Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions

SAUGATA GHOSE, KEVIN HSIEH, AMIRALI BOROUMAND, RACHATA AUSAVARUNGNIRUN
Carnegie Mellon University

ONUR MUTLU
ETH Zürich and Carnegie Mellon University

Performance improvements from DRAM technology scaling have been lagging behind the improvements from logic technology scaling for many years. As application demand for main memory continues to grow, DRAM-based main memory is increasingly becoming a larger system bottleneck in terms of both performance and energy consumption. A major reason for poor memory performance and energy efficiency is memory's inability to perform computation. Instead, data stored within DRAM memory *must* be moved into the CPU before any computation can take place. This data movement is costly, as it requires a high latency and consumes significant energy to transfer the data across the pin-limited memory channel. Moreover, the data moved to the CPU is often not reused, and thus does not benefit from being cached within the CPU, which makes it difficult to amortize the overhead of data movement.

Modern 3D-stacked DRAM architectures provide an opportunity to avoid unnecessary data movement between memory and the CPU. These multi-layer architectures include a *logic layer*, where compute logic can be integrated underneath multiple layers of DRAM cell arrays (i.e., the *memory layers*) within the same chip. Architects can take advantage of the logic layer to perform *processing-in-memory* (PIM), or *near-data processing*, where some of the computation is moved from the CPU to the logic layer underneath the memory layer. In a PIM architecture, the logic layer within DRAM has access to the high internal bandwidth available within 3D-stacked DRAM (which is much greater than the bandwidth available in the narrow memory channel between DRAM and the CPU). Thus, PIM architectures can effectively free up valuable bandwidth on the bandwidth-limited memory channel while at the same time reducing system energy consumption.

A number of important issues arise when we add compute logic to DRAM. In particular, logic within DRAM does not have low-latency access to common CPU structures that are essential for modern application execution, such as the virtual memory mechanisms, e.g., the translation lookaside buffer (TLB) or the page table walker, and the cache coherence mechanisms, e.g., the coherence directory. To ease the widespread adoption of PIM, we ideally would like to maintain traditional virtual memory abstractions and the shared memory programming model. This requires *efficient mechanisms* that can provide logic in DRAM with access to virtual memory and cache coherence without having to communicate frequently with the CPU, as off-chip communication between the CPU and DRAM consumes much of the limited bandwidth that PIM aims to avoid using. To this end, we propose and evaluate two general-purpose solutions that can be used by PIM architectures to minimize unnecessary off-chip communication. The first, IMPICA, is an efficient in-memory accelerator for pointer chasing, which can handle address translation entirely within DRAM. The second, LazyPIM, provides coherence support *without* the need to continually communicate with the CPU. We show that both of these mechanisms provide a significant benefit for a number of important memory-intensive applications, thereby both improving performance and reducing energy consumption.

Keywords: processing-in-memory; near-data processing; accelerators; cache coherence; pointer chasing; linked data structures; 3D-stacked memories; virtual memory; address translation; address space protection; speculative execution; shared memory programming model; multithreading; parallel applications; programming ease; energy efficiency

DRAM, the predominant technology used to build main memory, is a major component of modern computer systems. As the data working set sizes of modern applications grow [37, 48, 79, 225], the need for higher memory

capacity and higher memory performance continues to grow as well. However, even though CMOS technology scaling has yet to come to an end, DRAM technology scaling has been unable to keep up with the increasing memory demand from applications [2, 3, 27, 29, 37, 64, 65, 68, 79, 81, 95–97, 109, 113, 116–118, 125, 130, 137, 138, 143, 151, 152, 160, 226, 233, 239, 240]. For example, if we study the latency and throughput of Double Data Rate (DDR) DRAM over the last 15–20 years, we see that neither have been able to keep up with the growth in application working set size or CPU computational power [26, 27, 114, 117, 125].

A major bottleneck to improving the overall system performance is the high cost of *data movement*. Currently, in order to perform an operation on data that is stored within memory, the CPU must issue a request to the memory controller, which in turn sends a series of commands across an off-chip bus to the DRAM module. The data is then read from the DRAM module, at which point it is returned to the memory controller and typically stored within the CPU cache. Only after the data is placed in the CPU cache can the CPU operate (i.e., perform computation) on the data. The long latency to retrieve data from DRAM is exacerbated by two factors. First, it is difficult to send a large number of requests to memory in parallel, in part because of the narrow bandwidth of the off-chip bus between the memory controller and main memory. Second, despite the time spent on bringing the data into the cache, which is substantial [62, 63], much of the data brought into the caches is *not* reused by the CPU [179, 180], rendering the caching either very inefficient or sometimes even unnecessary. Ultimately, there is significant time and energy wasted on moving data between the CPU and memory, many times with little benefit in return, especially in workloads where caching is not very effective [2, 3].

Recent advances in memory design have unlocked the potential to avoid the unnecessary data movement. In an attempt to improve the scalability of capacity and bandwidth, memory manufacturers have turned to 3D-stacked memories, where multiple layers of memory arrays are stacked on top of each other [115, 132]. These layers are connected together using *through-silicon vias* (TSVs), which provide much greater internal memory bandwidth than the narrow off-chip bus to the CPU. Some prominent examples of these 3D-stacked memory architectures [71, 72, 74, 75, 98, 115] include a *logic layer*, which provides an opportunity to embed general-purpose computational logic *directly within main memory* to take advantage of the high internal bandwidth available.

The idea of performing *processing-in-memory* (PIM), or *near-data processing* (NDP), has been proposed for at least several decades [39, 44, 45, 56, 82, 100, 140, 166, 171, 208, 218]. However, these past efforts were *not* adopted at large scale due to the difficulty of integrating processing elements with DRAM. As a result of the potential enabled by the inclusion of a logic layer in modern memory architectures, various recent works explore a range of PIM architectures for multiple different purposes (e.g., [1–4, 9, 12, 20, 22, 28, 30, 47, 51, 52, 58, 59, 62, 63, 66–68, 80, 89, 90, 92, 93, 120, 122, 124, 131, 133, 149, 163, 172, 176, 195, 196, 199–201, 205, 221, 243, 246]).

While PIM avoids the need to move data from memory to the CPU for a number of data-intensive functions, it introduces new challenges for system architects and programmers. In particular, PIM processing logic does *not* have quick access to important mechanisms that exist within the CPU, such as address translation and cache coherence, which greatly aid the programmer. Preserving the functionality and efficiency of such mechanisms is essential for PIM, as these mechanisms can (1) preserve the traditional programming models that application developers rely on to productively write programs, and (2) provide significant performance benefits.. As we show in this work (see Section 4), simply forcing PIM processing logic to send queries to the CPU to accomplish address translation and cache coherence is very inefficient, since the overhead of a query can almost completely *eliminate* the benefits of moving computation to memory. Therefore, it is essential that we provide *PIM-specific* mechanisms that *efficiently* support the functionality of traditional address translation and cache coherence mechanisms and thus provide support for the use of existing programming models to program PIM architectures. Our goal is to design general-purpose address translation and cache coherence mechanisms that can be exploited by any PIM processing logic to provide low-overhead support for common functions, such as pointer chasing in virtual memory and cache coherence.

To this end, we propose two mechanisms to support PIM. The first mechanism, IMPICA, is an in-memory accelerator for pointer chasing, which exploits the high bandwidth available within 3D-stacked memory. IMPICA can 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) or without using the page walkers within the CPU. The second mechanism, LazyPIM, maintains cache coherence between PIM processing logic and CPU cores *without* sending coherence requests for every memory access. Instead, LazyPIM efficiently provides coherence by having PIM processing logic speculatively acquire coherence permissions, and then later sends compressed *batched* coherence lookups to the CPU to determine whether or not its speculative permission acquisition violated the memory ordering defined by the programming model.

In Section 1, we cover common design principles of modern PIM architectures. In Section 2, we discuss key issues that impact the flexibility and adoption of PIM architectures. In Section 3, we discuss IMPICA, an accelerator that we propose to efficiently support pointer-chasing operations within PIM architectures. In Section 4, we discuss LazyPIM, a mechanism that we propose to efficiently support cache coherence between PIM processing logic and the CPU cores. In Section 5, we discuss related work in the area, and in Section 6 we briefly discuss some future research challenges, with a focus on system-level challenges for the adoption of PIM architectures.

1 DESIGNING PROCESSING-IN-MEMORY ARCHITECTURES

Processing-in-memory (PIM) architectures place some form of processing logic (typically accelerators, simple cores, or reconfigurable logic) inside the DRAM subsystem. This *PIM processing logic*, which we also refer to as *PIM cores* or *PIM engines*, interchangeably, can execute portions of applications or entire application kernels, depending on the design of the architecture. In this section, we first discuss how the PIM processing logic is integrated within DRAM modules (Section 1.1), and then we discuss how applications make use of this PIM processing logic (Section 1.2).

1.1 Placing Processing Logic Within the DRAM Subsystem

Modern PIM architectures rely on implementing processing logic in the DRAM chip itself (e.g., [2–4, 12, 20, 22, 28, 30, 47, 51, 52, 58, 59, 66–68, 89, 90, 92, 93, 118, 120, 122, 131, 133, 149, 172, 176, 195, 196, 199–203, 221, 243, 246]) or on the DRAM module or the DRAM controller [9, 62, 63, 201]. DRAM consists of multiple arrays of capacitive *cells*, where each cell holds one bit of data. By placing processing logic in close proximity of the cell arrays, PIM architectures are *not* restricted to the limited bandwidth offered by the narrow off-chip bus between the DRAM module and the CPU. Instead, PIM processing logic benefits from the much wider buses that are available within the chip and/or module in modern DRAM architectures.

Figure 1 shows an overview of a 3D-stacked DRAM based architecture. Examples of 3D-stacked DRAM include High-Bandwidth Memory (HBM) [75, 115] and the Hybrid Memory Cube (HMC) [2, 71, 72]. As the figure shows, a 3D-stacked DRAM consists of multiple layers. 3D-stacked DRAM has a much greater internal data bandwidth than conventional memory, due to its use of *through-silicon vias* (TSVs), which are vertical links that connect the multiple layers of a DRAM stack together [115, 132]. In addition to containing multiple layers of DRAM, a number of 3D-stacked DRAM architectures, such as HBM [75, 115] and HMC [71, 72], include a *logic layer*, typically the bottommost layer, where architects can implement functionality that interacts with both the processor and the DRAM cells [2, 3, 71, 72]. Currently, 3D-stacked DRAM makes limited use of the logic layer (e.g., HMC implements command scheduling logic within the logic layer [71, 72]).

Recent PIM proposals (e.g., [2–4, 12, 20, 22, 30, 47, 51, 52, 58, 59, 66–68, 89, 90, 92, 93, 120, 131, 133, 149, 163, 172, 176, 196, 221, 243, 246]) add processing logic to the logic layer to exploit the high bandwidth available between the logic layer and the DRAM cell arrays. The proposed PIM processing logic design varies based on the specific architecture, and can range from fixed-function accelerators to simple *in-order* cores, and to reconfigurable logic. The complexity of the processing logic that can be added to the logic layer is currently limited by the

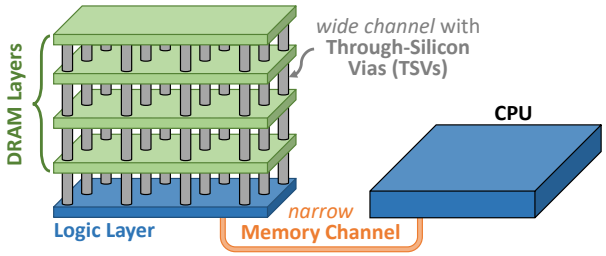


Fig. 1. High-level overview of a 3D-stacked DRAM based architecture.

manufacturing process technology and thermal design points, which may prevent highly-sophisticated processors (e.g., out-of-order processor cores with large caches and sophisticated instruction-level parallelism techniques) from being implemented within the logic layer at this time [43, 172, 243].

1.2 Using PIM Processing Logic Functionality in Applications

In order for applications to make use of PIM processing logic that resides within DRAM, each PIM architecture exposes an interface to the CPU. While there currently is no standardization of this interface, most contemporary works on PIM architectures follow similar models for CPU–PIM interactions. PIM processing logic is typically treated as a coprocessor, and executes only when some code (which we refer to as a *PIM kernel*) is launched by the CPU on the PIM processing logic. PIM kernels vary widely in current proposals in terms of granularity. Some works (e.g., [2, 52]) treat an *entire application* thread as a PIM kernel, in order to minimize the amount of synchronization and data sharing that takes place between the main CPU and main compute-capable memory. Many works (e.g., [4, 9, 20, 47, 51, 59, 68, 90, 92, 93, 122, 131, 172, 195, 196, 199, 200, 243]) treat only portions of an application thread (e.g., *functions*) as a PIM kernel, and launch the kernel when a CPU core reaches the PIM kernel call. Yet other works (e.g., [3, 120, 163]) use a much finer granularity for offloading code to PIM: they offload only a *single instruction* as the PIM kernel, which is completed atomically.

An open question for all of these architectures is how a PIM kernel is identified and demarcated, and who is responsible for identification and demarcation. Current works on PIM expect the compiler or programmer to mark sections of the code and/or data that are to be dispatched to the PIM processing logic. When a program reaches a point at which a PIM kernel should be executed, the CPU uses the off-chip memory channel to dispatch the kernel to a free PIM core. The PIM core then executes the kernel, and upon completing the kernel, notifies the CPU using the memory channel. Several works [2, 3, 21, 68] provide a detailed explanation of this process.

Due to the simple nature of PIM processing logic (i.e., that PIM processing logic is expected to be fixed-function accelerators, small in-order general-purpose cores, or simple reconfigurable logic), current PIM architectures do *not replace* the CPU cores with PIM cores; they instead *augment* the existing CPU cores. OS threads continue to run on the CPU cores, and key structures to support application execution, such as large caches, translation lookaside buffers (TLBs), page walkers, and cache coherence hardware, are expected to remain within the CPU. While these decisions minimize the changes that need to be made to write programs for PIM architectures, the decisions introduce new issues that PIM architectures must address to maintain ease of adoption. We discuss two such critical issues in the next section.

2 KEY ISSUES IN ENABLING PROCESSING-IN-MEMORY

Pushing some or all of the computation for a program from the CPU to the DRAM introduces new challenges for system architects (as well as programmers) to overcome. In particular, PIM processing logic does *not* have direct

access to structures within the CPU that are essential to memory operations, such as address translation and cache coherence hardware. A naive solution is to simply have PIM processing logic access these structures remotely over the off-chip memory channel. Unfortunately, such likely-frequent remote accesses can introduce a high performance and energy overhead, and often undermine many, if not all, of the benefits that PIM architectures provide. A second naive solution is to limit the functionality of the PIM processing logic such that it *cannot* perform address translation or cache coherence, and to expose these limitations to programmers. However, this alters the programming model of the system, and can lead to great difficulty for the widespread adoption of PIM as an execution model. In this section, we focus on the address translation and cache coherence challenges, and discuss why naive solutions are not practical. We discuss new PIM-specific solutions that can overcome these challenges in Sections 3 and 4.

2.1 Address Translation

A large amount of code relies on pointers, which are stored as *virtual* memory addresses. When the application follows a pointer, a core must perform *address translation*, which converts the pointer's stored virtual address into a *physical* address within main memory. If PIM processing logic relies on existing CPU-side address translation mechanisms, any performance gains from performing pointer chasing in memory could easily be nullified, as the processing logic needs to send a long-latency translation request to the CPU via the off-chip channel for each memory access. The translation can sometimes require a page table walk, where the CPU must issue *multiple* memory requests to read the page table, which further increases traffic on the memory channel.

A naive solution is to simply duplicate the TLB and page walker within memory (i.e., within the PIM processing logic). Unfortunately, this is prohibitively difficult or expensive for three reasons: (1) coherence would have to be maintained between the CPU and memory-side TLBs, introducing extra complexity and off-chip requests; (2) the duplicated hardware is very costly in terms of storage overhead and complexity; and (3) a memory module can be used in conjunction with many different processor architectures, which use different page table implementations and formats, and ensuring compatibility between the in-memory TLB/page walker and all of these different designs is difficult.

