbers and unique keys with high throughput, low latency, and low power. As a result, PIM systems can effectively generate true random numbers and unique keys directly using DRAM itself. Doing so can improve the security and privacy of the system: PIM applications can directly generate random numbers or unique keys within DRAM and do not require off-DRAM devices to generate them and transfer them over the CPU to DRAM bus. Thus, random numbers or unique keys are no longer transferred across buses, and security-critical computations can securely happen inside memory, which likely vastly improves the security guarantees of a PIM-enabled system.

7. Processing Near Memory (PNM)

Processing near memory (PNM) involves adding or integrating PIM logic (e.g., accelerators, simple process-

ing cores, reconfigurable logic) close to or inside the memory (e.g., [7, 13, 50, 51, 52, 53, 62, 81, 82, 83, 84, 86, 87, 88, 91, 92, 93, 94, 95, 96, 98, 99, 100, 102, 103, 113, 115, 117, 118, 119, 178, 179, 321, 322, 323, 324, 325, 326, 327]) Many of these works place PIM logic inside the logic layer of 3D-stacked memories [149]. 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 [149], e.g., through-silicon vias. In this section, we discuss several examples of 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.

7.1. Tesseract: Coarse-Grained Application-Level PNM Acceleration of Graph Processing

A promising approach to using PNM is to enable coarse-grained acceleration of *entire applications* that are heavily memory bound. In such a fundamentally coarse-grained (i.e., application-granularity) approach, an entire application is re-written to completely execute on the PNM substrate, potentially using a specialized programming model and specialized architecture/hardware. This approach is especially promising because it can provide the maximum performance and energy benefits achievable from PNM acceleration of a given application, since it enables the customization of the entire PNM system for the application. We believe this approach can be especially promising for widely-used data-intensive applications, such as graph processing, machine learning, databases, media processing, genome analysis.

A popular modern application is large-scale graph processing [113, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361]. 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 PNM accelerator for large-scale graph processing, called Tesseract [52], depicted at a high level in Figure 15. 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 (i.e., computations and temporary values) to data that is to be updated 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 provided 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.

A significant amount of recent research has built upon Tesseract to enable the graph processing PNM system to be much more powerful [324, 325, 326, 327]. Among these, GraphP [325] proposes a new graph partitioning scheme that greatly reduces the costly communication across 3D-stacked memory chips. Better partitioning is also proposed in GraphH [324], together with a reconfigurable double mesh network that provides higher bandwidth across 3D-stacked memory chips. GraphQ [327] employs static and structured communication patterns to eliminate irregular communication, which is one of the key bottlenecks of Tesseract. Hetraph [326] combines memristor-based analog computation units and CMOS-

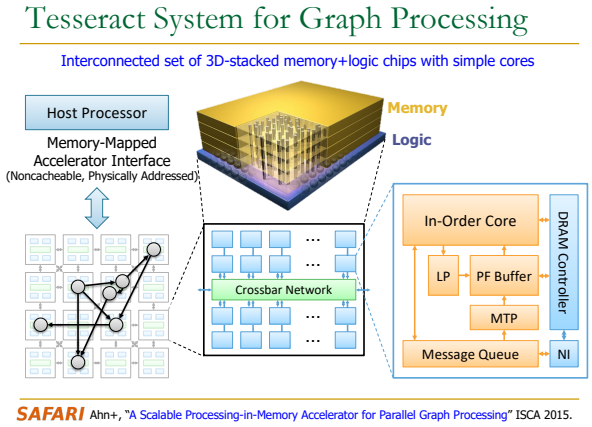

Figure 15: Overview of the Tesseract system for graph processing. Reproduced from [214]. Originally presented in [52, 302].

based digital compute cores on the logic layer of 3D-stacked memory chips, in order to use the most suitable one for each phase of computation. Overall, combining the multiple proposals reported by these works, using the Tesseract-based PNM approach to accelerate graph processing can lead to more than two orders of magnitude improvement both in performance as well as energy efficiency compared to a conventional processor-centric system with high-bandwidth memory. This demonstrates the potential promise of designing an entire PNM system from the ground up completely for an important data-intensive application.

7.2. *Function-Level PNM Acceleration of Mobile Consumer Workloads*

Another promising approach to using PNM, *function-level offloading*, is less intrusive than Tesseract's application-granularity approach described in Section 7.1. This approach can still be coarse-grained since the function that is offloaded to the PNM logic can be potentially arbitarily long. However, the entire application does not need to be re-written. This approach is promising because it can enable easier adoption of PNM while still providing significant benefits. The key question in this approach is which functions in an application should be offloaded for PNM acceleration. Several recent works tackle this question for various applications, e.g., mobile consumer workloads [7], GPGPU workloads [86, 87], graph processing and in-memory database workloads [62, 179], and a wide variety of workloads from many domains [16]. We will discuss function-level PNM acceleration of mobile consumer workloads in this section, focusing on our recent work on the topic [7].

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 mobile consumer applications we study (described in the next paragraph), we find that 62.7% of the total system energy, on average, is spent on data movement across the memory hierarchy [7]. As described before, this large fraction consumed on data movement is directly the result of the processor-centric design paradigm of modern computing systems.

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 [362], which is a very popular browser used in mobile devices and laptops; (2) TensorFlow Mobile [363], 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 [364], and (4) the VP9 video capture engine [364], 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 [151] 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%).

As evident from these results, function-level acceleration provides significant performance and energy benefits, but the benefits are not as high as full application-level offloading and customization of the PNM system, as we have shown for Tesseract in Section 7.1. This is expected since function-level offloading makes much fewer changes to the system and the programming model than application-level offloading, customization and rethinking of the system.

7.3. Programmer-Transparent Function-Level PNM Acceleration of 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. The heavily-multithreaded GPU 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 [365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375], because the limited off-chip pin bandwidth cannot supply enough data to the running threads.

Processing near memory in 3D-stacked memory architectures presents 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. Figure 16 presents an example configuration with a main GPU system connected to four 3D-stacked memories. In the logic layer of each 3D-stacked memory, there are GPU cores (also known as streaming multiprocessors, SMs) connected to memory vault controllers via a crossbar switch. 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) [86] 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 offloading to PIM engines. The compiler estimates the potential memory bandwidth

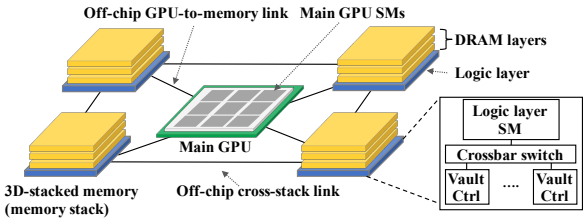

Figure 16: Overview of a PNM GPU system with a powerful main GPU and less powerful logic-layer GPUs distributed across four 3D-stacked memories. Reproduced from [86].

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 dynamic 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 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 as the code that needs it. 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. We find that 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 [87] 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.

7.4. Instruction-Level PNM Acceleration with PIM-Enabled Instructions (PEI)

A finer-grained approach to using PNM is *instruction-level offloading*. With this approach, individual instructions can be offloaded to the PNM engine and accelerated. As we describe below, this fine-grained approach

can have significant benefits in terms of potential adoption since existing processor-centric execution models already operate (i.e., perform computation) at the granularity of individual instructions and all such machinery can be reused to aid offloading to be as seamless as possible with existing programming models and system mechanisms. PIM-Enabled Instructions (PEI) [53] 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 small 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, generated by the compiler or programmer to indicate potentially PIM-offloadable operations in the 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: the programmer may not even be aware that the code is executing on a PIM-capable system and the exact same program containing PEIs can be executed on conventional systems that do not implement PIM.

Figure 17 shows an example architecture that can be used to enable PEIs. In this architecture, a PEI is executed on a PEI Computation Unit (PCU). To enable PEI execution in either the host CPU or in memory, a PCU is added to each host CPU and to each vault in an HMC-like 3D-stacked memory. While the work done in a PCU for a PEI might have required multiple CPU instructions in the baseline CPU-only architecture, the CPU only needs to execute a single PEI instruction, which is sent to a central PEI Management Unit (PMU in Figure 17). The PMU, which is in charge of the Locality-Aware Execution, launches the appropriate PEI operation on one of the PCUs, either on the CPU or in 3D-stacked memory.

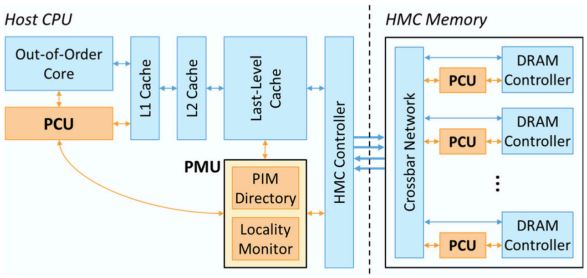

Figure 17: Example architecture for PIM-enabled instructions. Reproduced from [309]. Originally presented in [53, 376].

Examples of PEIs are integer increment, integer minimum, floating-point addition, hash table probing, histogram bin index, Euclidean distance, and dot product [53]. 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, on reasonably large data set sizes.

As such, the benefits provided by the fine-grained PEI approach are quite promising: with minimal changes to the system, performance and energy improve significantly. We therefore believe that the PEI mechanism can ease the adoption of PIM systems going into the future, a key issue we discuss in detail next.

7.5. Function-Level PNM Acceleration of Genome Analysis Workloads

Genome analysis is a critical data-intensive domain that can greatly benefit from acceleration [12, 13, 189, 190, 191, 192, 342, 377, 378, 379], specifically processing-in-memory acceleration. We find that function-level PNM acceleration via algorithm-architecture co-design is especially beneficial for data-intensive genome analysis workloads, as demonstrated in two of our recent works [13, 115].

GRIM-Filter [115] is an in-memory accelerator for genome seed filtering. In order to read the genome (i.e., DNA sequence) of an organism, geneticists often need to reconstruct the genome from small segments of DNA known as reads, as current DNA extraction techniques are unable to extract the entire DNA sequence. A genome

read mapper can perform the reconstruction by matching the reads against a reference genome, and a core part of read mapping is a computationally-expensive dynamic programming algorithm that aligns the reads to the reference genome. One technique to significantly improve the performance and efficiency of read mapping is seed filtering [190, 191, 192, 342, 380], which reduces the number of reference genome seeds (i.e., segments) that a read must be checked against for alignment by quickly eliminating seeds with no probability of matching. GRIM-Filter proposes a state-of-the-art filtering algorithm, and places the entire algorithm inside memory [115].