We explore a tractable solution for PIM address translation as part of our in-memory pointer chasing accelerator, which we discuss in Section 3.

2.2 Cache Coherence

PIM processing logic can modify the data it processes, and this data may also be needed by CPU cores. In a traditional multithreaded execution model that uses shared memory between threads, writes to memory must be coordinated between multiple cores, to ensure that threads do not operate on stale data values. Due to the per-core caches used in CPUs, this requires that 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, which is known as *cache coherence*. Cache coherence involves a protocol that is designed to handle write permissions for each core, invalidations and updates, and arbitration when multiple cores request exclusive access to the same memory address. Within a chip multiprocessor (CMP), the per-core caches can perform coherence actions over a shared interconnect. Both snoopy [57, 167] and directory-based [23] coherence mechanisms are employed in existing multiprocessor systems.

Cache coherence is a major system challenge for enabling PIM architectures as general-purpose execution engines. 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, it is impractical for PIM to perform traditional fine-grained cache coherence, as this forces a large number of coherence

messages to traverse a narrow off-chip interconnect, potentially undoing the benefits of high-bandwidth and low-latency PIM execution, as we show in Section 4. Prior works have proposed intermediate solutions that *sidestep* coherence by either requiring the programmer to ensure data coherence or making PIM data non-cacheable in the CPU (e.g., [2–4, 28, 47, 51, 52, 59, 67, 68, 149, 163, 172, 195, 196, 199, 200, 243]). Unfortunately, these solutions either place some restrictions on the programming model or limit the performance and energy gains achievable by a PIM architecture.

In this chapter, we describe a new coherence protocol, which allows PIM processing logic to efficiently perform coherence *without* incurring high overhead or changing the programming model, which we discuss in Section 4.

3 IMPICA: AN IN-MEMORY POINTER-CHASING ACCELERATOR

Linked data structures, such as trees, hash tables, and linked lists are commonly used in many important applications [5, 35, 46, 50, 54, 63, 142, 153, 155, 227, 236]. For example, many databases use B/B$^+$-trees to efficiently index large data sets [46, 54], key-value stores use linked lists to handle collisions in hash tables [50, 142], and graph processing workloads [2, 3, 209] use pointers to represent graph edges. These structures link nodes using pointers, where each node points to at least one other node by storing its address. Traversing the link requires serially accessing consecutive nodes by retrieving the address(es) of the next node(s) from the pointer(s) stored in the current node. This fundamental operation is called *pointer chasing* in linked data structures.

Pointer chasing is currently performed by the CPU cores, as part of an application thread. While this approach eases the integration of pointer chasing into larger programs, pointer chasing can be inefficient within the CPU, as it introduces several sources of performance degradation: (1) dependencies exist between memory requests to the linked nodes, resulting in serialized memory accesses and limiting the available instruction-level and memory-level parallelism [63, 153, 154, 159, 186]; (2) irregular allocation or rearrangement of the connected nodes leads to access pattern irregularity [35, 40, 78, 83, 153, 155, 238], causing frequent cache and TLB misses; and (3) link traversals in data structures that diverge at each node (e.g., hash tables, B-trees) frequently go down different paths during different iterations, resulting in little reuse, further limiting cache effectiveness [136]. Due to these inefficiencies, a significant *memory bottleneck* arises when executing pointer chasing operations in the CPU, which stalls on a large number of memory requests that suffer from the long round-trip latency between the CPU and the memory.

Many prior works (e.g., [33–35, 40, 69, 70, 78, 83, 128, 135, 136, 153, 155, 186, 187, 213, 232, 238, 242, 248]) proposed mechanisms to predict and prefetch the next node(s) of a linked data structure early enough to hide the memory latency. Unfortunately, prefetchers for linked data structures suffer from several shortcomings: (1) they usually do *not* provide significant benefit for data structures that diverge at each node [83, 153, 155], due to low prefetcher accuracy and low miss coverage; (2) aggressive prefetchers can consume too much of the limited off-chip memory bandwidth and, as a result, slow down the system [40–42, 62, 78, 110, 111, 204, 216]; and (3) a prefetcher that works well for some pointer-based data structure(s) and access patterns (e.g., a Markov prefetcher designed for mostly-static linked lists [78]) usually does not work efficiently for different data structures and/or access patterns. Thus, it is important to explore new solution directions to alleviate the significant performance and efficiency loss due to pointer chasing.

Our goal in this section is to accelerate pointer chasing by *directly minimizing the memory bottleneck* caused by pointer chasing operations. To this end, we propose to perform pointer chasing *inside main memory* by leveraging processing-in-memory (PIM) mechanisms, *avoiding the need to move data to the CPU*. In-memory pointer chasing greatly reduces (1) the latency of the operation, as an address does not need to be brought all the way into the CPU before it can be dereferenced; and (2) the reliance on caching and prefetching in the CPU, which are largely ineffective for pointer chasing over large data structures. In this section, we *describe an in-memory accelerator for chasing pointers* in any linked data structure, called the *In-Memory PoInter Chasing Accelerator* (IMPICA) [67].

IMPICA leverages the low memory access latency at the logic layer of 3D-stacked memory to speed up pointer chasing operations.

We identify *two fundamental challenges that we believe exist for a wide range of in-memory accelerators*, and evaluate them as part of a case study in designing a pointer chasing accelerator in memory. These fundamental challenges are (1) how to achieve high parallelism in the accelerator (in the presence of serial accesses in pointer chasing), and (2) how to effectively perform virtual-to-physical address translation on the memory side without performing costly accesses to the CPU's memory management unit. We call these, respectively, the *parallelism challenge* and the *address translation challenge*.

The Parallelism Challenge. Parallelism is challenging to exploit in an in-memory accelerator even with the reduced latency and higher bandwidth available within 3D-stacked memory, as the performance of pointer chasing is limited by *dependent sequential accesses*. The serialization problem can be exacerbated when the accelerator traverses multiple streams of links: while traditional out-of-order or multicore CPUs can service memory requests from multiple streams in parallel due to their ability to exploit high levels of instruction- and memory-level parallelism [55, 63, 154, 156–159, 222], simple accelerators are unable to exploit such parallelism unless they are carefully designed (e.g., [2, 47, 176, 246]).

We observe that accelerator-based pointer chasing is primarily bottlenecked by memory access latency, and that the address generation computation for link traversal takes only a small fraction of the total traversal time, leaving the accelerator idle for a majority of the traversal time. In IMPICA, we exploit this idle time by *decoupling* link address generation from the issuing and servicing of a memory request, which allows the accelerator to generate addresses for one link traversal stream while waiting on the request associated with a different link traversal stream to return from memory. We call this design *address-access decoupling*. Note that this form of decoupling bears resemblance to decoupled access/execute architectures [210–212], and we in fact take inspiration from past works [36, 102, 210–212], except our design is *specialized* for building a pointer chasing accelerator in 3D-stacked memory, and this paper solves specific challenges within the context of pointer chasing acceleration.

The Address Translation Challenge. An in-memory pointer chasing accelerator must be able to perform address translation, as each pointer in a linked data structure node stores the *virtual* address of the next node, even though main memory is *physically* addressed. To determine the next address in the pointer chasing sequence, the accelerator must resolve the virtual-to-physical address mapping. As we discuss in Section 2.1, relying on existing CPU-side address translation mechanisms or duplicating the TLB and page walker within DRAM are impractical solutions.

We observe that traditional address translation techniques do *not* need to be employed for pointer chasing, as link traversals are (1) limited to linked data structures, and (2) touch only certain data structures in memory. We exploit this in IMPICA by allocating data structures accessed by IMPICA into contiguous *regions* within the virtual memory space, and designing a new address translation mechanism, the *region-based page table*, which is optimized for in-memory acceleration. Our approach provides translation within memory at low latency and low cost, while minimizing the cost of maintaining TLB coherence.

Evaluation. By solving both key challenges, IMPICA provides significant performance and energy benefits for pointer chasing operations and applications that use such operations. First we examine three microbenchmarks, each of which performs pointer chasing on a widely-used data structure (linked list, hash table, B-tree), and find that IMPICA improves their performance by 92%, 29%, and 18%, respectively, on a quad-core system over a state-of-the-art baseline. Second, we evaluate IMPICA on a real database workload, DBx1000 [241], on a quad-core system, and show that IMPICA increases *overall* database transaction throughput by 16% and reduces transaction latency by 13%. Third, IMPICA reduces *overall* system energy, by by 41%, 23%, and 10% for the three microbenchmarks and by 6% for DBx1000. These benefits come at a very small hardware cost: our evaluations show that IMPICA comprises only 7.6% of the area of a small embedded core (the ARM Cortex-A57 [7]).

Our IMPICA proposal, originally published in the ICCD 2016 conference [67], makes the following major contributions:

- This is the first work to propose an in-memory accelerator for pointer chasing. Our proposal, IMPICA, accelerates linked data structure traversal by chasing pointers inside the logic layer of 3D-stacked memory, thereby eliminating inefficient, high-latency serialized data transfers between the CPU and main memory.
- We identify two fundamental challenges in designing an efficient in-memory pointer chasing accelerator (Section 3.2). These challenges can greatly hamper performance if the accelerator is not designed *carefully* to overcome them. First, multiple streams of link traversal can unnecessarily get serialized at the accelerator, thereby degrading performance (the *parallelism challenge*). Second, an in-memory accelerator needs to perform virtual-to-physical address translation for each pointer, but this critical functionality does *not* exist on the memory side (the *address translation challenge*).
- IMPICA solves the *parallelism challenge* by decoupling link address generation from memory accesses, and utilizes the idle time during memory accesses to service *multiple* pointer chasing streams simultaneously. We call this approach *address-access decoupling* (Section 3.3.1).
- IMPICA solves the *address translation challenge* by allocating data structures it accesses into contiguous virtual memory regions, and using an optimized and low-cost *region-based page table* structure for address translation (Section 3.3.3).
- We evaluate IMPICA extensively using both microbenchmarks and a real database workload. Our results (Section 3.6) show that IMPICA improves both system performance and energy efficiency for all of these workloads, while requiring only very modest hardware overhead in the logic layer of 3D-stacked DRAM.

3.1 Motivation

To motivate the need for a pointer chasing accelerator, we first examine the usage of pointer chasing in contemporary workloads. We then discuss opportunities for acceleration within 3D-stacked memory.

3.1.1 Pointer Chasing in Modern Workloads. Pointers are ubiquitous in fundamental data structures such as linked lists, trees, and hash tables, where the nodes of the data structure are linked together by storing the addresses (i.e., pointers) of neighboring nodes. Pointers make it easy to dynamically add/delete nodes in these data structures, but link traversal is often serialized, as the address of the next node can be known only after the current node is fetched. The serialized link traversal is commonly referred to as *pointer chasing*.

Due to the flexibility of insertion/deletion, pointer-based data structures and link traversal algorithms are essential building blocks in programming, and they enable a very wide range of workloads. For instance, at least seven different types of modern data-intensive applications rely *heavily* on linked data structures: (1) **databases and file systems** use B/B$^+$-trees for indexing tables or metadata [46, 53, 54, 183]; (2) **in-memory caching** applications based on key-value stores, such as Memcached [50] and Masstree [142], use linked lists to resolve hash table collisions and trie-like B$^+$-trees as their main data structures; (3) **graph processing workloads** use pointers to represent the edges that connect the vertex data structures together [2, 209]; (4) **garbage collectors** in high level languages typically maintain reference relations using trees [76, 77, 227]; (5) **3D video games** use binary space partitioning trees to determine the objects that need to be rendered [76, 164]; (6) **dynamic routing tables** used by networks employ balanced search trees for high-performance IP address lookups [224]; and (7) **hash table based DNA read mappers** that store and find potential locations of a read in a reference genome index [5, 6, 92, 93, 112, 235, 236].

While linked data structures are widely used in many modern applications, chasing pointers is very inefficient in general-purpose processors. There are three major reasons behind the inefficiency. First, the inherent serialization that occurs when accessing consecutive nodes limits the available instruction-level and memory-level

parallelism [78, 128, 136, 153–159, 186, 187]. As a result, out-of-order execution provides only limited performance benefit when chasing pointers [153–156]. Second, as nodes can be inserted and removed dynamically, they can get allocated to different regions of memory. The irregular memory allocation causes pointer chasing to exhibit irregular access patterns, which lead to frequent cache and TLB misses [40, 78, 83, 153, 238]. Third, for data structures that diverge at each node, such as B-trees, link traversals often go down different paths during different iterations, as the inputs to the traversal function change. As a result, lower-level nodes that were recently referenced during a link traversal are unlikely to be reused in subsequent traversals, limiting the effectiveness of many caching policies [99, 128, 136], such as LRU replacement.

To quantify the performance impact of chasing pointers in real-world workloads, we profile two popular applications that heavily depend on linked data structures, using a state-of-art Intel Xeon system:[1] (1) *Memcached* [50], an in-memory caching system, using a real Twitter dataset [48] as its input; and (2) *DBx1000* [241], an in-memory database system, using the TPC-C benchmark [223] as its input. We profile the pointer chasing code within the application separately from other parts of the application code. Figure 2 shows how pointer chasing compares to the rest of the application in terms of execution time, cycles per instruction (CPI), and the ratio of last-level cache (LLC) miss cycles to the total cycles.

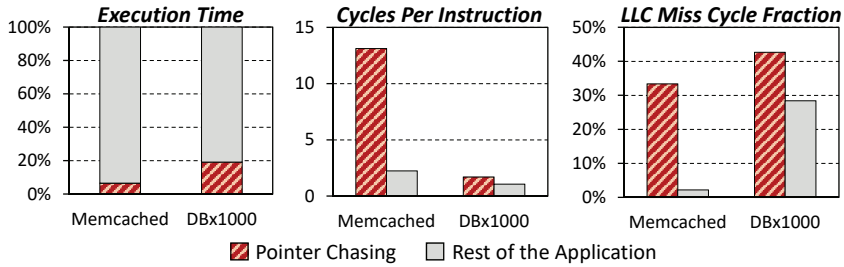


Fig. 2. Profiling results of pointer chasing portions of code vs. the rest of the application code in Memcached and DBx1000. Figure adapted from [67].

We make three major observations from Figure 2. First, both Memcached and DBx1000 spend a significant fraction of their total execution time (7% and 19%, respectively) on pointer chasing, as a result of dependent cache misses [63, 153, 155, 186]. Though these percentages might sound small, real software often does *not* have a single type of operation that consumes this significant a fraction of the total time. Second, we find that pointer chasing is significantly more inefficient than the rest of the application, as it requires much higher cycles per instruction (6× in Memcached, and 1.6× in DBx1000). Third, pointer chasing is largely memory-bound, as it exhibits much higher cache miss rates than the rest of the application and as a result spends a much larger fraction of cycles waiting for LLC misses (16× in Memcached, and 1.5× in DBx1000). From these observations, we conclude that (1) pointer chasing consumes a significant fraction of execution time in two important sophisticated applications, (2) pointer chasing operations are bound by memory, and (3) executing pointer chasing code in a modern general-purpose processor is very inefficient and thus can lead to a large performance overhead. Other works made similar observations for different workloads [35, 63, 153, 155, 186].