GRIM-Filter represents the entire reference genome by dividing it into short continuous segments, called *bins*, and performs analyses on metadata associated to each bin. This metadata, represented as a *bitvector*, stores whether or not a particular *token* (a short DNA sequence) is present in the associated bin. Bitvectors are stored in DRAM in column order, such that a DRAM access to a row fetches the bits of the same token across many bitvectors, as the left block of Figure 18 shows. GRIM-Filter places custom logic for each vault in the logic layer of 3D-stacked memory (center block of Figure 18). In each vault, there are multiple *per-bin logic modules* which operate on the bitvector of a single bin. Each logic module consists of an incrementer, accumulator, and comparator, as the right block of Figure 18 shows.

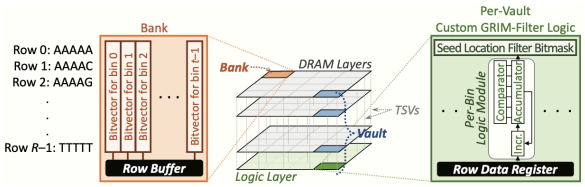

Figure 18: Left block: GRIM-Filter bitvector layout within a DRAM bank. Center block: 3D-stacked DRAM with tightly integrated logic layer stacked underneath with TSVs for high inter-layer data transfer bandwidth. Right block: Custom GRIM-Filter logic placed in the logic layer, for each vault. Reproduced from [115].

GRIM-Filter introduces a communication protocol between the read mapper and the filter. The communication protocol allows GRIM-Filter to be integrated into a full genome read mapper (e.g., FastHASH [380], mrFAST [381], BWA-MEM [382], Minimap2 [383], by allowing (1) the read mapper to notify GRIM-Filter about the DRAM addresses on which to execute customized in-memory filtering operations, (2) GRIM-Filter to notify the read mapper once the filter generates a list of seeds for alignment. Across 10 real genome read sets, GRIM-Filter improves the performance of a full state-of-the-art read mapper by 3.65× over a conventional CPU-only system [115].

In a more recent work [13], we develop an algorithm-architecture co-design to accelerate *approximate string matching (ASM)*, which is used at multiple points during the mapping process of genome analysis. ASM enables read mapping to account for sequencing errors and genetic variations in the reads. Our work, GenASM, is the first ASM acceleration framework for genome sequence analysis. GenASM performs bitvector-based ASM, which can efficiently accelerate multiple steps of genome sequence analysis. We modify the underlying ASM algorithm (Bitap [384, 385]) to significantly increase its parallelism and reduce its memory footprint. We accelerate this modified ASM algorithm, called *GenASM-DC* for Distance Calculation, using an accelerator that performs very efficient Distance Calculation between two input strings. We also develop a novel Bitap-compatible algorithm for *traceback* (i.e., a method to collect information about the different types of alignment errors, or differences, between two input strings), called *GenASM-TB*. Using our modified algorithm and the new GenASM-TB algorithm, we design the first hardware accelerator for Bitap. Figure 19 illustrates a high-level overview of GenASM, depicting the flow of input and intermediate data in the system as well as the communication paths of the two accelerators for GenASM-DC and GenASM-TB. Our hardware accelerator, which is placed in the logic layer of 3D-stacked memory to minimize data movement overheads, consists of specialized systolic-array-based compute units and on-chip SRAMs that are designed to match the rate of computation with memory capacity and bandwidth, resulting in an efficient design whose performance scales linearly as we increase the number of compute units working in parallel. Our detailed performance and energy evaluations demonstrate that GenASM provides significant performance and power benefits for three different use cases in genome sequence analysis, outperforming the best prior hardware accelerators as well as software baselines by one or more orders of magnitude. We believe these results are quite promising and point to the need for further exploration of PIM accelerators in genome analysis.

7.6. Application-Level PNM Acceleration of Time Series Analysis

NATSA [118] is a near-memory processing accelerator for time series analysis. Time series analysis is a powerful technique for extracting and predicting events with applications in epidemiology, genomics, neuroscience, astronomy, environmental sciences, economics,

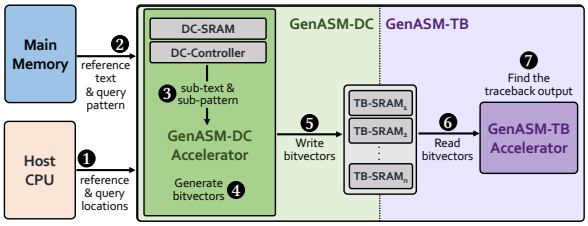

Figure 19: Overview of GenASM. Different components are described in detail in [13]. Figure reproduced from [13].

etc. NATSA implements *matrix profile* [386], the state-of-the-art algorithm for time series analysis fully via PNM. *Matrix profile* operates on large amounts of time series data, but it has low arithmetic intensity. As a result, data movement represents a major performance bottleneck and energy waste, which NATSA alleviates by performing the complete time series analysis processing near memory using specialized accelerators. NATSA places energy-efficient floating point arithmetic processing units (PUs in Figure 20) close to 3D-stacked HBM memory [149, 152], connected via silicon interposers, as Figure 20 shows. NATSA improves performance by up to 14.2× (9.9× on average) and reduces energy by up to 27.2× (19.4× on average) over the state-of-the-art multi-core implementation. NATSA also improves performance by 6.3× and reduces energy by 10.2× over a general-purpose PNM platform with 64 in-order cores.

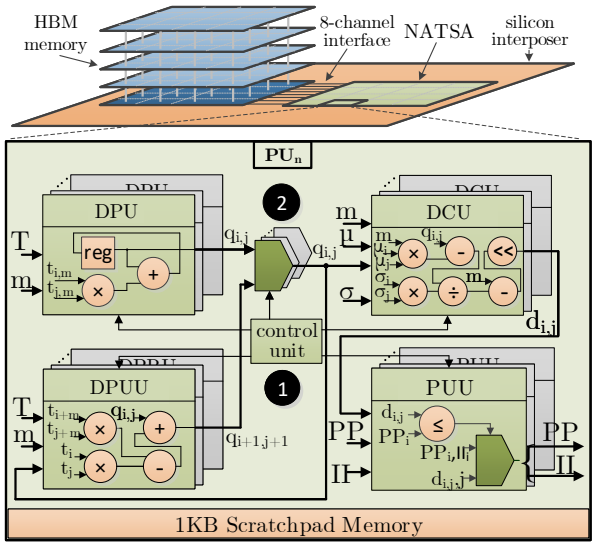

Figure 20: NATSA design and integration near HBM memory. A PU is a NATSA Processing Unit that can do energy-efficient floating point arithmetic for time series analysis. Its components are described in [118]. Figure reproduced from [118].

8. 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. Figure 21 lists some of these key challenges. These challenges must be addressed carefully and systematically 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. We believe future research should examine solutions to these challenges with an open mindset that is keen on enabling adoption, since the widespread success of the PIM paradigm critically depends on effective solutions to these challenges.

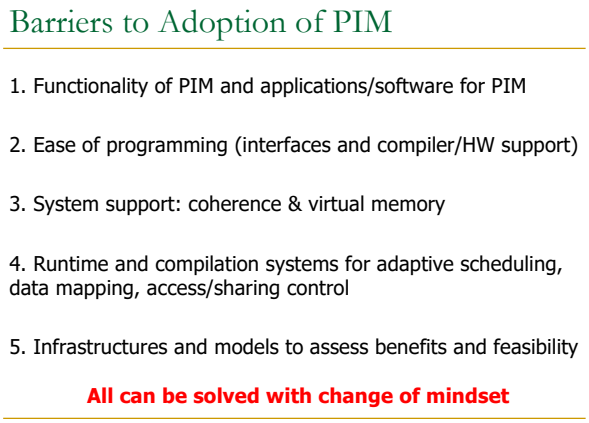

Figure 21: Potential barriers to adoption of PIM. Reproduced from [203, 214].

8.1. Programming Models and Code Generation for PIM

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 [53] work well for offloading fine-grained and 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 [86] 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 [87] examines system-level techniques that decide which GPU application kernels are suitable for PIM execution.

As described in Section 7 with multiple promising examples, different granularities of code offloading in Processing Near Memory architectures have different implications for performance and energy as well as system complexity. These different granularities also have implications on programming and code generation complexity. Adoption-minded solutions should clearly take into account the granularity of code offloading and how a PNM system supports code execution.

Similarly, programming and code generation frameworks for Processing Using Memory approaches like Ambit are also critical for such approaches to become widely adopted. Programming model, compiler and library support for expressing, extracting and generating bulk bitwise operations in a program can greatly help the adoption of in-memory bulk bitwise execution models like Ambit. We believe there is exciting research to do in these directions.

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

8.2. 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, yet much research remains to be done to enable a robust and effective PIM runtime system that can be effective under many conditions.

The first key issue is to identify which portions of an application are suitable for PIM. We call such portions *PIM offloading candidates*. While PIM offloading candidates can be identified manually by a programmer, the identification would require significant programmer effort along with a detailed understanding of the hardware tradeoffs between CPU cores and PIM cores. For architects who are adding custom PIM logic (e.g., fixed-function accelerators, which we call PIM accelerators) to memory, the tradeoffs between CPU cores and PIM accelerators may not be known before determining which portions of the application are PIM offloading candidates, since the PIM accelerators are tailored for the PIM offloading candidates. To alleviate the burden of manually identifying PIM offloading candidates, we develop a systematic toolflow for identifying PIM offloading candidates in an application [7, 62, 179, 321]. This toolflow uses a system that executes the entire application on the CPU to evaluate whether each PIM offloading candidate meets the constraints of the system under consideration. For example, when we evaluate workloads for mobile consumer devices (e.g., Chrome web browser, TensorFlow Mobile, video playback, and video capture) [7], we use hardware performance counters and our energy model to identify candidate functions that could be PIM offloading candidates. A function is a PIM offloading candidate in a mobile consumer device if it meets the following conditions:

1. It consumes a significant fraction (e.g., more than 30%) of the overall workload energy consumption, since energy reduction is a primary objective in mobile systems and workloads.
2. Its data movement consumes a significant fraction (e.g., more than 30%) of the total workload energy to maximize the potential energy benefits of offloading to PIM.
3. It is memory-intensive (e.g., its last-level cache misses per kilo instruction, or MPKI, is greater than 10 [200, 387, 388, 389]), as the energy savings of PIM is higher when more data movement is eliminated.
4. Data movement is the single largest component of the function's energy consumption.

Figure 22 shows two example functions in Google's Mobile TensorFlow machine learning inference framework [363, 390] that are identified to be PIM offloading candidates using the afore-described methodology: *packing/unpacking* and *quantization* [7]. Note that these functions are together responsible for more than 54% of the data movement energy in the examined neural networks for this workload, which spend more than 57% of their execution energy on data movement, as depicted in Figure 22.

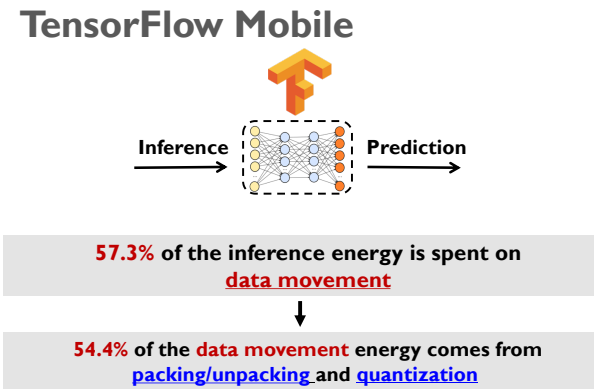

Figure 22: A majority of the data movement energy in TensorFlow Mobile machine learning inference framework [363, 390] is caused by two key functions. Reproduced from [214]. Originally presented in [7, 391].

Some of our other recent works in PIM identify suitable PIM offloading candidates with different granularities. PIM-Enabled Instructions [53] 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. GPU applications also contain several parts that are suitable for offloading to PIM engines [86, 87]. Bulk memory operations (copy, initialization) and bulk bitwise operations are good candidates for Ambit-like processing using DRAM approaches [108, 109, 112, 125], as we discussed earlier. For PUM approaches that can execute more complex operations (e.g., addition, multiplication) using memory, the *operation complexity* (i.e., the latency of an operation for a certain data type) can determine how beneficial offloading to PUM can be compared to CPU execution. A recent analytical model [392] helps to evaluate such offloading tradeoffs in memristor-based PUM [136, 137, 138].

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 [53], we develop a locality-aware scheduling mechanism for PIM-enabled instructions. For GPU-based systems [86, 87], 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 as well as the type of PIM employed in the system. For instance, in order to be able operate on two source arrays inside DRAM with PUM approaches [108, 109, 110, 111, 112, 120, 124, 145, 319], one key issue is how to guarantee the alignment of the two arrays inside the same DRAM subarray. Practical solutions for this issue need to involve both the memory controller and the operating system to enable that arrays aligned in virtual memory can also be physically aligned in DRAM. he programmer and/or the compiler also likely need to carefully annotate and communicate computation patterns on large data blocks so that the system software and the memory controller can cooperatively map the data blocks in an appropriate manner that is amenable to bulk bitwise computation via PUM. Another key issue is how to move partial results generated in one DRAM subarray to other DRAM subarrays to continue the execution with other input operands residing in those subarrays. Several of our recent works [108, 121, 123, 260] propose mechanisms for in-DRAM internal data movement that can facilitate gathering of data in appropriate rows/subarrays/banks in a DRAM chip.

Programmer-transparent data and code mapping mechanisms are especially desirable for PIM adoption. In [86], we present a software/hardware cooperative mechanism to map data and code to several 3D-stacked memory chips 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 can be offloaded from different GPU cores to the PNM GPU cores in 3D-stacked memory chips. Developing new approaches to data/code mapping and scheduling for a wide 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?

8.3. 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 [393]) 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 [52, 53, 86] 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, *CoNDA* [62, 179, 321], that maintains cache coherence between PIM processing logic and CPU cores *without* sending coherence requests for every memory access. Instead, as shown in Figure 23, CoNDA enables efficient coherence by having the PIM logic:

1. *speculatively* acquire coherence permissions for multiple memory operations over a given period of time (which we call *optimistic execution*; ❶ in the figure);
2. *batch* the coherence requests from the multiple memory operations into a set of compressed coherence *signatures* (❷ and ❸);
3. send the signatures to the CPU to determine whether the speculation violated any coherence semantics.

Whenever the CPU receives compressed signatures from the PIM core (e.g., when the PIM kernel finishes), the CPU performs *coherence resolution* (❹), where it checks if any coherence conflicts occurred. If a conflict exists, any dirty cache line in the CPU that caused the conflict is flushed, and the PIM core rolls back and re-executes the code that was optimistically executed.

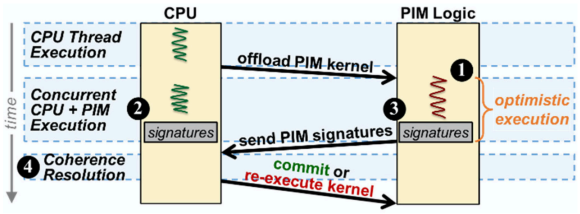

Figure 23: High-level operation of CoNDA, a new coherence mechanism for near-data accelerators, including PNM and PUM. Reproduced from [309]. Originally presented in [179].

As a result of this "lazy" checking of coherence violations, CoNDA approaches near-ideal coherence behavior: the performance and energy consumption of a PIM architecture with CoNDA are, respectively, within 10.4% 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 CoNDA [62, 179, 321] 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 as well as different approaches to PIM.

8.4. 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 [89]. 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 [89], 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 [372, 373, 394].

Looking forward, we recently introduced a fundamentally-new virtual memory framework, the Virtual Block Interface (VBI) [395], which proposes to delegate physical memory management duties completely to the memory controller hardware as well as other specialized hardware. Figure 24 compares VBI to conventional virtual memory at a very high level. Designing VBI-based PIM units that manage memory allocation and address translation can help fundamentally overcome this important virtual memory challenge of PIM systems. We refer the reader to our VBI work [395] for details.

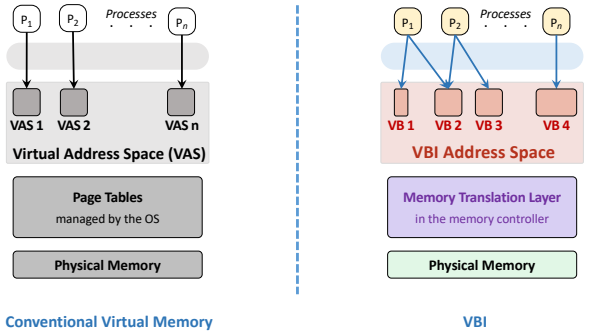

Figure 24: The Virtual Block Interface versus conventional virtual memory. Reproduced from [395].

8.5. 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 [98]. 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 [98]. 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 [98] 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.

8.6. 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 [52, 53], pointer chasing [51, 89], databases [62, 89, 97, 179, 321], consumer workloads [7], time series analysis [118], genome analysis [13, 115], machine learning [7], and GPGPU workloads [86, 87]. However, there is significant room for methodical development of a large-scale PIM benchmark suite, which our very recent work [16] takes the first steps for.

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. In a very recent work that is to appear in 2021 [16], we take the first steps in developing such a methodology and the first benchmark suite for PIM. We refer the reader to that work [16] for detailed description and analyses of the methodology and the new PIM benchmark suite. We believe this work opens up many more steps to extend the methodology and develop other new methodologies for identifying PIM kernels as well as automatic tools (e.g., profilers, compilers, runtime systems) that implement these methodologies, generate optimized code for PIM (potentially with help from programmer annotations), coordinate offloading to PIM cores, etc.

Along these lines, our NAPEL [119] work is an early example of an ML-based performance and energy estimation framework for PNM. NAPEL leverages ensemble learning techniques to generate PNM performance and energy prediction models that are based on microarchitecture parameters and application characteristics. Figure 25 shows the high-level overview of NAPEL training and prediction, the components of which are explained in detail in [119]. Our evaluations show that NAPEL can make fast yet accurate predictions of PIM offloading suitability for previously-unseen applications on general-purpose PNM architectures.

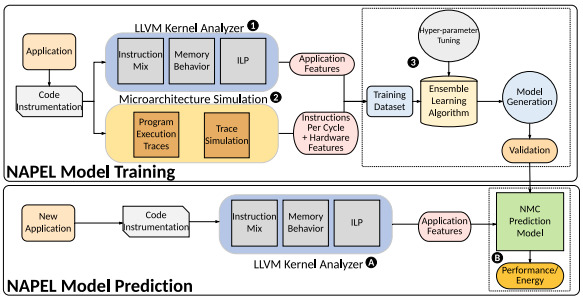

Figure 25: Overview of NAPEL training and prediction. Components are explained in detail in [119]. Figure reproduced from [119].

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 [148, 396], SoftMC [38, 332]) can be combined with full-system simulators (e.g., gem5 [397], zsim [398], gem5-gpu [399], GPGPUSim [400]) 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. A powerful open-source simulation infrastructure that provides such environment is Ramulator-PIM [401], first introduced by our NAPEL framework [119], which combines Ramulator [148, 396] and zsim [398]. Ramulator-PIM can simulate a wide range of configurations of PIM in-order and out-of-order cores and accelerators with different memory technologies.

8.7. Real PIM Hardware Systems and Prototypes

As industry and academia push toward enabling the PIM paradigm, it will be important to also provide real PIM hardware or prototypes. Such hardware can greatly enable and accelerate evaluations of both adoption and research issues in PIM, leading to learnings from real workloads executed on real systems and thus better PIM systems over time. Such real hardware for PIM is very much useful for both PUM and PNM approaches.

We are aware of at least two such real hardware systems. First, ComputeDRAM [122], which is based on the SoftMC memory controller infrastructure [38] can potentially provide the opportunity to test Rowclone (Section 6.1) and Ambit (Section 6.2) PUM approaches on real workloads, albeit likely at reduced reliability since it exploits off-the-shelf DRAM chips, as we discussed in Section 6.2.

Second, the recent UPMEM PIM architecture [402, 403], shown in Figure 26 is the first real-world publicly-available PIM architecture. This PNM system consists of one simple processors (called DRAM Processing Unit, DPU) implemented next to each bank in a DRAM chip. A DPU has high-bandwidth, low-latency, low-energy access to all the data in its corresponding bank. UPMEM has produced real DRAM modules that contain 16 PNM-capable DRAM chips each. Each DRAM chip includes eight 64-MB DRAM banks, each of which has a DPU attached running at a few hundred MHz. A full-blown UPMEM system configuration is expected to soon have 2560 DPUs capable of operating on 160 GB of DRAM memory. We believe the existence of such real PIM hardware can greatly enable and accelerate software and adoption-related research for PIM, specifically PNM architectures, and can set a promising and useful baseline for future research in PNM systems.