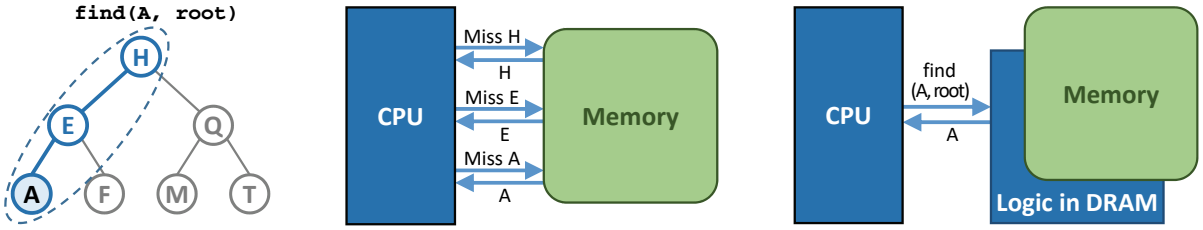
Prior works (e.g., [33–35, 40, 69, 70, 78, 83, 128, 135, 136, 153, 155, 186, 187, 213, 232, 238, 242, 248]) proposed specialized prefetchers that predict and prefetch the next node of a linked data structure to hide memory latency. While prefetching can mitigate part of the memory latency problem, it has three major shortcomings. First, the efficiency of a prefetcher degrades significantly when the traversal of linked data structures diverges

[1]We use the Intel® VTune™ profiling tool on a machine with a Xeon® W3550 processor (3GHz, 8-core, 8 MB LLC) [73] and 18 GB memory. We profile each application for 10 minutes after it reaches steady state.

into multiple paths and the access order is irregular [83, 153, 155]. Second, prefetchers can sometimes slow down the entire system due to contention caused by inaccurate as well as accurate prefetch requests [40–42, 62, 78, 110, 111, 204, 216]. Third, these specialized hardware prefetchers are usually designed for specific data structure implementations, and tend to be very inefficient when dealing with other data structures. For example, a Markov prefetcher [78] can potentially be very effective for static linked lists, but it becomes very inefficient for trees with dynamic access patterns. It is difficult to design a prefetcher that is efficient and effective for *all* types of linked data structures. **Our goal** in this work is to improve the performance of pointer chasing applications *without* relying on prefetchers, regardless of the types and access patterns of linked data structures used in an application.

3.1.2 Accelerating Pointer Chasing in 3D-Stacked Memory. We propose to improve the performance of pointer chasing by leveraging processing-in-memory (PIM) to alleviate the memory bottleneck. Instead of sequentially fetching *each node* from memory and sending it to the CPU when an application is looking for a particular node, PIM-based pointer chasing consists of (1) traversing the linked data structures *in memory*, and (2) returning only the final node found to the CPU.

Unlike prior works that proposed general architectural models for in-memory computation by embedding logic in main memory [1, 3, 9, 22, 28, 47, 51, 52, 59, 62, 63, 66, 68, 80, 124, 131, 149, 163, 172, 195, 196, 199–201, 205, 221, 243], we propose to design a *specialized In-Memory PoInter Chasing Accelerator* (IMPICA) that exploits the logic layer of 3D-stacked memory [71, 72, 74, 75, 115]. 3D die-stacked memory achieves low latency (and high bandwidth) by stacking memory dies on top of a logic die, and interconnecting the layers using through-silicon vias (TSVs). Figure 3 shows a binary tree traversal using IMPICA, compared to a traditional architecture where the CPU traverses the binary tree. The traversal sequentially accesses the nodes from the root to a particular node (e.g., **H→E→A** in Figure 3a). In a traditional architecture (Figure 3b), these serialized accesses to the nodes miss in the caches and three memory requests are sent to memory serially across a high-latency off-chip channel. In contrast, IMPICA traverses the tree inside the logic layer of 3D-stacked memory, and as Figure 3c shows, only the final node (**A**) is sent from the memory to the host CPU in response to the traversal request. Doing the traversal in memory minimizes both traversal latency (as queuing delays in the on-chip interconnect and the CPU-to-memory bus are eliminated) and off-chip bandwidth consumption, as shown in Figure 3c.



(a) Binary tree (b) Traditional architecture (c) IMPICA architecture

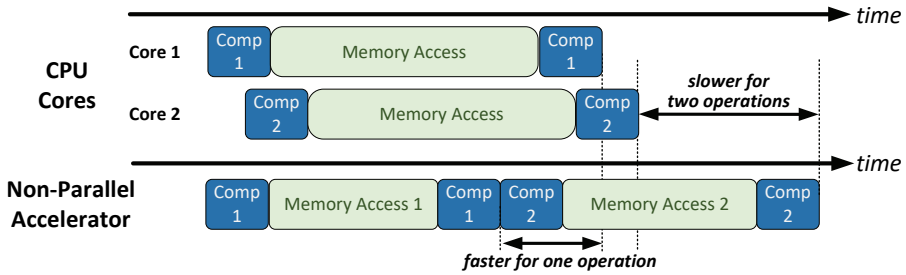
Fig. 3. Pointer chasing (a) in a traditional architecture (b) and in IMPICA with 3D-stacked memory (c). Figure adapted from [67].

Our accelerator architecture has three major advantages. First, it improves performance and reduces memory bandwidth consumption by eliminating the round trips required for memory accesses over the CPU-to-memory interconnects. Second, it frees the CPU to execute other work than linked data structure traversal, thereby increasing system throughput. Third, it minimizes the cache contention caused by pointer chasing operations.

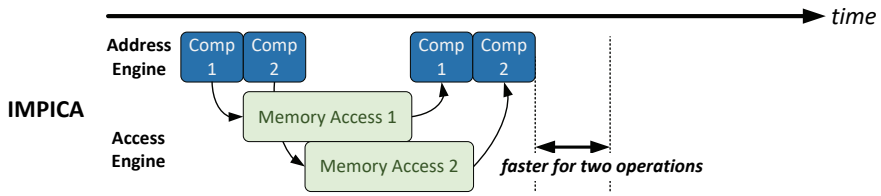
3.2 Design Challenges

We identify and describe two new challenges that are crucial to the performance and functionality of our new pointer chasing accelerator in memory: (1) the *parallelism challenge*, and (2) the *address translation challenge*. Section 3.3 describes our IMPICA architecture, which centers around two key ideas that solve these two challenges.

3.2.1 Challenge 1: Parallelism in the Accelerator. A pointer chasing accelerator supporting a multicore system needs to handle *multiple* link traversals (from different cores) in parallel at low cost. A simple accelerator that can handle only one request at a time (which we call a *non-parallel accelerator*) would serialize the requests and could potentially be slower than using multiple CPU cores to perform the multiple traversals. As depicted in Figure 4a, while a non-parallel accelerator speeds up each *individual* pointer chasing operation done by one of the CPU cores due to its shorter memory latency, the accelerator is slower *overall* for two pointer chasing operations, as *multiple cores* can operate in *parallel* on independent pointer chasing operations.



(a) Pointer chasing on two CPU cores vs. one non-parallel accelerator



(b) Pointer chasing using our IMPICA proposal

Fig. 4. Execution time of two independent pointer chasing operations, broken down into address computation time (*Comp*) and memory access time. Figure adapted from [67].

To overcome this deficiency, an in-memory accelerator needs to exploit parallelism when it services requests. However, the accelerator must do this *at low cost* and *low complexity*, due to its placement within the logic layer of 3D-stacked memory, where complex logic, such as out of order execution circuitry, is currently not feasible. The straightforward solution of adding multiple accelerators to service independent pointer chasing operations (e.g., [99]) does not scale well, and also can lead to excessive energy dissipation (and, thus, potentially thermal violations) and die area usage in the logic layer.

A key observation we make is that pointer chasing operations are bottlenecked by memory stalls, as shown in Figure 2. In our evaluation, the memory access time is 10–15× the computation time (see Section 3.5 for our methodology). As a result, the accelerator spends a significant amount of time waiting for memory, causing its compute resources to sit idle. This makes typical in-order or out-of-order execution engines *inefficient* for an

in-memory pointer-chasing accelerator. If we utilize the hardware resources in a more efficient manner, we can enable parallelism by handling *multiple* pointer chasing operations *within a single accelerator*.

Based on our observation, we *decouple* address generation from memory accesses in IMPICA using two engines (the address engine and the access engine), allowing the accelerator to generate addresses from one pointer chasing operation while it *concurrently* performs memory accesses for a different pointer chasing operation (as shown in Figure 4b). We describe the details of our decoupled accelerator design in Section 3.3.

3.2.2 Challenge 2: Virtual Address Translation. A second challenge arises when pointer chasing is moved out of the CPU cores, which are equipped with facilities for address translation. Within the program data structures, each pointer is stored as a virtual address, and requires *translation* to a physical address before its memory access can be performed. This is a challenging task for an in-memory accelerator, which has no easy access to the virtual address translation engine that sits in the CPU core. While sequential array operations could potentially be constrained to work within page boundaries or directly in physical memory, indirect memory accesses that come with pointer-based data structures require some support for virtual memory address translation, as they might touch many parts of the virtual address space.

There are two major issues when designing a virtual address translation mechanism for an in-memory accelerator. First, different processor architectures have different page table implementations and formats. This lack of compatibility makes it very expensive to simply replicate the CPU page table walker in the in-memory accelerator as this approach requires replicating TLBs and page walkers for many architecture formats. Second, a page table walk tends to be a high-latency operation involving multiple memory accesses due to the heavily layered format of a conventional page table. As a result, TLB misses are a major performance bottleneck in data-intensive applications [10, 11, 13, 15–17, 139, 173, 175, 215]. If the accelerator requires many page table walks that are supported by the CPU's address translation mechanisms, which require high-latency off-chip accesses for the accelerator, its performance can degrade greatly.

To address these issues, we *completely decouple* the page table of IMPICA from that of the CPU, thereby obviating the need for compatibility between the two page tables. This presents us with an opportunity to develop a new page table design that is much more efficient for our in-memory accelerator. We make two key observations about the behavior of a pointer chasing accelerator. First, the accelerator operates only on certain data structures that can be mapped to *contiguous regions* in the virtual address space, which we refer to as *IMPICA regions*. As a result, it is possible to map contiguous IMPICA regions with a *smaller, region-based* page table without needing to duplicate the page table mappings for the *entire* address space. Second, we observe that if we need to map *only* IMPICA regions, we can collapse the hierarchy present in conventional page tables, which allows us to limit the hardware and storage overhead of the IMPICA page table. We describe the IMPICA page table in detail in Section 3.3.3.

3.3 IMPICA Architecture

We propose a new in-memory accelerator, IMPICA, that addresses the two design challenges that face in-memory accelerators for pointer chasing. The IMPICA architecture consists of a single specialized core designed to decouple address generation from memory accesses. Our approach, which we call *address-access decoupling*, allows us to *efficiently* overcome the parallelism challenge (Section 3.3.1). The IMPICA core uses a novel *region-based page table* design to perform efficient address translation locally in the accelerator, which allows us to overcome the address translation challenge (Section 3.3.3).

3.3.1 IMPICA Core Architecture. Our IMPICA core uses what we call address-access decoupling, where we separate the core into two parts: (1) an *address engine*, which generates the address specified by the pointer; and (2) an *access engine*, which performs memory access operations using addresses generated by the address engine. The key advantage of this design is that the address engine supports fast context switching between

multiple pointer chasing operations, allowing it to utilize the idle time during memory access(es) to compute addresses from a different pointer chasing operation. As Figure 4b shows, an IMPICA core can process multiple pointer chasing operations faster than multiple cores because it has the ability to overlap address generation with memory accesses.

Our address-access decoupling has similarities to, and is in fact inspired by, the decoupled access-execute (DAE) architecture [210–212], with two key differences. First, the goal of DAE is to exploit instruction-level parallelism (ILP) within a *single* thread, whereas our goal is to exploit thread-level parallelism (TLP) across pointer chasing operations from *multiple* threads. Second, unlike DAE, the decoupling in IMPICA does not require any programmer or compiler effort. Our approach is much simpler than both general-purpose DAE and out-of-order execution [169, 170, 222], as it can switch between different independent execution streams, without the need for dependency checking.

Figure 5 shows the architecture of the IMPICA core. The host CPU initializes a pointer chasing operation by moving its code to main memory, and then enqueuing the request in the *request queue* (❶ in Figure 5). Section 3.4.1 describes the details of the CPU interface.

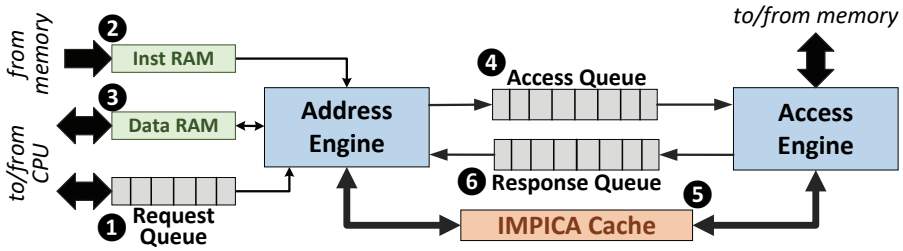


Fig. 5. IMPICA core architecture. Figure adapted from [67].

The *address engine* services the enqueued request by loading the pointer chasing code into its *instruction RAM* (❷). This engine contains all of IMPICA's functional units, and executes the code in its instruction RAM while using its *data RAM* (❸) as a stack. All instructions that do not involve memory accesses, such as ALU operations and control flow, are performed by the address engine. The number of pointer chasing operations that can be processed in parallel is limited by the size of the stack in the data RAM.

When the address engine encounters a memory instruction, it enqueues the address (along with the data RAM stack pointer) into the *access queue* (❹), and then performs a *context switch* to an independent stream. For the switch, the engine pushes the hardware context (i.e., architectural registers and the program counter) into the data RAM stack. When this is done, the address engine can work on a different pointer chasing operation.

The *access engine* services requests waiting in the access queue. This engine translates the enqueued address from a virtual address to a physical address, using the IMPICA page table (see Section 3.3.3). It then sends the physical address to the memory controller, which performs the memory access. Since the memory controller handles data retrieval, the access engine can issue multiple requests to the controller without waiting on the data, just as the CPU does today, thereby quickly servicing the queued requests. Note that the access engine does *not* contain any functional units.

When the access engine receives data back from the memory controller, it stores this data in the *IMPICA cache* (❺), a small cache that contains data destined for the address engine. The access queue entry corresponding to the returned data is moved from the access queue to the *response queue* (❻).

The address engine monitors the response queue. When a response queue entry is ready, the address engine reads it, and uses the stack pointer to access and reload the registers and PC that were pushed onto the data

RAM stack. It then resumes execution for the pointer chasing operation, continuing until it encounters the next memory instruction.

3.3.2 IMPICA Cache. IMPICA uses a cache to deliver data fetched by the access engine to the address engine. The cache employs three features that cater to pointer-chasing applications. First, it uses *cache line locking* to guarantee that data is not displaced from the cache until the address engine processes the data. Cache line locking is achieved using a *lock bit* in the tag that is set when the cache line is inserted, and is cleared only after the address engine processes the associated entry in the response queue. If all of the cache lines in a set are locked, the access engine stalls until one of the cache lines becomes unlocked. Second, when a traversal is completed, the IMPICA cache *immediately* evicts cache lines fetched by that pointer-chasing operation. A *request ID* associated with the tag is used to determine if a cache line belongs to a completed task. Third, the IMPICA cache prioritizes nodes that closer to the root of the data structure in the cache, by leveraging the observation that pointer-based structures traverse multiple paths and usually do *not* re-reference the leaf nodes. To achieve this, the cache sets a **root bit** in the tag if a cache line is fetched by the first few memory accesses of a pointer-chasing operation. Figure 6 shows the structure of the IMPICA cache, including cache line metadata.

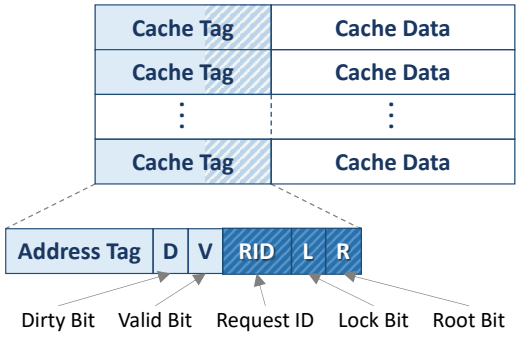


Fig. 6. Structure of the IMPICA cache.

3.3.3 IMPICA Page Table. IMPICA uses a *region-based* page table (RPT) design optimized for in-memory pointer chasing, leveraging the continuous ranges of accesses (*IMPICA regions*) discussed in Section 3.2.2. Figure 7 shows the structure of the RPT in IMPICA. The RPT is split into three levels: (1) a first-level *region table*, which needs to map only a small number of the contiguously-allocated IMPICA regions; (2) a second-level *flat page table* for each region with a larger (e.g., 2MB) page size; and (3) third-level *small page tables* that use conventional small (e.g., 4KB) pages. In the example in Figure 7, when a 48-bit virtual memory address arrives for translation, bits 47–41 of the address are used to index the region table (❶ in Figure 7) to find the corresponding flat page table. Bits 40–21 are used to index the flat page table (❷), providing the location of the small page table, which is indexed using bits 20–12 (❸). The entry in the small page table provides the physical page number of the page, and bits 11–0 specify the offset within the physical page (❹).