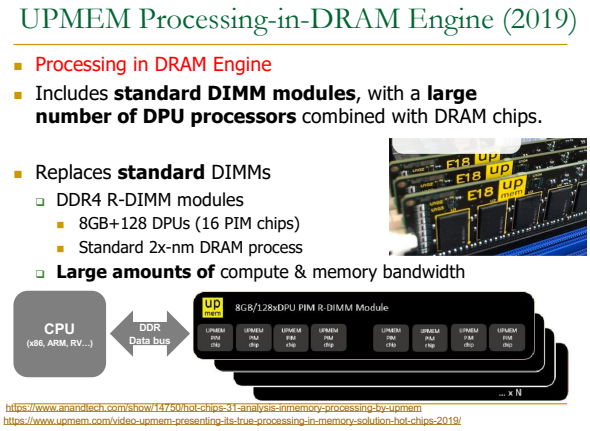

Figure 26: UPMEM PIM architecture and hardware. Reproduced from [214].

8.8. Security Considerations

As a new processing paradigm, PIM introduces new security considerations related to its integration in real-world computing systems. First, there is a need to provide security guarantees in systems with PIM capabilities so that applications that offload code can execute securely in PIM computation units. Naively providing access to PIM computation units for all concurrently-executing applications may lead to potentially unforeseen data leakage and other issues. Second, the ability to perform computation inside or near memory using PIM can enable the opportunity to specialize such computation mechanisms to enhance system security (as briefly discussed in Section 6.4). We cover each of these topics briefly but envision many future ideas related to them in future PIM research and designs.

First, PIM computation units should provide at least as good security primitives as processor-centric computation units of today. This means that there should at least be isolation between concurrently-executing processes on PIM computation units and access control to PIM resources (both data storage and computation units) should be securely managed. Partitioning of computation units, as done in [395], can enable isolation. We believe new approaches to virtualization and cross-layer design that provide extensive hardware management capabilities in the memory controllers, such as the Virtual Block Interface (VBI) [395] or Expressive Memory [404, 405] can not only make PIM security mechanisms easier and more effective to implement but can provide much more

enhanced PIM security mechanisms than existing systems.

As in existing systems, *reliability* and *data integrity* are important in PIM systems, especially in PUM approaches, where memory rows can be frequently activated and deactivated. The RowHammer vulnerability [20, 24, 45, 46, 167, 211] (Section 2) can potentially become exacerbated in PIM systems but it can also be more easily preventable using an intelligent memory controller as in PIM. The cell wearout problem due to endurance limitations in some modern NVM technologies can limit the reliability and thus effectiveness of NVM-based PUM approaches [104, 136, 137, 138, 140] and thus needs to be addressed. Employing in-memory error correcting code (ECC) techniques [170, 171, 406] is likely necessary in future PIM approaches and PIM systems should likely be designed to support ECC techniques to maintain data reliability in the presence of computation mechanisms using/near memory and increasing noise and reliability problems due to technology scaling.

Second, the PIM paradigm enables new opportunities to increase the security and privacy of computations and data, and thus entire computing systems. If data and computation stay within one chip, then the exposure of such data and computation to many attacks will likely be minimized. By eliminating data movement between memory and processor, the PIM paradigm takes a large step towards getting rid of one of the most attacker-exposed type of data movement, i.e., data movement over the main memory bus. Enabling the secure and private execution of computations in PIM systems can therefore potentially enable fundamentally more secure computing systems. This requires providing support for such secure computation, as we discussed earlier in Section 6.4. For example, our afore-described DRAM latency PUF [261] and DRAM latency True Random Number Generator [172] are notable examples of novel in-DRAM security primitives that take advantage of Processing Using Memory that were briefly discussed in Section 6.4. We envision future works on PIM will provide many other security primitives, applications, and use cases.

9. 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 paradigm 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 paradigm 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: PUM (Processing Using Memory) and PNM (Processing Near Memory). First, we show that PUM exploits the existing DRAM architecture and the operational principles of the DRAM circuitry, enabling a number of important and widely-used operations (e.g., memory copy, data initialization, bulk bitwise operations, data reorganization) within DRAM, with minimal changes to DRAM chips. Similar PUM approaches are also applicable to other types of memory chips, and all yield large performance and energy benefits. Second, we demonstrate that PNM can exploit the embedded computation capability in the logic layer of 3D-stacked memory in a variety of ways to provide significant performance improvements and energy savings, across a large range of application domains and computing platforms. Similar PNM approaches are applicable to different types of memories and also to memory controllers.

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 [126, 127]. 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 where data resides (i.e., 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 chapter is a drastically revised and extended version of an earlier article published in 2019 [9]. This chapter also incorporates revised material from another earlier article published in 2019 [11]. The shorter, initial version of this work [9] 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 [407].

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 accumulation of new works and feedback received from many audiences. Recent versions of the talk were delivered as a distinguished lecture at George Washington University in February 2019 [408], as an Invited Talk at ISSCC Special Forum on "Intelligence at the Edge: How Can We Make Machine Learning More Energy Efficient?", as part of the 2019 International Solid State Circuits Conference in February 2019 [409], as a keynote talk at the 29th ACM Great Lakes Symposium on VLSI [410], as a keynote talk at the International Symposium on Advanced Parallel Processing Technology in August 2019 [411], and as a keynote talk at the 37th IEEE International Conference on Computer Design in November 2019 [203].

This article and the associated talks are based on research done over the course of the past nine 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, ASML, 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 and NIH grants, and various awards, including the NSF CAREER Award, the Intel Faculty Honor Program Award, and a number of Google and IBM 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, Communications of the ACM (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] O. Mutlu, S. Ghose, J. Gómez-Luna, R. Ausavarungnirun, Enabling Practical Processing in and near Memory for Data-Intensive Computing, in: DAC, 2019.

[9] O. Mutlu, et al., Processing Data Where It Makes Sense: Enabling In-Memory Computation, MicPro (2019).

[10] O. Mutlu, Intelligent Architectures for Intelligent Machines, `https://people.inf.ethz.ch/omutlu/pub/intelligent-architectures-for-intelligent-machines_keynote-paper_VLSI20.pdf` (2020).

[11] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, O. Mutlu, Processing-in-Memory: A Workload-driven Perspective, IBM JRD (2019).

[12] M. Alser, Z. Bingöl, D. Senol Cali, J. Kim, S. Ghose, C. Alkan, O. Mutlu, Accelerating Genome Analysis: A Primer on an Ongoing Journey, IEEE Micro (2020).

[13] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, et al., GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis, in: MICRO, 2020.

[14] S. Koppula, L. Orosa, A. G. Yağlıkçı, R. Azizi, T. Shahroodi, K. Kanellopoulos, O. Mutlu, EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference using Approximate DRAM, in: MICRO, 2019.

[15] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. Mansouri Ghiasi, T. Shahroodi, J. Gomez-Luna, O. Mutlu, SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations, in: MICRO, 2019.

[16] G. F. Oliveira, J. Gomez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, O. Mutlu, A New Methodology and Open-Source Benchmark Suite for Evaluating Data Movement Bottlenecks: A Near-Data Processing Case Study, in: SIGMETRICS, 2021.

[17] 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.

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

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

[20] 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.

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

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

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

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

[25] 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.

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

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

[28] 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).

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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).

[38] 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.

[39] 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.

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

[41] M. Patel, J. S. Kim, O. Mutlu, The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions, in: ISCA, 2017.

[42] H. Hassan, M. Patel, J. S. Kim, A. G. Yaglikci, N. Vijaykumar, N. M. Ghiasi, S. Ghose, O. Mutlu, CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability, in: ISCA, 2019.

[43] S. Ghose, A. G. Yaglikçi, 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.

[44] J. Kim, M. Patel, H. Hassan, O. Mutlu, Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines, in: ICCD, 2018.

[45] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, O. Mutlu, Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques, in: ISCA, 2020.

[46] O. Mutlu, J. S. Kim, RowHammer: A Retrospective, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[47] Y. Wang, A. Tavakkol, L. Orosa, S. Ghose, N. Mansouri Ghiasi, M. Patel, J. S. Kim, H. Hassan, M. Sadrosadati, O. Mutlu, Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration, in: MICRO, 2018.

[48] O. Mutlu, S. Ghose, R. Ausavarungnirun, Recent Advances in DRAM and Flash Memory Architectures, Invited Journal Issue IPSI Transactions on Internet Research (2018).

[49] S. Ghose, T. Li, N. Hajinazar, D. S. Cali, O. Mutlu, Demystifying Complex Workload-DRAM Interactions: An Experimental Study, in: SIGMETRICS, 2019.

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

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

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

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

[54] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, D. Glasco, GPUs and the Future of Parallel Computing, Micro, IEEE (2011).

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

[56] M. K. Qureshi, M. A. Suleman, Y. N. Patt, Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines, in: HPCA, 2007.
[57] R. Kumar, G. Hinton, A Family of 45nm IA Processors, in: ISSCC, 2009.
[58] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, T. Mattson, A 48-Core IA-32 Message-passing Processor with DVFS in 45nm CMOS, in: ISSCC, 2010.
[59] R. Jotwani, S. Sundaram, S. Kosonocky, A. Schaefer, V. Andrade, G. Constant, A. Novak, S. Naffziger, An x86-64 Core Implemented in 32nm SOI CMOS, in: ISSCC, 2010.
[60] K. Gillespie, H. R. Fair, C. Henrion, R. Jotwani, S. Kosonocky, R. S. Orefice, D. A. Priore, J. White, K. Wilcox, 5.5 Steamroller: An x86-64 Core Implemented in 28nm Bulk CMOS, in: ISSCC, 2014.
[61] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer, E. Chang, J. Bell, M. Co, 3.2 Zen: A Next-generation High-performance x86 Core, in: ISSCC, 2017.
[62] 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).
[63] H. S. Stone, A Logic-in-Memory Computer, IEEE Transactions on Computers (1970).
[64] 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).
[65] D. G. Elliott, W. M. Snelgrove, M. Stumm, Computational RAM: A Memory-SIMD Hybrid and Its Application to DSP, in: CICC, 1992.
[66] P. M. Kogge, EXECUBE–A New Architecture for Scaleable MPPs, in: ICPP, 1994.
[67] M. Gokhale, B. Holmes, K. Iobst, Processing in Memory: The Terasys Massively Parallel PIM Array, IEEE Computer (1995).
[68] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick, A Case for Intelligent RAM, IEEE Micro (1997).
[69] M. Oskin, F. T. Chong, T. Sherwood, Active Pages: A Computation Model for Intelligent Memory, in: ISCA, 1998.
[70] 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.
[71] 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.
[72] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, M. Horowitz, Smart Memories: A Modular Reconfigurable Architecture, in: ISCA, 2000.
[73] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, R. McKenzie, Computational RAM: Implementing Processors in Memory, IEEE Design & Test (1999).
[74] E. Riedel, G. Gibson, C. Faloutsos, Active Storage for Large-scale Data Mining and Multimedia Applications, in: VLDB, 1998.
[75] K. Keeton, D. A. Patterson, J. M. Hellerstein, A Case for Intelligent Disks (IDISKs), SIGMOD Rec. (1998).
[76] 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.
[77] A. Acharya, M. Uysal, J. Saltz, Active Disks: Programming Model, Algorithms and Evaluation, in: ASPLOS, 1998.
[78] M. Jino, J. W. S. Liu, Intelligent Magnetic Bubble Memories, in: ISCA, 1978.
[79] Doty, Greenblatt, Stanley Y.W. Su, Magnetic Bubble Memory Architectures for Supporting Associative Searching of Relational Databases, IEEE Transactions on Computers (1980).
[80] Bongiovanni, Luccio, Maintaining Sorted Files in a Magnetic Bubble Memory, IEEE Transactions on Computers (1980).
[81] 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.
[82] 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.
[83] 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.
[84] 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.
[85] 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.
[86] 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.
[87] 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.
[88] B. Akin, F. Franchetti, J. C. Hoe, Data Reorganization in Memory Using 3D-Stacked DRAM, in: ISCA, 2015.
[89] 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.
[90] O. O. Babarinsa, S. Idreos, JAFAR: Near-Data Processing for Databases, in: SIGMOD, 2015.
[91] J. H. Lee, J. Sim, H. Kim, BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models, in: PACT, 2015.
[92] M. Gao, C. Kozyrakis, HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing, in: HPCA, 2016.
[93] 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.
[94] 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.
[95] D. Kim, J. Kung, S. Chai, S. Yalamanchili, S. Mukhopadhyay, Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory, in: ISCA, 2016.
[96] 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.
[97] 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.
[98] Z. Liu, I. Calciu, M. Herlihy, O. Mutlu, Concurrent Data Structures for Near-Memory Computing, in: SPAA, 2017.
[99] M. Gao, G. Ayers, C. Kozyrakis, Practical Near-Data Processing for In-Memory Analytics Frameworks, in: PACT, 2015.
[100] 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.
[101] 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.
[102] A. Morad, L. Yavits, R. Ginosar, GP-SIMD Processing-in-Memory, ACM TACO (2015).
[103] S. M. Hassan, S. Yalamanchili, S. Mukhopadhyay, Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore, in: MEMSYS, 2015.
[104] 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.
[105] 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.
[106] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, R. Das, Compute Caches, in: HPCA, 2017.
[107] A. Shafiee, A. Nag, N. Muralimanohar, et al., ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars, in: ISCA, 2016.
[108] 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.
[109] 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).
[110] 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.
[111] 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).
[112] 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.
[113] 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.
[114] 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).
[115] 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).
[116] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, Y. Xie, DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator, in: MICRO, 2017.
[117] G. Kim, N. Chatterjee, M. O'Connor, K. Hsieh, Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs, in: SC, 2017.
[118] I. Fernandez, R. Quislant, C. Giannoula, M. Alser, J. Gomez-Luna, E. Gutierrez, O. Plata, O. Mutlu, NATSA: A Near-Data Processing Accelerator for Time Series Analysis, in: ICCD, 2020.
[119] G. Singh, J. Gomez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stujik, O. Mutlu, H. Corporaal, NAPEL: Near-memory Computing Application Performance Prediction via Ensemble Learning, in: DAC, 2019.
[120] V. Seshadri, O. Mutlu, In-DRAM Bulk Bitwise Execution Engine (2020).
[121] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi, et al., FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching, in: MICRO, 2020.
[122] F. Gao, G. Tziantzioulis, D. Wentzlaff, ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs, in: MICRO, 2019.
[123] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, M. Daneshtalab, NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories, CAL (2020).
[124] V. Seshadri, O. Mutlu, Simple Operations in Memory to Reduce Data Movement, in: Advances in Computers, Volume 106, 2017.
[125] V. Seshadri, Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems, Ph.D. thesis, Carnegie Mellon University (2016).
[126] 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.
[127] 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).
[128] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, J. Yang, DrAcc: a DRAM Based Accelerator for Accurate CNN Inference, in: DAC, 2018.
[129] X. Xin, Y. Zhang, J. Yang, ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM, in: HPCA, 2020.
[130] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, R. Das, Neural Cache: Bit-serial In-cache Acceleration of Deep Neural Networks, in: ISCA, 2018.
[131] D. Fujiki, S. Mahlke, R. Das, Duality Cache for Data Parallel Acceleration, in: ISCA, 2019.
[132] S. Angizi, Z. He, D. Fan, PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-efficient Logic Computation, in: DAC, 2018.
[133] S. Angizi, A. S. Rakin, D. Fan, CMP-PIM: An Energy-efficient Comparator-based Processing-in-Memory Neural Network Accelerator, in: DAC, 2018.
[134] S. Angizi, J. Sun, W. Zhang, D. Fan, AlignS: A Processing-in-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM, in: DAC, 2019.
[135] 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).
[136] 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).

[137] S. Kvatinsky, A. Kolodny, U. C. Weiser, E. G. Friedman, Memristor-Based IMPLY Logic Design Procedure, in: ICCD, 2011.
[138] 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).
[139] P.-E. Gaillardon, L. Amaru, A. Siemon, et al., The Programmable Logic-in-Memory (PLiM) Computer, in: DATE, 2016.
[140] D. Bhattacharjee, R. Devadoss, A. Chattopadhyay, ReVAMP: ReRAM based VLIW Architecture for In-memory Computing, in: DATE, 2017.
[141] S. Hamdioui, L. Xie, H. A. D. Nguyen, et al., Memristor Based Computation-in-Memory Architecture for Data-intensive Applications, in: DATE, 2015.
[142] L. Xie, H. A. D. Nguyen, M. Taouil, et al., Fast Boolean Logic Papped on Memristor Crossbar, in: ICCD, 2015.
[143] S. Hamdioui, S. Kvatinsky, e. a. G. Cauwenberghs, Memristor for Computing: Myth or Reality?, in: DATE, 2017.
[144] J. Yu, H. A. D. Nguyen, L. Xie, et al., Memristive Devices for Computation-in-Memory, in: DATE, 2018.
[145] V. Seshadri, O. Mutlu, The Processing Using Memory Paradigm: In-DRAM Bulk Copy, Initialization, Bitwise AND and OR, arXiv:1610.09603 [cs:AR] (2016).
[146] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, D. S. Milojicic, PUMA: A Programmable Ultra-Efficient Memristor-Based Accelerator for Machine Learning Inference, in: ASPLOS, 2019.
[147] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gomez-Luna, S. Stuijk, O. Mutlu, H. Corporaal, NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling, in: FPL, 2020.
[148] Y. Kim, W. Yang, O. Mutlu, Ramulator: A Fast and Extensible DRAM Simulator, CAL (2015).
[149] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, O. Mutlu, Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost, TACO (2016).
[150] Hybrid Memory Cube Consortium, HMC Specification 1.1 (2013).
[151] Hybrid Memory Cube Consortium, HMC Specification 2.0 (2014).
[152] JEDEC, High Bandwidth Memory (HBM) DRAM, Standard No. JESD235 (2013).
[153] B. Gopireddy, J. Torrellas, Designing Vertical Processors in Monolithic 3D, in: ISCA, 2019.
[154] S. Mitra, Abundant-data computing: The N3XT 1,000X, in: VLSI-TSA, 2018.
[155] W. Hwang, W. Wan, S. Mitra, H. . P. Wong, Coming Up N3XT, After 2D Scaling of Si CMOS, in: ISCAS, 2018.
[156] S. Mitra, From Nanodevices to Nanosystems: The N3XT Information Technology, in: E3S, 2015.
[157] D. Rich, A. Bartolo, C. Gilardo, B. Le, H. Li, R. Park, R. M. Radway, M. M. Sabry Aly, H.-S. P. Wong, S. Mitra, Heterogeneous 3D Nano-systems: The N3XT Approach?, 2020.
[158] M. M. Sabry Aly, M. Gao, G. Hills, C. Lee, G. Pitner, M. M. Shulaker, T. F. Wu, M. Asheghi, J. Bokor, F. Franchetti, K. E. Goodson, C. Kozyrakis, I. Markov, K. Olukotun, L. Pileggi, E. Pop, J. Rabaey, C. Ré, H. . P. Wong, S. Mitra, Energy-Efficient Abundant-Data Computing: The N3XT 1,000x, Computer (2015).
[159] M. M. Sabry Aly, T. F. Wu, A. Bartolo, Y. H. Malviya, W. Hwang, G. Hills, I. Markov, M. Wootters, M. M. Shulaker, H. . Philip Wong, S. Mitra, The N3XT Approach to Energy-Efficient Abundant-Data Computing, Proceedings of the IEEE (2019).
[160] R. H. Dennard, Field-effect Transistor Memory, US Patent 3,387,286 (1968).
[161] G. Pekhimenko, T. C. Mowry, O. Mutlu, Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency, in: PACT, 2012.
[162] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework, in: MICRO, 2013.
[163] I. Churin, A. Georgiev, A CAMAC Crate Controller for the IBM PC/XT Family Computers with Built-in Selftest Features, Microprocessing and Microprogramming (1988).
[164] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, T. B. Smith, Memory Expansion Technology (MXT): Software support and performance, IBM Journal of Research and Development (2001).
[165] J. Friedrich, H. Le, W. Starke, J. Stuechli, B. Sinharoy, E. J. Fluhr, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, D. Hogenmiller, F. Malgioglio, R. Nett, R. Puri, P. Restle, D. Shan, Z. T. Deniz, D. Wendel, M. Ziegler, D. Victor, The POWER8TM Processor: Designed for Big Data, Analytics, and Cloud Environments, in: IEEE International Conference on IC Design Technology, 2014.
[166] 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.
[167] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, K. Razavi, TRRespass: Exploiting the Many Sides of Target Row Refresh, in: S&P, 2020.
[168] A. Das, H. Hassan, O. Mutlu, VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency, in: DAC, 2018.
[169] H. Luo, T. Shahroodi, H. Hassan, M. Patel, A. G. Yaglikci, L. Orosa, J. Park, O. Mutlu, CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off, in: ISCA, 2020.
[170] M. Patel, J. S. Kim, H. Hassan, O. Mutlu, Understanding and Modeling On-die Error Correction in Modern DRAM: An Experimental Study using Real Devices, in: DSN, 2019.
[171] M. Patel, J. S. Kim, T. Shahroodi, H. Hassan, O. Mutlu, Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics, in: MICRO, 2020.
[172] J. Kim, M. Patel, H. Hassan, L. Orosa, O. Mutlu, D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput, in: HPCA, 2019.
[173] P. J. Denning, T. G. Lewis, Exponential Laws of Computing Growth, Commun. ACM (Jan. 2017).
[174] International Technology Roadmap for Semiconductors (ITRS) (2009).
[175] A. Ailamaki, D. J. DeWitt, M. D. Hill, D. A. Wood, DBMSs on a Modern Processor: Where Does Time Go?, in: VLDB, 1999.
[176] P. A. Boncz, S. Manegold, M. L. Kersten, Database Architecture Optimized for the New Bottleneck: Memory Access, in: VLDB, 1999.
[177] 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.
[178] S. L. Xi, O. Babarinsa, M. Athanassoulis, S. Idreos, Beyond the Wall: Near-Data Processing for Databases, in: DaMoN, 2015.
[179] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, O. Mutlu, CoNDA: Efficient Cache Coherence Sup-