The RPT is optimized to take advantage of the properties of pointer chasing. The region table is almost always cached in the IMPICA cache, as the total number of IMPICA regions is small, requiring small storage (e.g., a 4-entry region table needs only 68B of cache space). We employ a flat table with large (e.g., 2MB) pages at the second level in order to reduce the number of page misses, though this requires more memory capacity than the conventional 4-level page table structure. As the number of regions touched by the accelerator is limited, this additional capacity overhead remains constrained. Our page table can optionally use traditional smaller page sizes to maximize memory management flexibility. The OS can freely choose large (2MB) pages or small (4KB)

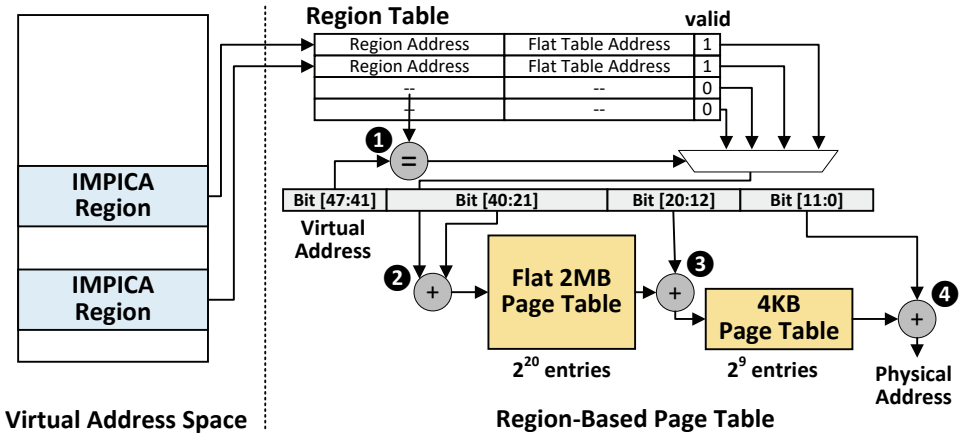


Fig. 7. IMPICA virtual memory architecture. Figure adapted from [67].

pages at the last level. Thanks to this design, a page walk in the RPT usually results in only two misses, one for the flat page table and another for the last-level small page table. This represents a 2× improvement over a conventional four-level page table, while our flattened page table still provides coverage for a 2TB memory range. The size of the IMPICA region is configurable and can be increased to cover more virtual address space. We believe that our RPT design is general enough for use in a variety of in-memory accelerators that operate on a specific range of memory regions.

We discuss how the OS manages the IMPICA RPT in Section 3.4.3.

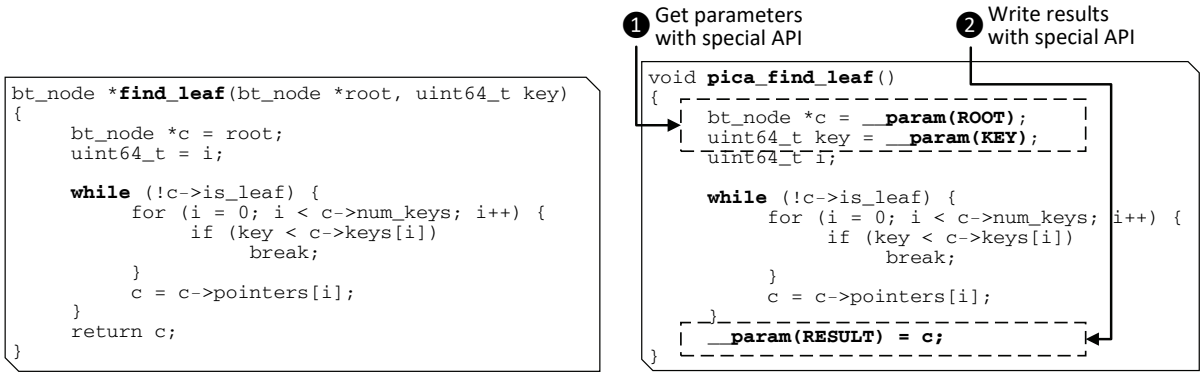
3.4 Interface and Design Considerations

In this section, we discuss how we expose IMPICA to the CPU and the operating system (OS). Section 3.4.1 describes the communication interface between the CPU and IMPICA. Section 3.4.3 discusses how the OS manages the page tables in IMPICA. In Section 3.4.4, we discuss how cache coherence is maintained between the CPU and IMPICA caches.

3.4.1 CPU Interface and Communication Model. We use a packet-based interface between the CPU and IMPICA. Instead of communicating individual operations or operands, the packet-based interface buffers requests and sends them in a burst to minimize the communication overhead. Executing a function in IMPICA consists of four steps on the interface. (1) The CPU sends to memory a packet comprising the function call and parameters. (2) This packet is written to a specific location in memory, which is memory-mapped to the *data RAM* in IMPICA and triggers IMPICA execution. (3) IMPICA loads the specific function into the *instruction RAM* with appropriate parameters, by reading the values from predefined memory locations. (4) Once IMPICA finishes the function execution, it writes the return value back to the memory-mapped locations in the *data RAM*. The CPU periodically polls these locations and receives the IMPICA output. Note that the IMPICA interface is similar to the interface proposed for the Hybrid Memory Cube (HMC) [2, 71, 72].

3.4.2 IMPICA Programming Model. The programming model for IMPICA is similar to the CPU programming model. An IMPICA program can be written in C with a new API that handles passing the parameters and returning the results to the IMPICA accelerator. Figure 8a shows the pseudocode for a B-tree traversal in the CPU, and Figure 8b shows the equivalent pseudocode for IMPICA. We observe that the code fragments are very similar, differing in only two places. First, the parameters passed in the function call of the CPU code are accessed with the

__param API call in IMPICA (❶ in Figure 8). The __param API call ensures that the program explicitly reads the parameters from the predefined memory-mapped locations of the data RAM. Second, instead of using the return statement, IMPICA uses the same __param API call to write the return value to a specific memory location (❷). This API call makes sure that the CPU can receive the output through the IMPICA interface.



(a) Conventional B-tree traversal pseudocode

(b) B-tree traversal pseudocode in IMPICA

Fig. 8. B-tree traversal pseudocode demonstrating the differences between the (a) conventional and (b) IMPICA programming models.

3.4.3 Page Table Management. In order for the RPT to identify IMPICA regions, the regions must be tagged by the application. For this, the application uses a special API to allocate pointer-based data structures. This API allocates memory to a contiguous virtual address space. To ensure that all API allocations are contiguous, the OS reserves a portion of the unused virtual address space for IMPICA, and always allocates memory for IMPICA regions from this portion. The use of such a special API requires minimal changes to applications, and it allows the system to provide more efficient virtual address translation. This also allows us to ensure that when multiple memory stacks are present within the system, the OS can allocate all IMPICA regions belonging to a single application (along with the associated IMPICA page table) into one memory stack, thereby avoiding the need for the accelerator to communicate with a remote memory stack.

The OS maintains coherence between the IMPICA RPT and the CPU page table. When memory is allocated in the IMPICA region, the OS allocates the IMPICA page table. The OS also shoots down TLB entries in IMPICA if the CPU performs any updates to IMPICA regions. While this makes the OS page fault handler more complex, the additional complexity does not cause a noticeable performance impact, as page faults occur rarely and take a long time to service in the CPU.

3.4.4 Cache Coherence. Coherence must be maintained between the CPU and IMPICA caches, and with memory, to avoid using stale data and thus ensure correct execution. We maintain coherence by executing *every* function that operates on the IMPICA regions in the accelerator. This solution guarantees that no data is shared between the CPU and IMPICA, and that IMPICA always works on up-to-date data. Other PIM coherence solutions (e.g., LazyPIM in Section 4, or those proposed by prior works [3, 51]) can also be used to allow the CPU to update the linked data structures, but we choose not to employ these solutions in our evaluation, as our workloads do *not* perform any such updates.

3.4.5 Handling Multiple Memory Stacks. Many systems need to employ multiple memory stacks to have enough memory capacity, as the current die-stacking technology can integrate only a limited number of DRAM dies into a single memory stack [145]. In systems that use multiple memory stacks, the efficiency of an in-memory accelerator such as IMPICA could be significantly degraded whenever the data that the accelerator accesses is placed on different memory stacks. Without any modifications, IMPICA would have to go through the off-chip memory channels to access the data, which would effectively eliminate the benefits of in-memory computation.

Fortunately, this challenge can be tackled with our proposed modifications to the operating system (OS) in Section 3.4.3. As we can identify the memory regions that IMPICA needs to access, the OS can easily map *all IMPICA regions* of an application into the same memory stack. In addition, the OS can allocate all IMPICA page tables into the same memory stack. This ensures that an IMPICA accelerator can access all of that data that it needs from within the memory stack that it resides in without incurring any additional hardware cost or latency overhead.

3.5 Evaluation Methodology for IMPICA

We use the gem5 [18] full-system simulator with DRAMSim2 [185] to evaluate our proposed design. We choose the 64-bit ARMv8 architecture, the accuracy of which has been validated against real hardware [60]. We conservatively model the internal memory bandwidth of the memory stack to be 4× that of the external bandwidth, similar to the configuration used in prior works [47, 243]. Our simulation parameters are summarized in Table 1. Our source code is available openly at our research group's GitHub site [188, 190]. This distribution includes the source code of our microbenchmarks as well.

Table 1. Major simulation parameters used for IMPICA evaluations.

Baseline Main Processor (CPU)	
ISA	ARMv8 (64-bits)
Core Configuration	4 OoO cores, 2 GHz, 8-wide issue, 128-entry ROB
Operating System	64-bit Linux from Linaro [127]
L1 I/D Cache	32KB/2-way each, 2-cycle
L2 Cache	1MB/8-way, shared, 20-cycle
Baseline Main Memory Parameters	
Memory Configuration	DDR3-1600, 8 banks/device, FR-FCFS scheduler [182, 249]
DRAM Bus Bandwidth	12.8 GB/s for CPU, 51.2 GB/s for IMPICA
IMPICA Accelerator	
Accelerator Core	500 MHz, 16 entries for each queue
Cache[2]	32KB / 2-way
Address Translator	32 TLB entries with region-based page table
RAM	16KB data RAM and 16KB instruction RAM

3.5.1 Workloads. We use three data-intensive microbenchmarks, which are essential building blocks in a wide range of workloads, to evaluate the native performance of pointer-chasing operations: linked lists, hash tables, and B-trees. We also evaluate the performance improvement in a real data-intensive workload, measuring the transaction latency and transaction throughput of DBx1000 [241], an in-memory OLTP database. We modify all four workloads to offload each pointer chasing request to IMPICA. To minimize communication overhead, we map the IMPICA registers to user mode address space, thereby avoiding the need for costly kernel code intervention.

Linked List. We use the linked list traversal microbenchmark [247] derived from the *health* workload in the Olden benchmark suite [184]. The parameters are configured to approximate the performance of the *health* workload. We measure the performance of the linked list traversal after 30,000 iterations.

Hash Table. We create a microbenchmark from the hash table implementation of *Memcached* [50]. The hash table in Memcached resolves hash collisions using chaining via linked lists. When there are more than $1.5n$ items in a table of n buckets, it doubles the number of buckets. We follow this rule by inserting 1.5×2^{20} random keys into a hash table with 2^{20} buckets. We run evaluations for 100,000 random key look-ups.

B-Tree. We use the B-tree implementation of DBx1000 for our B-tree microbenchmark. It is a 16-way B-tree that uses a 64-bit integer as the key of each node. We randomly generate 3,000,000 keys and insert them into the B-tree. After the insertions, we measure the performance of the B-tree traversal with 100,000 random keys. This is the most time consuming operation in the database index lookup.

DBx1000. We run DBx1000 [241] with the TPC-C benchmark [223]. We set up the TPC-C tables with 2,000 customers and 100,000 items. For each run, we spawn 4 threads and bind them to 4 different CPUs to achieve maximum throughput. We run each thread for a warm-up period for the duration of 2,000 transactions, and then record the software and hardware statistics for the next 5,000 transactions per thread,[3] which takes 300–500 million CPU cycles.

3.5.2 Die Area and Energy Estimation. We estimate the die area of the IMPICA processing logic at the 40nm process node based on recently-published work [134]. We include the most important components: processor cores, L1/L2 caches, and the memory controller. We use the area of ARM Cortex-A57 [7, 49], a small embedded processor, for the baseline main CPU. We *conservatively* estimate the die area of IMPICA using the area of the Cortex-R4 [8], an 8-stage dual issue RISC processor with 32 KB I/D caches. We believe the actual area of an optimized IMPICA design can be much smaller. Table 2 lists the area estimate of each component.

Table 2. Die area estimates using a 40nm process for IMPICA evaluations.

Baseline CPU Core (Cortex-A57)	5.85 mm^2 per core
L2 Cache	5 mm^2 per MB
Memory Controller	10 mm^2
Complete Baseline Chip	38.4 mm^2
IMPICA Core (including 32 KB I/D caches)	0.45 mm^2 (1.2% of the baseline chip area)

IMPICA comprises only 7.6% the area of a single baseline main CPU core, or only 1.2% the total area of the baseline chip (which includes four CPU cores, 1MB L2 cache, and one memory controller). Note that we conservatively model IMPICA as a RISC core. A much more specialized engine can be designed for IMPICA to solely execute pointer chasing code. Doing so would reduce the area and energy overheads of IMPICA greatly, but can reduce the generality of the pointer chasing access patterns that IMPICA can accelerate. We leave such optimizations, evaluations, and analyses for future work.

We use McPAT [123] to estimate the energy consumption of the CPU, caches, memory controllers, and IMPICA. We conservatively use the configuration of the Cortex-R4 to estimate the energy consumed by IMPICA. We use DRAMSim2 [185] to analyze DRAM energy.

[3]Based on our experiments on a real Intel Xeon machine, we find that this is large enough to satisfactorily represent the behavior of 1,000,000 transactions.

[3]We sweep the size of the IMPICA cache from 32KB to 128KB, and find that it has negligible effect on our results.

3.6 Evaluation of IMPICA

We first evaluate the effect of IMPICA on system performance, using both our microbenchmarks (Section 3.6.1) and the DBx1000 database (Section 3.6.2). We investigate the impact of different IMPICA page table designs in Section 3.6.3, and examine system energy consumption in Section 3.6.4. We compare a system containing IMPICA to an accelerator-free baseline that includes an additional 128KB of L2 cache (which is equivalent to the area of IMPICA) to ensure area-equivalence across evaluated systems.

3.6.1 Microbenchmark Performance. Figure 9 shows the speedup of IMPICA and the baseline with extra 128KB of L2 cache over the baseline for each microbenchmark. IMPICA achieves significant speedups across all three data structures — 1.92× for the linked list, 1.29× for the hash table, and 1.18× for the B-tree. In contrast, the extra 128KB of L2 cache provides very small speedup (1.03×, 1.01×, and 1.02×, respectively). We conclude that IMPICA is much more effective than the area-equivalent additional L2 cache for pointer chasing operations.

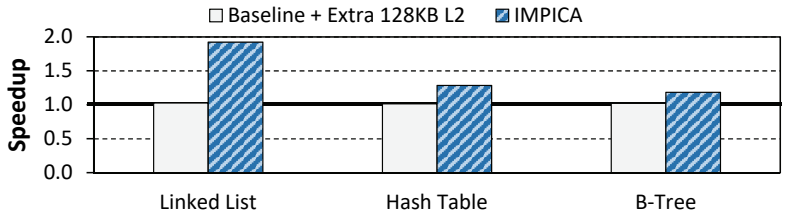


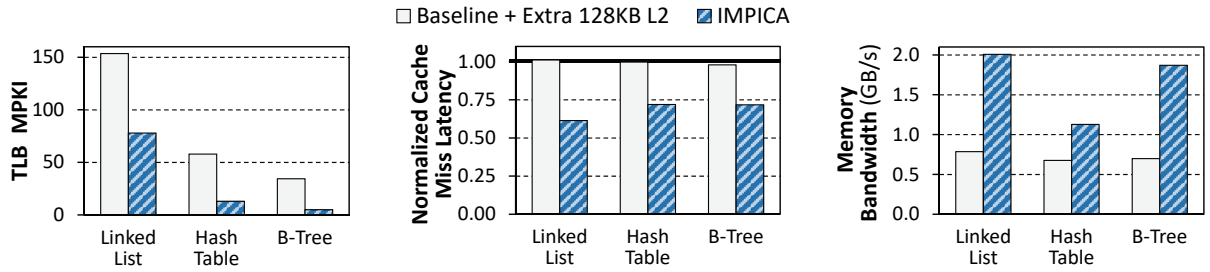
Fig. 9. Microbenchmark performance with IMPICA. Figure adapted from [67].