port for near-Data Accelerators, in: ISCA, 2019.
[180] Y. Umuroglu, D. Morrison, M. Jahre, Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform, in: FPL, 2015.
[181] Q. Xu, H. Jeon, M. Annavaram, Graph Processing on GPUs: Where Are the Bottlenecks?, in: IISWC, 2014.
[182] A. J. Awan, M. Brorsson, V. Vlassov, E. Ayguade, Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server, in: CCBD, 2015.
[183] 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.
[184] A. Yasin, Y. Ben-Asher, A. Mendelson, Deep-Dive Analysis of the Data Analytics Workload in CloudSuite, in: IISWC, 2014.
[185] C. D. Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. Aberger, K. Olukotun, C. Re, High-Accuracy Low-Precision Training, in: arXiv, 2018.
[186] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, W. J. Dally, EIE: Efficient Inference Engine on Compressed Deep Neural Network, in: ISCA, 2016.
[187] Y. Long, T. Na, S. Mukhopadhyay, ReRAM-Based Processing-in-Memory Architecture for Recurrent Neural Network Acceleration, in: TVLSI, 2018.
[188] F. Schuiki, M. Schaffner, F. K. Gürkaynak, L. Benini, A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets, in: arXiv, 2018.
[189] D. Senol, J. Kim, S. Ghose, C. Alkan, O. Mutlu, Nanopore Sequencing Technology and Tools for Genome Assembly: Computational Analysis of the Current State, Bottlenecks and Future Directions, in: Briefings in Bioinformatics (BIB), 2018.
[190] 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).
[191] M. Alser, T. Shahroodi, J. Gomez-Luna, C. Alkan, O. Mutlu, SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs (2020).
[192] M. Alser, H. Hassan, A. Kumar, O. Mutlu, C. Alkan, Shouji: a Fast and Efficient Pre-alignment Filter for Sequence Alignment, Bioinformatics (2019).
[193] T. Moscibroda, O. Mutlu, Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems, in: USENIX Security, 2007.
[194] O. Mutlu, T. Moscibroda, Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors, in: MICRO, 2007.
[195] O. Mutlu, T. Moscibroda, Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems, in: ISCA, 2008.
[196] L. Subramanian, Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management, Ph.D. thesis, Carnegie Mellon University (2015).
[197] 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.
[198] H. Usui, L. Subramanian, K. Chang, O. Mutlu, DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators, TACO (2016).
[199] 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.
[200] Y. Kim, M. Papamichael, O. Mutlu, M. Harchol-Balter, Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior, in: MICRO, 2010.
[201] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding Memory Interference Delay in COTS-based Multi-core Systems, in: RTAS, 2014.
[202] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding and Reducing Memory Interference in COTS-based Multi-core Systems, Real-Time Systems (2016).
[203] O. Mutlu, Processing Data Where It Makes Sense in Modern Computing Systems: Enabling In-Memory Computation, https://people.inf.ethz.ch/omutlu/pub/onur-ICCD-Keynote-EnablingInMemoryComputation-November-19-2019-unrolled.pptx, video available at https://www.youtube.com/watch?v=njX_14584Jw, keynote talk at 37th IEEE International Conference on Computer Design (ICCD), Abu Dhabi, UAE, 19 November 2019. (2019).
[204] K. K. Chang, Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization, https://people.inf.ethz.ch/omutlu/pub/understanding-latency-variation-in-DRAM-chips_kevinchang_sigmetrics16-talk.pdf, conference talk at SIGMETRICS 2016. (2016).
[205] K. K. Chang, Understanding and Improving the Latency of DRAM-Based Memory Systems, https://www.archive.ece.cmu.edu/~safari/thesis/kchang_dissertation.pdf, slides available at https://safari.ethz.ch/safari_public_wp/wp-content/uploads/2018/12/kchang_defense_slides.pptx (2016).
[206] 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.
[207] S. Khan, D. Lee, O. Mutlu, PARBOR: An Efficient System-Level Technique to Detect Data Dependent Failures in DRAM, in: DSN, 2016.
[208] 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).
[209] 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.
[210] 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.
[211] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, O. Mutlu, Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers, in: S&P, 2020.
[212] 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.
[213] SAFARI Research Group, RowHammer – GitHub Repository, https://github.com/CMU-SAFARI/rowhammer/.
[214] O. Mutlu, Intelligent Architectures for Intelligent Machines, https://people.inf.ethz.ch/omutlu/pub/onur-NSF-PIM-KeynoteTalk-IntelligentArchitecturesForIntelligentMachines-October-26-2020-final.pptx, video available at https://www.youtube.com/watch?v=2N-Knx6DHW8, keynote Talk at National Science Foundation Workshop on Processing-In-Memory Technology (NSF-PIM), Virtual, 26 October 2020. (2020).
[215] Y. Kim, Flipping Bits in Memory Without Accessing Them. DRAM Disturbance Errors, https:

//people.inf.ethz.ch/omutlu/pub/dram-row-hammer_kim_talk_isca14.pdf, conference talk at ISCA 2014. (2014).
[216] 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.
[217] M. Seaborn and T. Dullien, Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges, BlackHat (2016).
[218] 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
[219] 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.
[220] 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.
[221] E. Bosman, K. Razavi, H. Bos, C. Giuffrida, Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector, in: S&P, 2016.
[222] Y. Xiao, et al., One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation, in: USENIX Sec., 2016.
[223] D. Gruss, et al., Another Flip in the Wall of Rowhammer Defenses, in: S&P, 2018.
[224] R. Qiao, M. Seaborn, A New Approach for Rowhammer Attacks, in: HOST, 2016.
[225] S. Bhattacharya, D. Mukhopadhyay, Curious Case of RowHammer: Flipping Secret Exponent Bits using Timing Analysis, in: CHES, 2016.
[226] Y. Jang, J. Lee, S. Lee, T. Kim, SGX-Bomb: Locking Down the Processor via Rowhammer Attack, in: SysTEX, 2017.
[227] M. T. Aga, Z. B. Aweke, T. Austin, When Good Protections go Bad: Exploiting anti-DoS Measures to Accelerate Rowhammer Attacks, in: HOST, 2017.
[228] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, S. Mangard, DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks, in: USENIX Security, 2016.
[229] P. Frigo, et al., Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU, in: S&P, 2018.
[230] A. P. Fournaris, L. Pocero Fraile, O. Koufopavlou, Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks, Electronics (2017).
[231] D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, P. Rösler, Attacking Deterministic Signature Schemes using Fault Attacks, in: EuroS&P, 2018.
[232] M. Lipp, et al., Nethammer: Inducing Rowhammer Faults through Network Requests, arxiv.org (2018).
[233] A. Tatar, et al., Throwhammer: Rowhammer Attacks over the Network and Defenses, in: USENIX ATC, 2018.
[234] A. Tatar, C. Giuffrida, H. Bos, K. Razavi, Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer, in: RAID, 2018.
[235] S. Carre, M. Desjardins, A. Facon, S. Guilley, OpenSSL Bellcore's Protection Helps Fault Attack, in: DSD, 2018.
[236] A. Barenghi, L. Breveglieri, N. Izzo, G. Pelosi, Software-only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks, in: IVSW, 2018.
[237] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, G. Karsai, Triggering Rowhammer Hardware Faults on ARM: A Revisit, in: ASHES, 2018.
[238] S. Bhattacharya, D. Mukhopadhyay, Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug, in: Fault Tolerant Architectures for Cryptography and Hardware Security, 2018.
[239] L. Cojocar, K. Razavi, C. Giuffrida, H. Bos, Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against RowHammer Attacks, in: S&P, 2019.
[240] J.-B. Lee, Green Memory Solution, Samsung Electronics, Investor's Forum.
[241] Micron, DDR4 SDRAM Datasheet, p. 380.
[242] O. Mutlu, RowHammer, in: Top Picks in Hardware and Embedded Security, 2018.
[243] T.-Y. Oh, H. Chung, J.-Y. Park, K.-W. Lee, S. Oh, S.-Y. Doo, H.-J. Kim, C. Lee, H.-R. Kim, J.-H. Lee, et al., A 3.2 Gbps/pin 8 gbit 1.0 v LPDDR4 SDRAM with Integrated ECC Engine for sub-1 v DRAM Core Operation, IEEE Journal of Solid-State Circuits (2014).
[244] Micron Technology Inc., ECC Brings Reliability and Power Efficiency to Mobile Devices, Tech. Rep. (2017).
[245] JEDEC, JESD79-5 DDR5 SDRAM standard (2020).
[246] N. Kwak, S.-H. Kim, K. H. Lee, C.-K. Baek, M. S. Jang, Y. Joo, S.-H. Lee, W. Y. Lee, E. Lee, D. Han, et al., 23.3 A 4.8 Gb/s/pin 2Gb LPDDR4 SDRAM with sub-100μA Self-refresh Current for IoT Applications, in: ISSCC, 2017.
[247] H.-J. Kwon, E. Seo, C.-Y. Lee, Y.-H. Seo, G.-H. Han, H.-R. Kim, J.-H. Lee, M.-S. Jang, S.-G. Do, S.-H. Cho, et al., 23.4 An Extremely Low-standby-power 3.733 Gb/s/pin 2Gb LPDDR4 SDRAM for Wearable Devices, in: ISSCC, 2017.
[248] Apple Inc., About the Security Content of Mac EFI Security Update 2015-001, https://support.apple.com/en-us/HT204934 (2015).
[249] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, T. W. Keller, Energy Management for Commercial Servers, Computer (2003).
[250] 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.
[251] I. Paul, W. Huang, M. Arora, S. Yalamanchili, Harmonia: Balancing Compute and Memory Power in High-Performance GPUs, in: ISCA, 2015.
[252] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, O. Mutlu, Memory Power Management via Dynamic Voltage/Frequency Scaling, in: 8th ACM International Conference on Autonomic Computing, 2011.
[253] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, R. Bianchini, Memscale: Active Low-power Modes for Main Memory, in: ASPLOS, 2011.
[254] J. Haj-Yahya, M. Alser, J. Kim, A. G. Yaglıkçı, N. Vijaykumar, E. Rotem, O. Mutlu, SysScale: Exploiting Multi-domain Dynamic Voltage and Frequency Scaling for Energy Efficient Mobile Processors, in: ISCA, 2020.
[255] J. Haj-Yahya, Y. Sazeides, M. Alser, E. Rotem, O. Mutlu, Techniques for Reducing the Connected-Standby Energy Consumption of Mobile Devices, in: HPCA, 2020.
[256] K. Kim, J. Lee, A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs, IEEE Electron Device Letters (2009).
[257] J. Liu, RAIDR: Retention-Aware Intelligent DRAM Refresh, https://people.inf.ethz.ch/omutlu/pub/liu_isca12_talk.pdf, conference talk at ISCA 2012. (2012).
[258] O. Mutlu, An Experimental Study of Data Retention Behavior in Modern DRAM Devices. Implications for Retention Time Profiling Mechanisms, https://people.inf.ethz.ch/omutlu/pub/mutlu_isca13_talk.pdf, conference talk at ISCA 2013. (2013).