To provide insight into why IMPICA improves performance, we present total (i.e., combined CPU and IMPICA) TLB misses per kilo instructions (MPKI), cache miss latency, and total memory bandwidth usage for these microbenchmarks in Figure 10. We make three observations.

First, a major factor contributing to the performance improvement is the reduction in TLB misses. The TLB MPKI in Figure 10a depicts the total (i.e., combined CPU and IMPICA) TLB misses in both the baseline system and IMPICA. The pointer chasing operations have low locality and pollute the CPU TLB. This leads to a higher overall TLB miss rate in the application. With IMPICA, the pointer chasing operations are offloaded to the accelerator. This reduces the pollution and contention at the CPU TLB, reducing the overall number of TLB misses. The linked list has a significantly higher TLB MPKI than the other data structures because linked list traversal requires far fewer instructions in an iteration. It simply accesses the next pointer, while a hash table or a B-tree traversal needs to compare the keys in the node to determine the next step.

Second, we observe a significant reduction in last-level cache miss latency with IMPICA. Figure 10b compares the average cache miss latency between the baseline last-level cache and the IMPICA cache. On average, the cache miss latency of IMPICA is only 60–70% of the baseline cache miss latency. This is because IMPICA leverages the faster and wider TSVs in 3D-stacked memory as opposed to the narrow, high-latency DRAM interface used by the CPU.

Third, as Figure 10c shows, IMPICA effectively utilizes the internal memory bandwidth of 3D-stacked memory, which is cheap and abundant. There are two reasons for high bandwidth utilization: (1) IMPICA runs much faster than the baseline so it generates more traffic within the same amount time; and (2) IMPICA always accesses memory at a larger granularity, retrieving each full node in a linked data structure with a single memory request, while a CPU issues multiple requests for each node as it can fetch only one cache line at a time. The CPU can avoid using some of its limited memory bandwidth by skipping some fields in the data structure that are not needed for the current loop iteration. For example, some keys and pointers in a B-tree node can be skipped



(a) Total TLB misses per kilo-instruction (MPKI)

(b) Average last-level cache miss latency

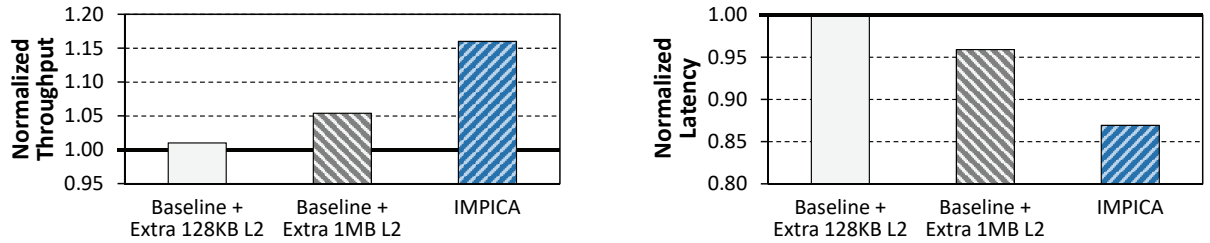
(c) Total DRAM memory bandwidth utilization

Fig. 10. Key architectural statistics for the evaluated microbenchmarks. Figure adapted from [67].

whenever a match is found. In contrast, IMPICA utilizes the wide internal bandwidth of 3D-stacked memory to retrieve a full node on each access.

We conclude that IMPICA is effective at significantly improving the performance of important linked data structures.

3.6.2 Real Database Throughput and Latency. Figure 11 presents two key performance metrics for our evaluation of DBx1000: *database throughput* and *database latency*. *Database throughput* represents how many transactions are completed within a certain period, while *database latency* is the average time to complete a transaction. We normalize the results of three configurations to the baseline. As mentioned earlier, the die area increase of IMPICA is similar to a 128KB cache. To understand the effect of additional LLC space better, we also show the results of adding 1MB of cache, which takes about 8× the area of IMPICA, to the baseline. We make two observations from our analysis of DBx1000.



(a) Database transaction throughput

(b) Database transaction latency

Fig. 11. Performance results for DBx1000, normalized to the baseline. Figure adapted from [67].

First, IMPICA improves the overall database throughput by 16% and reduces the average database transaction latency by 13%. The performance improvement is due to three reasons: (1) database indexing becomes faster with IMPICA, (2) offloading database indexing to IMPICA reduces the TLB and cache contention due to pointer chasing in the CPU, and (3) the CPU can do other useful tasks in parallel while waiting for IMPICA. Note that our profiling results in Figure 2 show that DBx1000 spends 19% of its time on pointer chasing. Therefore, a 16% overall improvement is very close to the upper bound that *any* pointer chasing accelerator can achieve for this database.

Second, IMPICA yields much higher database throughput than simply providing additional cache capacity. IMPICA improves the database throughput by 16%, while an extra 128KB of cache (with a similar area overhead as IMPICA) does so by only 2%, and an extra 1MB of cache (8× the area of IMPICA) by only 5%.

We conclude that by accelerating the fundamental pointer chasing operation, IMPICA efficiently improves the performance of a sophisticated real data-intensive workload.

3.6.3 Sensitivity to the IMPICA TLB Size and Page Table Design. To understand the effect of different TLB sizes and page table designs in IMPICA, we evaluate the speedup in the amount of time spent on address translation for IMPICA when different IMPICA TLB sizes (32 and 64 entries) and accelerator page table structures (the baseline 4-level page table; and the region-based page table, or RPT) are used inside the accelerator. Figure 12 shows the speedup in address translation time relative to IMPICA with a 32-entry TLB and the conventional 4-level page table. Two observations are in order.

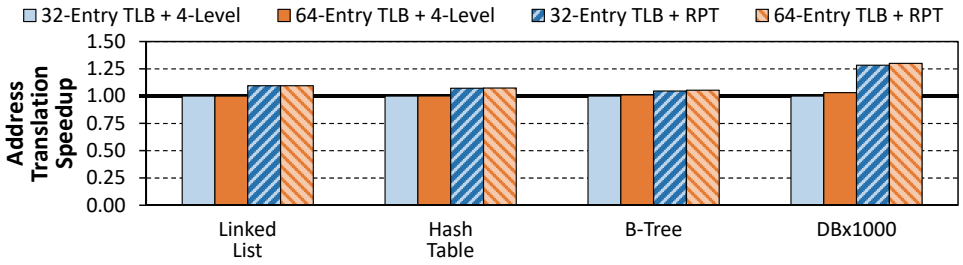


Fig. 12. Speedup of address translation with different TLB sizes and page table designs. Figure adapted from [67].

First, the performance of IMPICA is largely unaffected from small changes in the IMPICA TLB size. Doubling the IMPICA TLB entries from 32 to 64 barely improves the address translation time. This observation reflects the irregular nature of pointer chasing. Second, the benefit of the RPT is much more significant in a sophisticated workload (DBx1000) than in microbenchmarks. This is because the working set size of the microbenchmarks is much smaller than that of the database system. When the working set is small, the operating system needs only a small number of page table entries in the first and second levels of a traditional page table. These entries are used frequently, so they stay in the IMPICA cache much longer, reducing the address translation overhead. This caching benefit goes away with a larger working set, which would require a significantly larger TLB and IMPICA cache to reap locality benefits. The benefit of RPT is more significant in such a case because RPT does not rely on this caching effect. Its region table is *always* small irrespective of the workload working set size and it has fewer page table levels. Thus, we conclude that RPT is a much more efficient and high-performance page table design for our IMPICA accelerator than conventional page table design.

3.6.4 Energy Efficiency. Figure 13 shows the system power and system energy consumption for the microbenchmarks and DBx1000. We observe that the overall system *power* increases by 5.6% on average, due to the addition of IMPICA and higher utilization of internal memory bandwidth. However, as IMPICA significantly reduces the execution time of the evaluated workloads, the overall system *energy* consumption reduces by 41%, 24%, and 10% for the microbenchmarks, and by 6% for DBx1000. We conclude that IMPICA is an energy-efficient accelerator for pointer chasing.

3.7 Summary of IMPICA

We introduce the design and evaluation of an *in-memory accelerator*, called IMPICA, for performing pointer chasing operations in 3D-stacked memory. We identify two major challenges in the design of such an in-memory

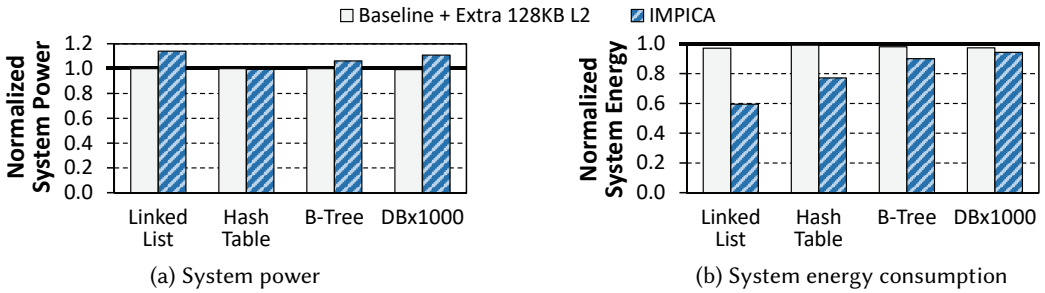


Fig. 13. Effect of IMPICA on system power (a) and system energy consumption (b). Figure 13b adapted from [67].

accelerator: (1) the *parallelism challenge* and (2) the *address translation challenge*. We provide new solutions to these two challenges: (1) *address-access decoupling* solves the parallelism challenge by decoupling the address generation from memory accesses in pointer chasing operations and exploiting the idle time during memory accesses to execute multiple pointer chasing operations in parallel, and (2) the *region-based page table* in 3D-stacked memory solves the address translation challenge by tracking only those limited set of virtual memory regions that are accessed by pointer chasing operations. Our evaluations show that for both commonly-used linked data structures and a real database application, IMPICA significantly improves both performance and energy efficiency. We conclude that IMPICA is an efficient and effective accelerator design for pointer chasing. We also believe that the two challenges we identify (parallelism and address translation) exist in various forms in other in-memory accelerators (e.g., for graph processing), and, therefore, our solutions to these challenges can be adapted for use by a broad class of (in-memory) accelerators. We believe ample future work potential exists on examining other solutions for these two challenges as well as our solutions for them within the context of other in-memory accelerators, such as those described in [2, 20, 68, 92, 93, 195, 196, 199, 200]. We also believe that examining solutions like IMPICA for other, non-in-memory accelerators is a promising direction to examine.

4 LAZYPIM: AN EFFICIENT CACHE COHERENCE MECHANISM FOR PROCESSING-IN-MEMORY

As discussed in Section 2.2, cache coherence is a major challenge for PIM architectures, as traditional coherence cannot be performed along the off-chip memory channel without potentially undoing the benefits of high-bandwidth and low-energy PIM execution. To work around the limitations presented by cache coherence, most prior works assume a limited amount of sharing between the PIM kernels and the processor threads of an application. Thus, they sidestep coherence by employing solutions that restrict PIM to execute on non-cacheable data (e.g., [2, 47, 52, 149, 243]) or force processor cores to flush or not access any data that could *potentially* be used by PIM (e.g., [3, 4, 28, 47, 51, 59, 67, 68, 163, 172, 195, 196, 199, 200]). In fact, the IMPICA accelerator design, described in Section 3, falls into the latter category.

To understand the trade-offs that can occur by sidestepping coherence, we analyze several data-intensive applications. We make two *key observations* based on our analysis: (1) some portions of the applications are better suited for execution in processor threads, and these portions often concurrently access the same region of data as the PIM kernels, leading to *significant data sharing*; and (2) poor handling of coherence eliminates a significant portion of the performance benefits of PIM. As a result, we find that a good coherence mechanism is *required* to ensure the correct execution of the program while maintaining the benefits of PIM (see Section 4.2). **Our goal** in this section is to describe a cache coherence mechanism for PIM architectures that *logically behaves* like traditional coherence, but retains all of the benefits of PIM.

To this end, we propose *LazyPIM*, a new cache coherence mechanism that efficiently batches coherence messages sent by the PIM processing logic. During PIM kernel execution, a PIM core *speculatively* assumes that it has acquired coherence permissions without sending a coherence message, and maintains all data updates speculatively in its cache. Only when the kernel finishes execution, the processor receives compressed information from the PIM core, and checks if any coherence conflicts occurred. If a conflict exists (see Section 4.3), the dirty cache lines in the processor are flushed, and the PIM core rolls back and re-executes the kernel. Our execution model *for PIM processing logic* is similar to *chunk-based execution* [24] (i.e., each *batch* of consecutive instructions executes atomically), which prior work has harnessed for various purposes [24, 38, 61, 161, 174, 192]. Unlike past works, however, the processor in LazyPIM executes conventionally and *never rolls back*, which can make it easier to enable PIM.

We make the following key contributions in this section:

- We propose a new hardware coherence mechanism for PIM. Our approach (1) reduces the off-chip traffic between the PIM cores and the processor, (2) avoids the costly overheads of prior approaches to provide coherence for PIM, and (3) retains the same logical coherence behavior as architectures without PIM to keep programming simple.
- LazyPIM improves average performance by 49.1% (coming within 5.5% of an ideal PIM mechanism), and reduces off-chip traffic by 58.8%, over the best prior coherence approach.

4.1 Baseline PIM Architecture

In our evaluation, we assume that the compute units inside memory consist of simple *in-order* cores. These PIM cores, which are ISA-compatible with the out-of-order processor cores, are much weaker in terms of performance, as they lack large caches and sophisticated ILP techniques, but are more practical to implement within the DRAM logic layer, as we discussed earlier in Section 1.1. Each PIM core has private L1 I/D caches, which are kept coherent using a MESI directory [23, 167] within the DRAM logic layer. A second directory in the processor acts as the main coherence point for the system, interfacing with both the processor cache and the PIM coherence directory. Like prior PIM works [2–4, 47, 52, 68, 172], we assume that direct segments [13] are used for PIM data, and that PIM kernels operate only on physical addresses.

4.2 Motivation for Coherence Support in PIM

Applications benefit the most from PIM execution when their memory-intensive parts, which often exhibit poor locality and contribute to a large portion of execution time, are dispatched to PIM processing logic. On the other hand, compute-intensive parts or those parts that exhibit high locality *must remain on the processor cores* to maximize performance [3, 68].

Prior work mostly assumes that there is only a limited amount of sharing between the PIM kernels and the processor. However, *this is not the case* for many important applications, such as graph and database workloads. For example, in multithreaded graph frameworks, each thread performs a graph algorithm (e.g., connected components, PageRank) on a shared graph [2, 209, 237]. We study a number of these algorithms [209], and find that (1) only certain portions of each algorithm are well suited for PIM, and (2) the PIM kernels and processor threads access the shared graph and intermediate data structures concurrently. Another example is modern in-memory databases that support Hybrid Transactional/Analytical Processing (HTAP) workloads [144, 193, 201, 219]. The analytical portions of these databases are well suited for PIM execution [99, 148, 234]. In contrast, even though transactional queries access the *same* data, they perform better if they are executed on the main processor (i.e., the CPU), as they are short-lived and latency sensitive, accessing only a few rows each. Thus, concurrent accesses from both PIM kernels (analytics) and processor threads (transactions) are inevitable.