[259] D. Lee, Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity, Ph.D. thesis, Carnegie Mellon University (2016).
[260] 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.
[261] J. Kim, M. Patel, H. Hassan, O. Mutlu, The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices, in: HPCA, 2018.
[262] J. Meza, J. Chang, H. Yoon, O. Mutlu, P. Ranganathan, Enabling Efficient and Scalable Hybrid Memories using Fine-granularity DRAM Cache Management, CAL (2012).
[263] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, O. Mutlu, Utility-based Hybrid Memory Management, in: CLUSTER, 2017.
[264] M. K. Qureshi, V. Srinivasan, J. A. Rivers, Scalable High Performance Main Memory System Using Phase-Change Memory Technology, in: ISCA, 2009.
[265] L. E. Ramos, E. Gorbatov, R. Bianchini, Page Placement in Hybrid Memory Systems, in: ICS, 2011.
[266] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, S. Devadas, Banshee: Bandwidth-efficient DRAM Caching via Software/Hardware Cooperation, in: MICRO, 2017.
[267] W. Zhang, T. Li, Exploring Phase Change Memory and 3D Die-stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures, in: PACT, 2009.
[268] S. Song, A. Das, O. Mutlu, N. Kandasamy, Improving Phase Change Memory Performance with Data Content Aware Access, in: ISMM, 2020.
[269] 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.
[270] 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).
[271] Y. Cai, NAND Flash Memory: Characterization, Analysis, Modeling, and Mechanisms, Ph.D. thesis, Carnegie Mellon University (2013).
[272] Y. Luo, Architectural Techniques for Improving NAND Flash Memory Reliability, Ph.D. thesis, Carnegie Mellon University (2018).
[273] 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.
[274] 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.
[275] 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).
[276] Y. Cai, E. F. Haratsch, O. Mutlu, K. Mai, Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis, in: DATE, 2012.
[277] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. S. Unsal, K. Mai, Flash Correct-and-Refresh: Retention-aware Error Management for Increased Flash Memory Lifetime, in: ICCD, 2012.
[278] Y. Cai, O. Mutlu, E. F. Haratsch, K. Mai, Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation, in: ICCD, 2013.
[279] Y. Cai, E. F. Haratsch, O. Mutlu, K. Mai, Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling, in: DATE, 2013.
[280] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, K. Mai, Neighbor-cell Assisted Error Correction for MLC NAND Flash Memories, in: SIGMETRICS, 2014.
[281] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, O. Mutlu, Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery, in: HPCA, 2015.
[282] Y. Cai, Y. Luo, S. Ghose, O. Mutlu, Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery, in: DSN, 2015.
[283] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, E. F. Haratsch, Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques, in: HPCA, 2017.
[284] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, O. Mutlu, HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness, in: HPCA, 2018.
[285] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, O. Mutlu, Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation, in: SIGMETRICS, 2018.
[286] Y. Luo, Y. Cai, S. Ghose, J. Choi, O. Mutlu, WARM: Improving NAND Flash Memory Lifetime with Write-hotness Aware Retention Management, in: MSST, 2015.
[287] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, O. Mutlu, Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory, JSAC (2016).
[288] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Crista, O. S. Unsal, K. Mai, Error Analysis and Retention-Aware Error Management for NAND Flash Memory, Intel Technology Journal (2013).
[289] M. Kim, J. Park, G. Cho, Y. Kim, L. Orosa, O. Mutlu, J. Kim, Evanesco: Architectural Support for Efficient Data Sanitization in Modern Flash-Based Storage Systems, in: ASPLOS, 2020.
[290] A. W. Burks, H. H. Goldstine, J. von Neumann, Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946).
[291] G. Kestor, R. Gioiosa, D. J. Kerbyson, A. Hoisie, Quantifying the Energy Cost of Data Movement in Scientific Applications, in: IISWC, 2013.
[292] D. Pandiyan, C.-J. Wu, Quantifying the Energy Cost of Data Movement for Emerging Smart Phone Workloads on Mobile Platforms, in: IISWC, 2014.
[293] V. Seshadri, O. Mutlu, M. A. Kozuch, T. C. Mowry, The Evicted-address Filter: A Unified Mechanism to Address both Cache Pollution and Thrashing, in: PACT, 2012.
[294] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, D. A. Jimenez, Improving Cache Performance using Read-Write Partitioning, in: HPCA, 2014.
[295] M. K. Qureshi, D. N. Lynch, O. Mutlu, Y. N. Patt, A Case for MLP-aware Cache Replacement, in: ISCA, 2006.
[296] O. Mutlu, H. Kim, Y. N. Patt, Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses, IEEE Transactions on Computers (2006).
[297] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, F. Franchetti, Efficient SPMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization, in: MICRO, 2019.
[298] A. Gondimalla, N. Chesnut, M. Thottethodi, T. Vijaykumar, Sparten: A Sparse Tensor Accelerator for Convolutional Neural Networks, in: MICRO, 2019.
[299] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, C. W. Fletcher, Extensor: An Accelerator for Sparse Tensor Algebra, in: MICRO, 2019.

[300] M. Zhu, T. Zhang, Z. Gu, Y. Xie, Sparse Tensor Core: Algorithm and Hardware Co-design for Vector-wise Sparse Neural Networks on Modern GPUs, in: MICRO, 2019.
[301] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank Citation Ranking: Bringing Order to the Web, Tech. rep., Stanford InfoLab (1999).
[302] J. Ahn, A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing, https://people.inf.ethz.ch/omutlu/pub/tesseract-pim-architecture-for-graph-processing_isca15-talk.pdf, conference talk at ISCA 2015. (2015).
[303] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, A. R. LeBlanc, Design of Ion-implanted MOSFET's with Very Small Physical Dimensions, IEEE Journal of Solid-State Circuits (1974).
[304] W.J. Dally, Challenges for Future Computing Systems, HiPEAC Keynote, 2015.
[305] T. S. Kuhn, The Structure of Scientific Revolutions, 2012.
[306] G. H. Loh, 3D-Stacked Memory Architectures for Multi-Core Processors, in: ISCA, 2008.
[307] JEDEC, Wide I/O Single Data Rate (Wide I/O SDR), Standard No. JESD229 (2011).
[308] JEDEC, Wide I/O 2 (WideIO2), Standard No. JESD229-2 (2014).
[309] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, O. Mutlu, A Workload and Programming Ease Driven Perspective of Processing-in-Memory, arXiv:1907.12947 [cs:AR] (2019).
[310] B. C. Lee, E. Ipek, O. Mutlu, D. Burger, Phase Change Memory Architecture and the Quest for Scalability, CACM (2010).
[311] 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).
[312] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, K. E. Goodson, Phase Change Memory, Proc. IEEE (2010).
[313] P. Zhou, B. Zhao, J. Yang, Y. Zhang, A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology, in: ISCA, 2009.
[314] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, O. Mutlu, Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative, in: ISPASS, 2013.
[315] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, J. Tschanz, STT-RAM Scaling and Retention Failure, Intel Technology Journal (2013).
[316] L. Chua, Memristor—The Missing Circuit Element, IEEE TCT (1971).
[317] D. B. Strukov, G. S. Snider, D. R. Stewart, R. S. Williams, The Missing Memristor Found, Nature (2008).
[318] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, M.-J. Tsai, Metal-Oxide RRAM, Proc. IEEE (2012).
[319] S. Angizi, D. Fan, Graphide: A Graph Processing Accelerator Leveraging In-dram-computing, in: GLSVLSI, 2019.
[320] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, C. J. Radens, Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM), IBM JRD (2002).
[321] 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).
[322] M. Gao, J. Pu, X. Yang, M. Horowitz, C. Kozyrakis, Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory, in: ASPLOS, 2017.
[323] M. P. Drumond Lages De Oliveira, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel Obando, B. Falsafi, B. Grot, D. Pnevmatikatos, The Mondrian Data Engine, in: ISCA, 2017.
[324] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, H. Yang, GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing, IEEE TCAD (2018).
[325] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, X. Qian, GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition, in: HPCA, 2018.
[326] Y. Huang, L. Zheng, P. Yao, J. Zhao, X. Liao, H. Jin, J. Xue, A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing, in: IPDPS, 2020.
[327] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, X. Qian, GraphQ: Scalable PIM-based Graph Processing, in: MICRO, 2019.
[328] J. K. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware?, in: USENIX STC, 1990.
[329] M. Rosenblum, et al., The Impact of Architectural Trends on Operating System Performance, in: SOSP, 1995.
[330] Memcached: A High Performance, Distributed Memory Object Caching System, http://memcached.org.
[331] MySQL: An Open Source Database, http://www.mysql.com.
[332] SAFARI Research Group, SoftMC v1.0 – GitHub Repository, https://github.com/CMU-SAFARI/SoftMC/.
[333] D. E. Knuth, The Art of Computer Programming, Volume 4 Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams (2009).
[334] H. S. Warren, Hacker's Delight, 2nd Edition, Addison-Wesley Professional, 2012.
[335] C.-Y. Chan, Y. E. Ioannidis, Bitmap Index Design and Evaluation, in: SIGMOD, 1998.
[336] E. O'Neil, P. O'Neil, K. Wu, Bitmap Index Design Choices and Their Performance Implications, in: IDEAS, 2007.
[337] FastBit: An Efficient Compressed Bitmap Index Technology, https://sdm.lbl.gov/fastbit/.
[338] K. Wu, E. J. Otoo, A. Shoshani, Compressing Bitmap Indexes for Faster Search Operations, in: SSDBM, 2002.
[339] Y. Li, J. M. Patel, BitWeaving: Fast Scans for Main Memory Data Processing, in: SIGMOD, 2013.
[340] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, Y. He, BitFunnel: Revisiting Signatures for Search, in: SIGIR, 2017.
[341] G. Benson, Y. Hernandez, J. Loving, A Bit-Parallel, General Integer-Scoring Sequence Alignment Algorithm, in: CPM, 2013.
[342] 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).
[343] P. Tuyls, H. D. L. Hollmann, J. H. V. Lint, L. Tolhuizen, XOR-Based Visual Cryptography Schemes, Designs, Codes and Cryptography.
[344] J.-W. Han, C.-S. Park, D.-H. Ryu, E.-S. Kim, Optical Image Encryption Based on XOR Operations, SPIE OE (1999).
[345] S. A. Manavski, CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography, in: ICSPC, 2007.
[346] H. Kang, S. Hong, One-Transistor Type DRAM, US Patent 7701751 (2009).
[347] S.-L. Lu, Y.-C. Lin, C.-L. Yang, Improving DRAM Latency with Dynamic Asymmetric Subarray, in: MICRO, 2015.
[348] 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.
[349] GeForce GTX 745, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-745-oem/specifications.
[350] J. S. Kim, The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern Commodity DRAM Devices, https://people.inf.ethz.ch/omutlu/pub/dram-latency-puf_hpca18_talk.pdf, conference talk at HPCA 2018. (2018).
[351] J. S. Kim, D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput, https://people.inf.ethz.ch/omutlu/pub/drange-dram-latency-based-true-random-number-generator_hpca19-talk.pdf, conference talk at HPCA 2019. (2019).
[352] S. Salihoglu, J. Widom, GPS: A Graph Processing System, in: SSDBM, 2013.
[353] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, J. McPherson, From "Think Like a Vertex" to "Think Like a Graph", VLDB Endowment (2013).
[354] 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).
[355] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, Green-Marl: A DSL for Easy and Efficient Graph Analysis, in: ASPLOS, 2012.
[356] 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.
[357] Harshvardhan, et al., KLA: A New Algorithmic Paradigm for Parallel Graph Computation, in: PACT, 2014.
[358] J. E. Gonzalez, et al., PowerGraph: Distributed Graph-Parallel Computation on Natural Graph, in: OSDI, 2012.
[359] J. Shun, G. E. Blelloch, Ligra: A Lightweight Graph Processing Framework for Shared Memory, in: PPoPP, 2013.
[360] J. Xue, Z. Yang, Z. Qu, S. Hou, Y. Dai, Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing, in: HPDC, 2014.
[361] 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).
[362] Google LLC, Chrome Browser, https://www.google.com/chrome/browser/.
[363] Google LLC, TensorFlow: Mobile, https://www.tensorflow.org/mobile/.
[364] 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.
[365] 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.
[366] 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.
[367] 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.
[368] 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.
[369] 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.
[370] 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.
[371] 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.
[372] 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.
[373] 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.
[374] R. Ausavarungnirun, Techniques for Shared Resource Management in Systems with Throughput Processors, Ph.D. thesis, Carnegie Mellon University (2017).
[375] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, J. Yang, A Framework for Memory Oversubscription Management in Graphics Processing Units, in: ASPLOS, 2019.
[376] J. Ahn, PIM-Enabled Instructions: A Low-Overhead, Locality-Aware PIM Architecture, https://people.inf.ethz.ch/omutlu/pub/pim-enabled-instructons-for-low-overhead-pim_isca15-talk.pdf, conference talk at ISCA 2015. (2015).
[377] O. Mutlu, Accelerating Genome Analysis: A Primer on an Ongoing Journey, https://people.inf.ethz.ch/omutlu/pub/onur-AcceleratingGenomeAnalysis-AACBB-Keynote-Feb-16-2019-FINAL.pptx, video available at https://www.youtube.com/watch?v=hPnSmfwu2-A, keynote talk at 2nd Workshop on Accelerator Architecture in Computational Biology and Bioinformatics (AACBB), Washington, DC, USA, February 2019. (2019).
[378] Y. Turakhia, G. Bejerano, W. J. Dally, Darwin: A Genomics Co-processor Provides up to 15,000x Acceleration on Long Read Assembly, in: ASPLOS, 2018.
[379] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, S. Narayanasamy, Genax: a Genome Sequencing Accelerator, in: ISCA, 2018.
[380] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, C. Alkan, Accelerating Read Mapping with FastHASH, BMC Genomics (2013).
[381] C. Alkan, et al., Personalized Copy Number and Segmental Duplication Maps Using Next-Generation Sequencing, Nature Genetics (2009).
[382] H. Li, R. Durbin, Fast and Accurate Short Read Alignment with Burrows–Wheeler Transform, Bioinformatics (2009).
[383] H. Li, Minimap2: Pairwise Alignment for Nucleotide Sequences, Bioinformatics (2018).
[384] R. Baeza-Yates, G. H. Gonnet, A New Approach to Text Searching, Communications of the ACM (1992).
[385] S. Wu, U. Manber, Fast Text Searching: Allowing Errors, Communications of the ACM (1992).
[386] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, E. Keogh, Matrix profile I: All Pairs Similarity Joins for Time Series: a Unifying View that Includes Motifs, Discords and Shapelets, in: ICDM, 2016.
[387] C. Chou, P. Nair, M. K. Qureshi, Reducing Refresh Power in Mobile Devices with Morphable ECC, in: DSN, 2015.

[388] Y. Kim, D. Han, O. Mutlu, M. Harchol-Balter, ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers, in: HPCA, 2010.

[389] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, T. Moscibroda, Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning, in: MICRO, 2011.

[390] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A System for Large-scale Machine Learning, in: OSDI, 2016.

[391] A. Boroumand, Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks, https://people.inf.ethz.ch/omutlu/pub/Google-consumer-workloads-data-movement-and-PIM_asplos18-talk.pdf, conference talk at ASPLOS 2018. (2018).

[392] K. Korgaonkar, R. Ronen, A. Chattopadhyay, S. Kvatinsky, The Bitlet Model: Defining a Litmus Test for the Bitwise Processing-in-Memory Paradigm (2019). arXiv:1910.10234.

[393] M. S. Papamarcos, J. H. Patel, A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories, in: ISCA, 1984.

[394] 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).

[395] N. Hajinazar, P. Patel, M. Patel, K. Kanellopoulos, S. Ghose, R. Ausavarungnirun, G. F. d. Oliveira Jr, J. Appavoo, V. Seshadri, O. Mutlu, The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework, in: ISCA, 2020.

[396] SAFARI Research Group, Ramulator: A DRAM Simulator – GitHub Repository, https://github.com/CMU-SAFARI/ramulator/.

[397] N. Binkert, B. Beckman, A. Saidi, G. Black, A. Basu, The gem5 Simulator, CAN (2011).

[398] D. Sanchez, C. Kozyrakis, ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems, in: ISCA, 2013.

[399] J. Power, J. Hestness, M. S. Orr, M. D. Hill, D. A. Wood, gem5-gpu: A Heterogeneous CPU-GPU Simulator, CAL (Jan 2015).

[400] 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.

[401] SAFARI Research Group, Ramulator-PIM: A Processing-in-Memory Simulation Framework – GitHub Repository, https://github.com/CMU-SAFARI/ramulator-pim.

[402] UPMEM, Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM accelerator (2018).

[403] F. Devaux, The True Processing In Memory Accelerator, in: Hot Chips, 2019.

[404] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, O. Mutlu, A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory, in: ISCA, 2018.

[405] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, O. Mutlu, The Locality Descriptor: A Holistic Cross-layer Abstraction to Express Data Locality in GPUs, in: ISCA, 2018.

[406] X. Liu, D. Roberts, R. Ausavarungnirun, O. Mutlu, J. Zhao, Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability, in: MICRO, 2019.

[407] 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).

[408] 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).

[409] O. Mutlu, Processing Data Where It Makes Sense in Modern Computing Systems: Enabling In-Memory Computation, https://people.inf.ethz.ch/omutlu/pub/onur-ISSCC2019-talk.pptx, Invited Talk at ISSCC Special Forum on "Intelligence at the Edge: How Can We Make Machine Learning More Energy Efficient?", as part of the 2019 International Solid State Circuits Conference (ISSCC), San Francisco, CA, USA, February 2019. (2019).

[410] O. Mutlu, Processing Data Where It Makes Sense in Modern Computing Systems: Enabling In-Memory Computation, https://people.inf.ethz.ch/omutlu/pub/onur-GLSVLSI-KeynoteTalk-EnablingInMemoryComputation-May-10-2019-unrolled.pptx, keynote Talk at 29th ACM Great Lakes Symposium on VLSI (GLSVLSI), Washington, DC, USA, May 2019. (2019).

[411] O. Mutlu, Processing Data Where It Makes Sense in Modern Computing Systems: Enabling In-Memory Computation, https://people.inf.ethz.ch/omutlu/pub/onur-APPT-Keynote-EnablingInMemoryComputation-August-16-2019-unrolled.pptx, video available at https://www.youtube.com/watch?v=K0OcjxVVhEw, keynote talk at International Symposium on Advanced Parallel Processing Technology (APPT), Tianjin, China, 16 August 2019. (2019).