The shared data needs to remain coherent between the processor and PIM cores. Traditional, or *fine-grained*, coherence protocols (e.g., MESI [23, 167]) have several qualities well suited for pointer-intensive data structures, such as those in graph workloads and databases. Fine-grained coherence allows the processor or PIM to acquire permissions for only the pieces of data that are *actually accessed*. In addition, fine-grained coherence can ease programmer effort when developing PIM applications, as multithreaded programs already use this programming model. Unfortunately, if a PIM core participates in traditional coherence, it would have to send a message for *every cache miss* to the processor over a narrow shared interconnect (we call this type of interconnect traffic as *PIM coherence traffic*).

We study four mechanisms to evaluate how coherence protocols impact PIM: (1) *CPU-only*, a baseline where PIM is disabled; (2) *FG*, fine-grained coherence, which is the MESI protocol, variants of which are employed in many state-of-the-art systems; (3) *CG*, coarse-grained lock based coherence, where PIM cores gain *exclusive* access to all PIM data during PIM kernel execution; and (4) *NC*, non-cacheable, where the PIM data is not cacheable in the CPU. We describe CG and NC in more detail below. Figure 14 shows the speedup of PIM with these four mechanisms for certain graph workloads, normalized to CPU-only.[4] To illustrate the impact of inefficient coherence mechanisms, we also show the performance of an *ideal* mechanism where there is no performance penalty for coherence (*Ideal-PIM*). As shown in Figure 14, employing PIM with a state-of-the-art fine-grained coherence (*FG*) mechanism *always* performs worse than CPU-only execution.

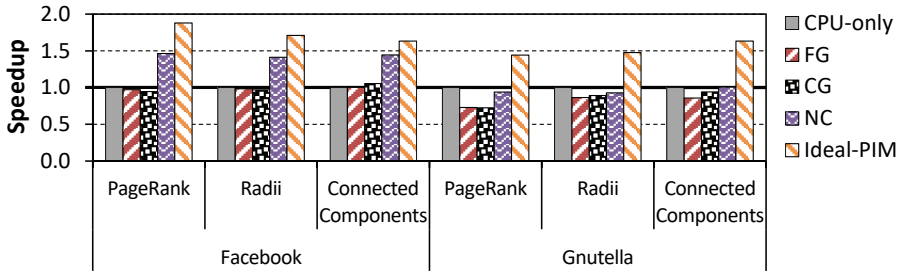


Fig. 14. PIM speedup with 16 threads, normalized to CPU-only, with three different and ideal coherence mechanisms.[4] Figure adapted from [22].

To reduce the impact of PIM coherence traffic, there are three general alternatives to fine-grained coherence for PIM execution: (1) coarse-grained coherence, (2) coarse-grained locks, and (3) making PIM data non-cacheable in the processor. We briefly examine these alternatives.

Coarse-Grained Coherence. One approach to reduce PIM coherence traffic is to maintain a single coherence entry for *all* of the PIM data. Unfortunately, this can still incur high overheads, as the processor must flush *all* of the dirty cache lines within the PIM data region *every time* the PIM core acquires permissions, *even if the PIM kernel may not access most of the data*. For example, with just four processor threads, the number of cache lines flushed for PageRank is 227x the number of lines *actually required by the PIM kernel*.[4] Coherence at a smaller granularity, such as page-granularity [51], does not cause flushes for pages not accessed by the PIM kernel. However, many data-intensive applications perform *pointer chasing*, where a large number of pages are accessed non-sequentially, but only a *few lines* in each page are used, forcing the processor to flush *every* dirty page.

Coarse-Grained Locks. Another drawback of coarse-grained coherence is that data can ping-pong between the processor and the PIM cores whenever the PIM data region is concurrently accessed by both. *Coarse-grained*

[4]See Section 4.5 for our experimental evaluation methodology.

locks avoid ping-ponging by having the PIM cores acquire *exclusive* access to a region for the duration of the PIM kernel. However, coarse-grained locks greatly restrict performance. Our application study shows that PIM kernels and processor threads often work in parallel on the same data region, and coarse-grained locks frequently cause thread serialization. PIM with coarse-grained locks (*CG* in Figure 14) performs 8.4% *worse*, on average, than CPU-only execution. We conclude that using coarse-grained locks is not suitable for many important applications for PIM execution.

Non-Cacheable PIM Data. Another approach sidesteps coherence by marking the PIM data region as *non-cacheable* in the processor [2, 47, 52, 149, 243], so that DRAM always contains up-to-date data. For applications where PIM data is almost exclusively accessed by the PIM processing logic, this incurs little penalty, but for many applications, the processor also accesses PIM data often. For our graph applications with a representative input (arXiV),[4] the processor cores generate 42.6% of the total number of accesses to PIM data. With so many processor accesses, making PIM data non-cacheable results in a high performance and bandwidth overhead. As shown in Figure 14, though marking PIM data as non-cacheable (*NC*) sometimes performs better than CPU-only, it still loses up to 62.7% (on average, 39.9%) of the improvement of Ideal-PIM. Therefore, while this approach avoids the overhead of coarse-grained mechanisms, it is a poor fit for applications that rely on processor involvement, and thus restricts the applications where PIM is effective.

We conclude that prior approaches to PIM coherence eliminate a significant portion of the benefits of PIM when data sharing occurs, due to their high coherence overheads. In fact, they sometimes cause PIM execution to *consistently degrade performance*. Thus, an *efficient* alternative to fine-grained coherence is necessary to retain PIM benefits across a wide range of applications.

4.3 LazyPIM Mechanism for Efficient PIM Coherence

Our goal is to design a coherence mechanism that maintains the logical behavior of traditional coherence while retaining the large performance benefits of PIM. To this end, we propose *LazyPIM*, a new coherence mechanism that lets PIM kernels *speculatively* assume that they have the required permissions from the coherence protocol, *without* actually sending off-chip messages to the main (processor) coherence directory during execution. Instead, coherence states are updated only *after* the PIM kernel completes, at which point the PIM core transmits a single batched coherence message (i.e., a compressed *signature* containing *all* addresses that the PIM kernel read from or wrote to) back to the processor coherence directory. The directory checks to see whether any *conflicts* occurred. If a conflict exists, the PIM kernel *rolls back* its changes, conflicting cache lines are written back by the processor to DRAM, and the kernel re-executes. If no conflicts exist, speculative data within the PIM core is *committed*, and the processor coherence directory is updated to reflect the data held by the PIM core. Note that in LazyPIM, the processor *always* executes *non-speculatively*, which ensures minimal changes to the processor design, thereby likely enabling easier adoption of PIM.

LazyPIM avoids the pitfalls of the coherence mechanisms discussed in Section 4.2 (FG, CG, NC). With its compressed signatures, LazyPIM causes much less PIM coherence traffic than traditional fine-grained coherence. Unlike coarse-grained coherence and coarse-grained locks, LazyPIM checks coherence *only after* it completes PIM execution, avoiding the need to unnecessarily flush a large amount of data. Unlike non-cacheable, LazyPIM allows processor threads to cache the data used by PIM kernels within the processor cores as well, avoiding the need for the processor to perform a large number of off-chip accesses that can hurt performance greatly. LazyPIM also allows for efficient concurrent execution of processor threads and PIM kernels: by executing speculatively, the PIM cores do *not* invoke coherence requests during concurrent execution, avoiding data ping-ponging between the PIM cores and the processor.

Conflicts. In LazyPIM, a PIM kernel *speculatively* assumes during execution that it has coherence permissions on a cache line, without checking the processor coherence directory. In the meantime, the processor continues to

execute *non-speculatively*. To resolve PIM kernel speculation, LazyPIM provides *coarse-grained atomicity*, where all PIM memory updates are treated as if they *all* occur *at the moment that a PIM kernel finishes execution*. (We explain how LazyPIM enables this in Section 4.4.) If, before the PIM kernel finishes, the processor updates a cache line that the PIM kernel read during its execution, a *conflict* occurs. LazyPIM detects and handles all potential conflicts once the PIM kernel finishes executing.

Figure 15 shows an example timeline where a PIM kernel is launched on PIM core PIM0 while execution continues on processor cores CPU0 and CPU1. Due to the use of coarse-grained atomicity, PIM kernel execution behaves as if *the entire kernel's memory accesses* take place at the moment coherence is checked (i.e., at the end of kernel execution), *regardless of the actual time at which the kernel's accesses are performed*. Therefore, for *every* cache line read by PIM0, if CPU0 or CPU1 modify the line before the coherence check occurs, PIM0 unknowingly uses stale data, leading to incorrect execution. Figure 15 shows two examples of this: (1) CPU0's write to line C *during* kernel execution; and (2) CPU0's write to line A *before* kernel execution, which was not written back to DRAM. To detect such conflicts, we record the addresses of processor writes and PIM kernel reads into two signatures, and then check to see if any addresses in them match (i.e., conflict) *after* the PIM kernel finishes (see Section 4.4.2).

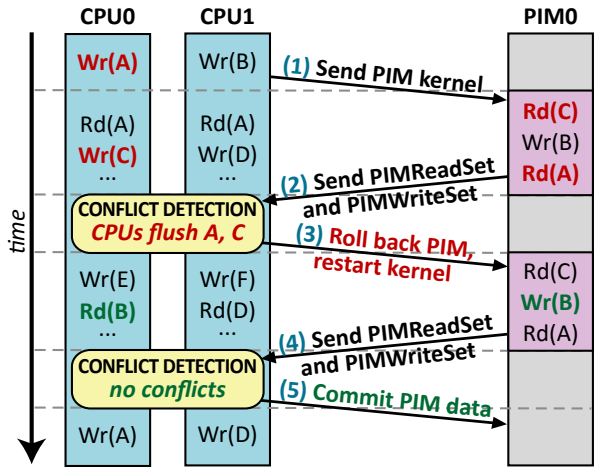


Fig. 15. Example timeline of LazyPIM coherence sequence. Figure reproduced from [22].

If the PIM kernel writes to a cache line that is subsequently read by the processor before the kernel finishes (e.g., the second write by PIM0 to line B in Figure 15), this is *not* a conflict. With coarse-grained atomicity, any read by the processor during PIM execution is ordered *before* the PIM kernel's write. LazyPIM ensures that the processor cannot read the PIM kernel's writes, by marking the PIM kernel writes as speculative inside the PIM processing logic until the kernel finishes (see Section 4.4.2). This is also the case when the processor and a PIM kernel write to the *same* cache line. Note that this ordering does not violate consistency models, such as sequential consistency.[5]

If the PIM kernel writes to a cache line that is subsequently *written to* by the processor before the kernel finishes, this is *not* a conflict. With coarse-grained atomicity, any write by the processor during PIM kernel execution is ordered before the PIM core's write since the PIM core write effectively takes place *after the PIM*

[5]A thorough treatment of memory consistency [106] is outside the scope of this work. Our goal is to deal with the coherence problem in PIM, not handle consistency issues.

kernel finishes. When the two writes modify different words in the same cache line, LazyPIM uses a per-word dirty bit mask in the PIM L1 cache to merge the writes, similar to prior work [119]. Note that the dirty bit mask is only in the PIM L1 cache; processor caches remain unchanged.

More details on the operation of the LazyPIM coherence mechanism are provided in our arXiv paper [21].

4.4 Architectural Support for LazyPIM

4.4.1 LazyPIM Programming Model. We provide a simple interface to port applications to LazyPIM. We show the implementation of a simple LazyPIM kernel within a program in Code Example 1. The programmer selects the portion(s) of the code to execute on PIM cores, using two macros (`#PIM_begin` and `#PIM_end`). The compiler converts the macros into instructions that we add to the ISA, which *trigger* and *end* PIM kernel execution. LazyPIM also needs to know which parts of the allocated data *might* be accessed by the PIM cores, which we refer to as the *PIM data region*.[6] We assume that either the programmer or the compiler can annotate all of the PIM data region using compiler directives or a PIM memory allocation API (`@PIM`). This information is saved in the page table using per-page flag bits, via communication to the system software using the system call interface.

```
1  PageRankCompute(@PIM Graph GA) { // GA is accessed by PIM cores in edgeMap()
2    const int n = GA.n;            // not accessed by PIM cores
3    const double damping = 0.85, epsilon = 0.0000001; // not accessed by PIM cores
4    @PIM double* p_curr, p_next;   // accessed by PIM cores in edgeMap()
5    @PIM bool* frontier;           // accessed by PIM cores in edgeMap()
6    @PIM vertexSubset Frontier(n, n, frontier); // accessed by PIM in edgeMap()
7    double L1_norm;                // not accessed by PIM cores
8    long iter = 0;                 // not accessed by PIM cores
9    ...
10   while(iter++ < maxIters) {
11     #PIM_begin
12       // only the edgeMap() function is offloaded to the PIM cores
13       vertexSubset output =
               edgeMap(GA, Frontier, @PIM PR_F<vertex>(p_curr, p_next, GA.V), 0);
14       // PR_F<vertex> object allocated during edgeMap() call, needs annotation
15     #PIM_end
16     vertexMap(Frontier, PR_Vertex_F(p_curr, p_next, damping, n));  // run on CPU
17
18     // compute L1-norm between p_curr and p_next
19     L1_norm = fabs(p_curr - p_next);  // run on CPU
20     if(L1_norm < epsilon) break;      // run on CPU
21     ...
22   }
23   Frontier.del();
24 }
```

Code Example 1. Example PIM program implementation. Modifications for PIM execution are shown in bold.

Code Example 1 shows a portion of the compute function used by PageRank, as modified for execution with LazyPIM. All of our modifications are shown in bold. In this example, we want to execute only the `edgeMap()` function (Line 13) on the PIM cores. To ensure that LazyPIM tracks all data accessed during the `edgeMap()`

[6]The programmer should be conservative in identifying PIM data regions, and should not miss *any possible data* that may be touched by a PIM core. If any data *not marked* as PIM data is accessed by the PIM core, the program can produce incorrect results.

call, we mark all of this data using `@PIM`, including any objects passed by value (e.g., `GA` on Line 1), any objects allocated in the function (e.g., those on Lines 4–6), and any objects allocated during functions that are executed on the PIM cores (e.g., the `PR_F<vertex>` object on Line 13). To tell the compiler that we want to execute only `edgeMap()` on the PIM cores, we surround it with the `#PIM_begin` and `#PIM_end` compiler directives on Lines 11 and 15, respectively. No other modifications are needed to execute our example code with LazyPIM.

4.4.2 Speculative Execution. When an application reaches a *PIM kernel trigger* instruction, the processor dispatches the kernel's starting PC to a free PIM core. The PIM core *checkpoints* the starting PC and registers, and starts executing the kernel. The kernel *speculatively* assumes that it has coherence permissions for *every* line it accesses, without *actually* checking the processor directory. We add a one-bit flag to each line in the PIM core cache, to mark all data updates as speculative. If a speculative line is selected for eviction, the core rolls back to the starting PC and discards the updates.

LazyPIM tracks three sets of addresses during PIM kernel execution. These are recorded into three *signatures*, as shown in Figure 16: (1) the *CPUWriteSet* (all *CPU writes* to the PIM data region), (2) the *PIMReadSet* (all *PIM reads*), and (3) the *PIMWriteSet* (all *PIM writes*). When the kernel starts, the dirty lines in the processor cache containing PIM data are recorded in the CPUWriteSet, by scanning the tag store (potentially using a Dirty-Block Index [198]). The processor uses the page table flag bits from Section 4.4.1 to identify which writes need to be added to the CPUWriteSet during kernel execution. The PIMReadSet and PIMWriteSet are updated for *every* read and write performed by the PIM kernel. When the kernel finishes execution, the three signatures are used to resolve speculation (see Section 4.4.3)

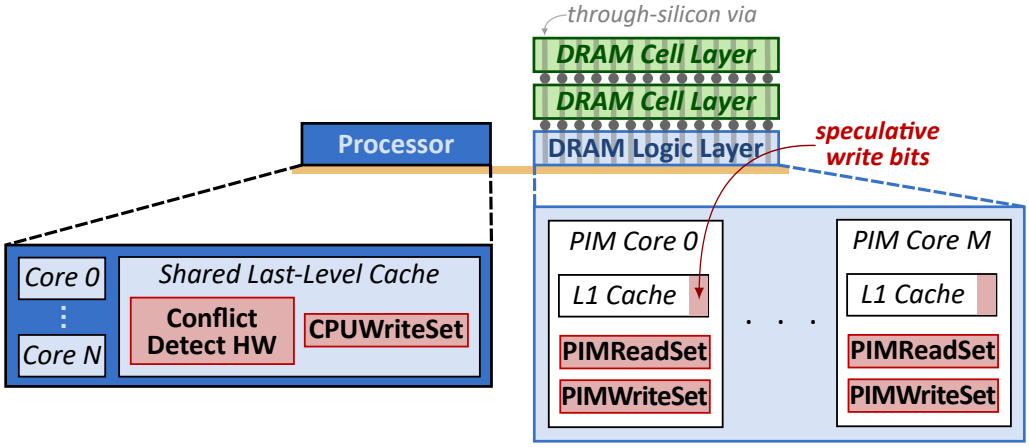


Fig. 16. High-level additions (in bold) to PIM architecture to support LazyPIM. Figure adapted from [22].

The signatures use parallel Bloom filters [19], which employ simple Boolean logic to hash multiple addresses into a single (256B) fixed-length register. If the speculative coherence requests were sent back to the processor without any sort of compression at the end of PIM kernel execution, the coherence messages would still consume a large amount of off-chip traffic, nullifying most of the benefits of the speculation. Bloom filters allow LazyPIM to compress these coherence messages into a much smaller size, while guaranteeing that there are no false negatives [19] (i.e., no coherence messages are lost during compression). The addresses of *all* data accessed speculatively by the PIM cores can be extracted and compared from the Bloom filter [19, 24]. The hashing introduces a limited number of false positives, with the false positive rate increasing as we store more addresses in a single fixed-length Bloom filter. In our evaluated system, each signature is 256B long, and can store up to 607

addresses without exceeding a 20.0% false positive rate (with *no* false negatives). To store more addresses, we use multiple filters to guarantee an upper bound on the false positive rate.

4.4.3 Handling Conflicts. As Figure 15 shows, we need to detect conflicts that occur during PIM kernel execution. In LazyPIM, when the kernel finishes executing, both the PIMReadSet and PIMWriteSet are sent back to the processor.

If no matches are detected between the PIMReadSet and the CPUWriteSet (i.e., no conflicts have occurred), PIM kernel *commit* starts. Any addresses (including false positives) in the PIMWriteSet are invalidated from the processor cache. A message is then sent to the PIM core, allowing it to write its speculative cache lines back to DRAM. During the commit process, all coherence directory entries for the PIM data region are locked to ensure atomicity of commit. Finally, all signatures are erased.

If an overlap is found between the PIMReadSet and the CPUWriteSet, a conflict may have occurred. At this point, only the dirty lines in the processor that match in the PIMReadSet are flushed back to DRAM. During this flush, all PIM data directory entries are locked to ensure atomicity. Once the flush completes, a message is sent to the PIM core, telling it to invalidate all speculative cache lines, and to *roll back* the PC to the checkpointed value. Now that all possibly conflicting cache lines are written back to DRAM, all signatures are erased, and the PIM core *restarts* the kernel. After re-execution of the PIM kernel finishes, conflict detection is performed again.

Note that during the commit process, processor cores do not stall unless they access the same data accessed by PIM processing logic. LazyPIM guarantees forward progress by acquiring a lock for each line in the PIMReadSet after a number of rollbacks (we empirically set this number to three rollbacks). This simple mechanism ensures there is no livelock even if the sharing of speculative data among PIM cores might create a cyclic dependency. Note that rollbacks are caused by CPU accesses to conflicting addresses, and not by the sharing of speculative data between PIM cores. As a result, once we lock conflicting addresses following three rollbacks, the PIM cores will not roll back again as there will be no conflicts, guaranteeing forward progress.

4.4.4 Hardware Overhead. LazyPIM's overhead consists mainly of (1) 1 bit per page (0.003% of DRAM capacity) and 1 bit per TLB entry for the page table flag bits (Section 4.4.1); (2) a 0.2% increase in PIM core L1 size to mark speculative data (Section 4.4.2); (3) a 1.6% increase in PIM core L1 size for the dirty bit mask (Section 4.3); and (4) in the worst case, 12KB for the signatures per PIM core (Section 4.4.2). This overhead can be greatly optimized (as part of future work): for PIM kernels that need multiple signatures, we could instead divide the kernel into smaller chunks where each chunk's addresses fit in a single signature, lowering signature overhead to 512B. We leave a detailed evaluation of LazyPIM hardware overhead optimization to future work. Some ideas related to this and a detailed hardware overhead analysis are presented in our arXiv paper [21].

4.5 Methodology for LazyPIM Evaluation

We study two types of data-intensive applications: graph workloads and databases. We use three Ligra [209] graph applications (PageRank, Radii, Connected Components), with input graphs constructed from real-world network datasets [217]: Facebook, arXiV High Energy Physics Theory, and Gnutella25 (peer-to-peer). We also use an in-house prototype of a modern in-memory database (IMDB) [144, 193, 201, 219] that supports HTAP workloads. Our transactional workload consists of 200K transactions, each randomly performing reads or writes on a few randomly-chosen tuples. Our analytical workload consists of 256 analytical queries that use the select and join operations on randomly-chosen tables and columns.

PIM kernels are selected from these applications with the help of OProfile [165]. We conservatively select candidate PIM kernels, choosing portions of functions where the application (1) spends the majority of its cycles, and (2) generates the majority of its last-level cache misses. From these candidates, we pick kernels that we believe minimize the coherence overhead for each evaluated mechanism, by minimizing data sharing between

the processor and PIM processing logic. We modify each application to ship the selected PIM kernels to the PIM cores. We manually annotate the PIM data set.

For our evaluations, we modify the gem5 simulator [18]. We use the x86-64 architecture in full-system mode, and use DRAMSim2 [185] to perform detailed timing simulation of DRAM. Table 3 shows our system configuration.

Table 3. Evaluated system configuration for LazyPIM evaluation.

Main Processor (CPU)	
ISA	x86-64
Core Configuration	4–16 cores, 2 GHz, 8-wide issue
Operating System	64-bit Linux from Linaro [127]
L1 I/D Cache	64KB per core, private, 4-way associative, 64B blocks, 2-cycle lookup
L2 Cache	2MB, shared, 8-way associative, 64B blocks, 20-cycle lookup
Cache Coherence	MESI directory [23, 167]
PIM Cores	
ISA	x86-64
Core Configuration	4–16 cores, 2 GHz, 1-wide issue
L1 I/D Cache	64KB per core, private, 4-way associative, 64B blocks, 2-cycle lookup
Cache Coherence	MESI directory [23, 167]
Main Memory Parameters	
Memory Configuration	HMC 2.0 [72], one 4GB cube, 16 vaults per cube, 16 banks per vault, FR-FCFS scheduler [182, 249]

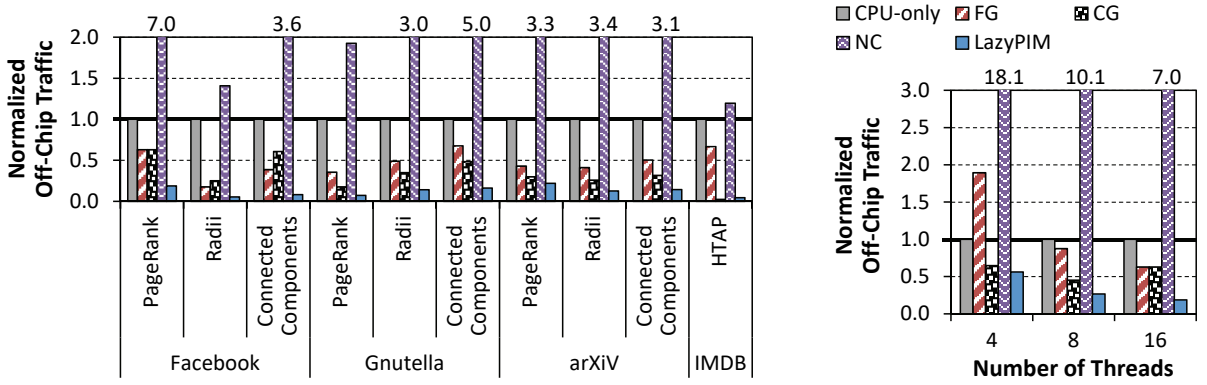
4.6 Evaluation of LazyPIM

We first analyze the off-chip traffic reduction of LazyPIM. This off-chip reduction leads to bandwidth and energy savings. We then analyze LazyPIM's effect on system performance. We show system performance results normalized to a processor-only baseline (*CPU-only*, as defined in Sec. 4.2), and compare LazyPIM's performance with using fine-grained coherence (*FG*), coarse-grained locks (*CG*), or non-cacheable data (*NC*) for PIM data.

4.6.1 Off-Chip Memory Traffic. Figure 17a shows the normalized off-chip memory traffic of the PIM coherence mechanisms for a 16-core architecture (with 16 processor cores and 16 PIM cores) Figure 17b shows the normalized off-chip memory traffic as the number of threads increases, for PageRank using the Facebook graph. LazyPIM significantly reduces the *overall* off-chip traffic (up to 81.2% over CPU-only, 70.1% over FG, 70.2% over CG, and 97.3% over NC), and scales better with thread count. LazyPIM reduces off-chip memory traffic by 58.8%, on average, over CG, the best prior approach in terms of off-chip traffic.

CG has greater traffic than LazyPIM, the majority of which is due to having to flush dirty cache lines before each PIM kernel invocation. Due to false sharing, the number of flushes scales *superlinearly* with thread count (not shown), increasing 13.1x from 4 to 16 threads. LazyPIM avoids this cost with speculation, as *only* the *necessary* flushes are performed *after* the PIM kernel finishes execution. As a result, it reduces the flush count (e.g., by 94.0% for 16-thread PageRank using Facebook), and thus lowers overall off-chip memory traffic (by 50.3% for our example).

NC suffers from the fact that *all* processor accesses to PIM data must go to DRAM, increasing average off-chip memory traffic by 3.3x over CPU-only. NC off-chip memory traffic also scales poorly with thread count, as more processor threads generate a greater number of accesses. In contrast, LazyPIM allows processor cores to cache PIM data, by enabling coherence efficiently.

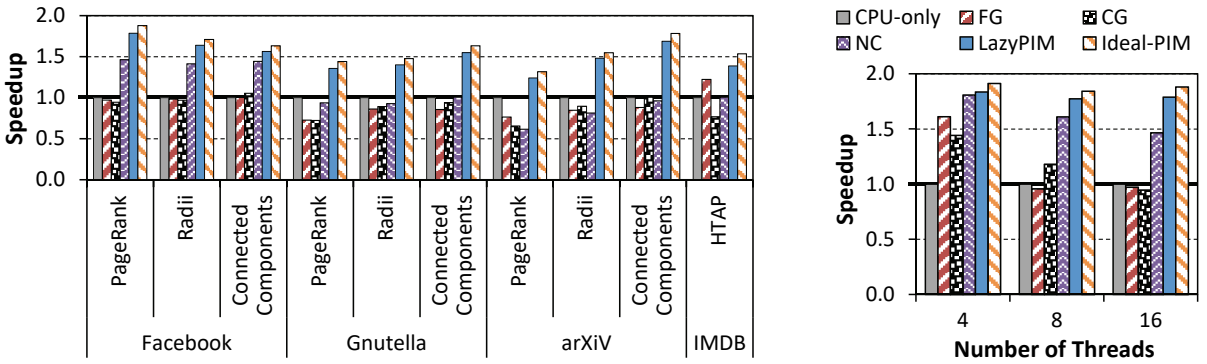


(a) 16-thread off-chip memory traffic

(b) Off-chip memory traffic sensitivity to thread count for PageRank

Fig. 17. Effect of LazyPIM on off-chip memory traffic, normalized to CPU-only. Figure adapted from [22].

4.6.2 Performance. Figure 18a shows the performance improvement for 16 threads. Without any coherence overhead, Ideal-PIM significantly outperforms CPU-only across *all* applications, showing PIM's potential on these workloads. Poor handling of coherence by FG, CG, and NC leads to drastic performance losses compared to Ideal-PIM, indicating that an efficient coherence mechanism is essential for PIM performance. For example, in some cases, NC and CG actually perform *worse* than CPU-only, and for PageRank running on the Gnutella graph, all prior mechanisms degrade performance. In contrast, LazyPIM consistently retains most of Ideal-PIM's benefits for all applications, coming within 5.5% on average. LazyPIM outperforms all of the other approaches, improving over the best-performing prior approach (NC) by 49.1%, on average.



(a) Speedup for all applications with 16 threads

(b) Speedup sensitivity to thread count for Gnutella

Fig. 18. Speedup of cache coherence mechanisms, normalized to CPU-only. Figure adapted from [22].

Figure 18b shows the performance of PageRank using Gnutella as we increase the thread count. LazyPIM comes within 5.5% of Ideal-PIM, which has no coherence overhead (as defined in Sec. 4.2), and improves performance by

73.2% over FG, 47.0% over CG, 29.4% over NC, and 39.4% over CPU-only, on average. With NC, the processor threads incur a large penalty for accessing DRAM frequently. CG suffers greatly due to (1) flushing dirty cache lines, and (2) blocking all processor threads that access PIM data during execution. In fact, processor threads are blocked for up to 73.1% of the total execution time with CG. With more threads, the negative effects of blocking worsen CG's performance. FG also loses a significant portion of Ideal-PIM's improvements, as it sends a large amount of off-chip messages. Note that NC, despite its high off-chip traffic, performs better than CG and FG, as it neither blocks processor cores nor slows down PIM execution.

One reason for the difference in performance between LazyPIM and Ideal-PIM is the number of conflicts that are detected at the end of PIM kernel execution. As we discuss in Section 4.4.3, any detected conflict causes a rollback, where the PIM kernel must be re-executed. We study the number of commits that contain conflicts for two representative 16-thread workloads: Components using the Enron graph, and HTAP-128 (results not shown). If we study an idealized version of full kernel commit, where no false positives exist, we find that a relatively high percentage of commits contain conflicts (47.1% for Components and 21.3% for HTAP). Using realistic signatures for full kernel commit, which includes the impact of false positives, the conflict rate increases to 67.8% for Components and 37.8% for HTAP. Despite the high number of commits that induce rollback, the overall performance impact of rollback is low, as LazyPIM comes within 5.5% of the performance of Ideal-PIM. We find that for all of our applications, a kernel never rolls back more than once, limiting the performance impact of conflicts. We can further improve the performance of LazyPIM by optimizing the commit process to reduce the rollback overhead, which we explore in our arXiv paper [21].

4.7 Summary of LazyPIM

We propose LazyPIM, a new cache coherence mechanism for PIM architectures. Prior approaches to PIM coherence generate very high off-chip traffic for important data-intensive applications. LazyPIM avoids this by avoiding coherence lookups *during* PIM kernel execution. The key idea is to use compressed coherence *signatures* to batch the lookups and verify correctness *after* the kernel completes. As a result of the more efficient approach to coherence employed by LazyPIM, applications that performed poorly under prior approaches to PIM coherence can now take advantage of the benefits of PIM execution. LazyPIM improves average performance by 49.1% (coming within 5.5% of an ideal PIM mechanism), and reduces off-chip traffic by 58.8%, over the best prior approach to PIM coherence while retaining the conventional multithreaded programming model.

5 RELATED WORK

We briefly survey related work in processing-in-memory, accelerator design, mechanisms for handling pointer chasing, and techniques for pointer chasing.

Early Processing-in-Memory (PIM) Proposals. Early proposals for PIM architectures had limited to no adoption, as the proposed logic integration was too costly and did not solve many of the obstacles facing the adoption of PIM. The earliest such proposals date from the 1970s, where small processing elements are combined with small amounts of RAM to provide a distributed array of memories that perform computation [208, 218]. Some of the other early works, such as EXECUBE [100], Terasys [56], IRAM [171], and Computational RAM [44, 45], add logic within DRAM to perform vector operations. Yet other early works, such as FlexRAM [82], DIVA [39], Smart Memories [140], and Active Pages [166], propose more versatile substrates that tightly integrate logic and reconfigurability within DRAM itself to increase flexibility and the available compute power.

Processing in 3D-Stacked Memory. With the advent of 3D-stacked memories, we have seen a resurgence of PIM proposals [133, 203]. Recent PIM proposals add compute units within the logic layer to exploit the high bandwidth available. These works primarily focus on the design of the underlying logic that is placed within memory, and in many cases propose special-purpose PIM architectures that cater only to a limited

set of applications. These works include accelerators for MapReduce [176], matrix multiplication [246], data reorganization [4], graph processing [2, 163], databases [12], in-memory analytics [51], genome sequencing [92, 93], data-intensive processing [58], consumer device workloads [20], and machine learning workloads [30, 89, 120]. Some works propose more generic architectures by adding PIM-enabled instructions [3], GPGPUs [68, 172, 243], single-instruction multiple-data (SIMD) processing units [149], or reconfigurable hardware [47, 52, 59] to memory.

Processing Using Memory. A number of recent works have examined how to perform memory operations directly within the memory array itself, which we refer to as *processing using memory* [203]. These works take advantage of inherent architectural properties of memory devices to perform operations in bulk. While such works can significantly improve computational efficiency within memory, they still suffer from many of the same programmability and adoption challenges that PIM architectures face, such as the address translation and cache coherence challenges that we focus on in this chapter. Mechanisms for processing using memory can perform a variety of functions, such as bulk copy and data initialization for DRAM [26, 28, 197, 200], bulk bitwise operations for DRAM [122, 195, 196, 199] and phase-change memory (PCM) [124], and simple arithmetic operations for SRAM [1, 80] and memristors [103–105, 121, 205].

Processing in the DRAM Module or Memory Controller. Several works have examined how to embed processing functionality near memory, but not within the DRAM chip itself. Such an approach can reduce the cost of PIM manufacturing, as the DRAM chip does not need to be modified or specialized for any particular functionality. However, these works (1) are often unable to take advantage of the high internal bandwidth of 3D-stacked DRAM, which reduces the efficiency of PIM execution, and (2) may still suffer from many of the same challenges faced by architectures that embed logic within the DRAM chip. Examples of this work include Chameleon [9], which proposes a method of integrating logic within the DRAM module but outside of the chip to reduce manufacturing costs, Gather-Scatter DRAM [201], which embeds logic within the memory controller to remap a single memory request across multiple rows and columns within DRAM, and work by Hashemi et al. [62, 63] to embed logic in the memory controller that accelerates dependent cache misses and performs runahead execution [154, 156, 158, 159].

Addressing Challenges to PIM Adoption. Recent work has examined design challenges for systems with PIM support that can affect PIM adoption. A number of these works improve PIM programmability, such as LazyPIM [21, 22], which provides efficient cache coherence support for PIM (as we described in detail in Section 4) the study by Sura et al. [221], which optimizes how programs access PIM data, and work by Liu et al. [131], which designs PIM-specific concurrent data structures to improve PIM performance. Other works tackle hardware-level design challenges, including IMPICA [67], which introduces in-memory support for address translation and pointer chasing (as we described in detail in Section 3), work by Hassan et al. [66] to optimize the 3D-stacked DRAM architecture for PIM, and work by Kim et al. [90] to enable the distribution of PIM data across multiple memory stacks.

Coherence for PIM Architectures. In order to avoid the overheads of fine-grained coherence, many prior works on PIM architectures design their systems in such a way that they do not need to utilize traditional coherence protocols. Instead, these works use one of two alternatives. Some works restrict PIM processing logic to execute on only non-cacheable data (e.g., [2, 47, 52, 149, 243]), which forces cores within the CPU to read PIM data directly from DRAM. Other works use coarse-grained coherence or coarse-grained locks, which force processor cores to not access any data that could *potentially* be used by the PIM processing logic, or to flush this data back to DRAM before the PIM kernel begins executing (e.g., [3, 4, 28, 47, 51, 59, 67, 67, 68, 163, 172, 195, 196, 199, 200]). Both of these approaches generates significant coherence overhead, as discussed in Section 4.2. Unlike these approaches, LazyPIM (Section 4) places no restriction on the way in which processor cores and PIM processing logic can access data. Instead, LazyPIM uses PIM-side coherence speculation and efficient coherence message

compression to provide cache coherence, which avoids the communication overheads associated with traditional coherence protocols.

Accelerators in CPUs. There have been various CPU-side accelerator proposals for database systems (e.g., [32, 99, 230, 231]) and key-value stores [126]. Widx [99] is a database indexing accelerator that uses a set of custom RISC cores in the CPU to accelerate hash index lookups. While a hash table is one of our data structures of interest, IMPICA (Section 3) differs from Widx in three major ways. First, it is an *in-memory* (as opposed to CPU-side) accelerator, which poses very different design challenges. Second, we solve the address translation challenge for in-memory accelerators, while Widx uses the CPU address translation structures. Third, we enable parallelism within a single accelerator core, while Widx achieves parallelism by replicating several RISC cores.

Prefetching for Linked Data Structures. Many works propose mechanisms to prefetch data in linked data structures to hide memory latency. These proposals are hardware-based (e.g., [33, 35, 69, 70, 78, 153, 155, 187, 242]), software-based (e.g., [128, 136, 186, 232, 238]), pre-execution-based (e.g., [34, 135, 213, 248]), or software/hardware-cooperative (e.g., [40, 83, 187]) mechanisms. These approaches have two major drawbacks. First, they usually rely on predictable traversal sequences to prefetch accurately. As a result, many of these mechanisms can become very inefficient if the linked data structure is complex or when access patterns are less regular. Second, the pointer chasing or prefetching is performed at the CPU cores or at the memory controller, which likely leads to pollution of the CPU caches and TLBs by these irregular memory accesses.

6 OTHER SYSTEM-LEVEL CHALLENGES FOR PIM ADOPTION

IMPICA (Section 3) and LazyPIM (Section 4) demonstrate the need for and gains that can be achieved by designing system-level solutions that are applicable across a wide variety of PIM architectures. In order for PIM to achieve widespread adoption, we believe there are a number of other system-level challenges that must be addressed. In this section, we discuss six research directions that aim towards solving these challenges: (1) the PIM programming model, (2) data mapping, (3) runtime scheduling support for PIM, (4) the granularity of PIM scheduling, (5) evaluation infrastructures and benchmark suites for PIM, and (6) applying PIM to emerging memory technologies.

PIM Programming Model. Programmers need a well-defined interface to incorporate PIM functionality into their applications. Determining the programming model for how a programmer should invoke and interact with PIM processing logic is an open research direction. Using a set of special instructions allows for very fine-grained control of when PIM processing logic is invoked, which can potentially result in a significant performance improvement. However, this approach can potentially introduce overheads while taking advantage of PIM, due to the need to frequently exchange information between 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. For example, one of our recent works [68] examines compiler-based mechanisms to decide what portions of code should be offloaded to PIM processing logic in a GPU-based system. Another recent work [172] examines system-level techniques that decide which GPU application kernels are suitable for PIM execution.

Data Mapping. Determining the ideal memory mapping for data used by PIM processing logic is another important research direction. To maximize the benefits of PIM, data that needs to be read from or written to by a single PIM kernel instance should be mapped to the same memory stack. Hence, it is important to examine both static and adaptive data mapping mechanisms to intelligently map (or remap) data. Even with such data mapping mechanisms, it is beneficial to provide low-cost and low-overhead data migration mechanisms to facilitate easier PIM execution, in case the data mapping needs to be adapted to execution and access patterns at runtime. One of our recent works provides a mechanism that provides programmer-transparent data mapping support for

PIM [68]. Future work can focus on developing new data mapping mechanisms, as well as designing systems that can take advantage of these new data mapping mechanisms.

PIM Runtime Scheduling Support. At least four key runtime issues in PIM are to decide (1) when to enable PIM execution, (2) what to execute near data, (3) how to map data to multiple (hybrid) 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. It is possible to build on our recent works that employ locality prediction [3] and combined compiler and dynamic code identification and scheduling in GPU-based systems [68, 172]. Several key research questions that should be investigated include:

- 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 the application code should be executed on PIM? What are simple mechanisms to identify such code?
- What are scheduling mechanisms to share PIM accelerators between multiple requesting cores to maximize PIM's benefits?

Granularity of PIM Scheduling. To enable the widespread adoption of PIM, we must understand the ideal granularity at which PIM operations can be scheduled without sacrificing PIM execution's efficiency and limiting changes to the shared memory programming model. Two key issues for scheduling code for PIM execution are (1) how large each part of the code should be (i.e., the granularity of PIM execution), and (2) the frequency at which code executing on a PIM engine should synchronize with code executing on the CPU cores (i.e., the granularity of PIM synchronization).

The optimal granularity of PIM execution remains an open question. For example, is it best to offload only a single instruction to the PIM processing logic? Should PIM kernels consist of a set of instructions, and if so, how large is each set? Do we limit PIM execution to work only on entire functions, entire threads, or even entire applications? If we offload too short a piece of code, the benefits of executing the code near memory may be unable to overcome the overhead of invoking PIM execution (e.g., communicating registers or data, taking checkpoints).

Once code begins to execute on PIM processing logic, there may be times where the code needs to synchronize with code executing on the CPU. For example, many shared memory applications employ locks, barriers, or memory fences to coordinate access to data and ensure correct execution. PIM system architects must determine (1) whether code executing on PIM should allow the support of such synchronization operations; and (2) if they do allow such operations, how to perform them efficiently. Without an efficient mechanism for synchronization, PIM processing logic may need to communicate frequently with the CPU when synchronization takes place, which can introduce overheads and undermine the benefits of PIM execution. Research on PIM synchronization can build upon our prior work, where we limit PIM execution to atomic instructions to avoid the need for synchronization [3], or where provide support within LazyPIM to perform synchronization during PIM kernel execution [21].

PIM Evaluation Infrastructures and Benchmark Suites. To ease adoption, it is critical that we accurately assess the benefits of PIM. Accurate assessment for PIM requires (1) a set of real-world memory-intensive applications that have the potential to benefit significantly when executed near memory, and (2) a simulation/evaluation infrastructure that allows architects and system designers to precisely analyze the benefits and overhead of adding PIM processing logic to memory and executing code on this processing logic.

In order to identify what processing logic should be introduced near memory, and to know what properties are ideal for PIM kernels, we must begin by developing a real-world benchmark suite 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. 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, 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 PIM mechanisms. As an example, in our prior work, we have started analyzing mechanisms for accelerating graph processing [2, 3]; pointer chasing [62, 67]; databases [21, 22, 67, 201]; consumer workloads [20], including web browsing, video encoding/decoding, and machine learning; and GPGPU workloads [68, 172].

Once we have established a set of applications to explore, it is essential for researchers to develop an extensive and flexible application profiling and simulation infrastructure and mechanisms that can (1) identify parts of these applications for which PIM execution can be beneficial; and (2) simulate in-memory acceleration. A systematic process for identifying potential PIM kernels within an application can not only ease the burden for performing PIM research, but could also inspire tools that programmers and compilers can use to automate the process of offloading portions of existing applications to PIM processing logic. Once we have identified potential PIM kernels, we need a simulator to accurately model the energy and performance consumed by PIM hardware structures, available memory bandwidth, and communication overhead when we execute the kernels near memory. Highly-flexible memory simulators (e.g., Ramulator [98, 189], SoftMC [65, 191]) can be combined with full-system simulation infrastructures (e.g., gem5 [18]) to provide a robust environment that can evaluate how various PIM architectures affect the *entire compute stack*, and can allow designers to identify memory characteristics (e.g., internal bandwidth, trade-off between number of PIM engines and memory capacity) that affect the efficiency of PIM execution.

Applicability to Emerging Memory Technologies. As DRAM scalability issues are becoming more difficult to work around [2, 3, 27, 29, 37, 64, 65, 68, 79, 81, 95–97, 109, 113, 116–118, 125, 130, 137, 138, 143, 151, 152, 160, 226, 233, 239, 240], there has been a growing amount of work on emerging non-volatile memory technologies to replace DRAM. Examples of these emerging memory technologies include *phase-change memory* (PCM) [107–109, 178, 229, 240, 245], *spin-transfer torque magnetic RAM* (STT-MRAM) [101, 162], *metal-oxide resistive RAM* (RRAM) [228], and *memristors* [31, 220]. These memories have the potential to offer much greater memory capacity and high internal memory bandwidth. Processing-in-memory techniques can take advantage of this potential, by exploiting the high available internal memory bandwidth, and by making use of the underlying memory device behavior, to perform computation.

PIM can be especially useful in single-level store settings [14, 88, 146, 181, 206, 207, 244], where multiple memory and storage technologies (including emerging memory technologies) are presented to the system as a single monolithic memory, which can be accessed quickly and at high volume by applications. By performing some of the computation in memory, PIM can take advantage of the high bandwidth and capacity available within a single-level store without being bottlenecked by the limited off-chip bandwidth between the various memory and system software components of the single-level store and the CPU.

Given the worsening DRAM scaling issues, and the limited bandwidth available between memory and the CPU, we believe that there is a growing need to investigate PIM processing logic that is designed for emerging memory technologies. We believe that many PIM techniques can be applicable in these technologies. Already, several prior works propose to exploit memory device behavior to perform processing using memory, where the memory consists of PCM [124] or memristors [103–105, 121, 205]. Future research should explore how PIM can take advantage of emerging memory technologies in other ways, and how PIM can work effectively in single-level stores.

7 CONCLUSION

Circuit and device technology scaling for main memory, built predominantly with DRAM, is already showing signs of coming to an end, with three major issues emerging. First, the reliability and data retention capability of DRAM have been decreasing, as shown by various error characterization and analysis studies [25, 65, 81, 84–87, 91, 114, 129, 130, 141, 147, 150, 152, 168, 177, 194, 214], and new failure mechanisms have been slipping into devices in the field (e.g., Rowhammer [94–96, 152]). Second, main memory performance improvements have not grown as rapidly as logic performance improvements have for several years now, resulting in significant performance bottlenecks [2, 26, 27, 113, 114, 116–118, 125, 143, 151, 160, 197, 233]. Third, the increasing application demand for memory places even greater pressure on the main memory system in terms of both performance and energy efficiency [2, 3, 20, 27, 29, 37, 68, 79, 81, 95–97, 109, 113, 116–118, 125, 130, 137, 138, 143, 151, 152, 160, 226, 233, 239, 240]. To solve these issues, there is an increasing need for architectural and system-level approaches [151, 152, 160].

A major hindrance to memory performance and energy efficiency is the high cost of moving data between the CPU and memory. Currently, this cost must be paid *every time* an application needs to perform an operation on data that is stored within memory. The recent advent of 3D-stacked memory architectures, which contain a layer dedicated for logic within the same stack as memory layers, open new possibilities to reduce unnecessary data movement by allowing architects to shift some computation into memory. Processing-in-memory (PIM), or near-data processing, allows architects to introduce simple PIM processing logic (which can be specialized acceleration logic, general-purpose cores, or reconfigurable logic) into the logic layer of the memory, where the PIM processing logic has access to the high internal bandwidth and low memory access latency that exist within 3D-stacked memory. As a result, PIM architectures can reduce costly data movement over the memory channel, lower memory access latency, and thereby also reduce energy consumption.

A number of challenges exist in enabling PIM at the system level, such that PIM can be adopted easily in many system designs. In this work, we examine two such key design issues, which we believe require efficient and elegant solutions to enable widespread adoption of PIM in real systems. First, because applications store memory references as virtual addresses, PIM processing logic needs to perform *address translation* to determine the physical addresses of these references during execution. However, PIM processing logic does not have an efficient way of accessing to the translation lookaside buffer or the page table walkers that reside in the CPU. Second, because PIM processing logic can often access the same data structures that are being accessed and modified by the CPU, a system that incorporates PIM cores needs to support cache coherence between the CPU and PIM cores to ensure that all of the cores are using the correct version of the data. Naive solutions to overcome the address translation and cache coherence challenges either place significant restrictions on the types of computation that can be performed by PIM processing logic, which can break the existing multithreaded programming model and prevent the widespread adoption of PIM, or force PIM processing logic to communicate with the CPU frequently, which can undo the benefits of moving computation to memory. Using key observations about the behavior of address translation and cache coherence for several memory-intensive applications, we propose two solutions that (1) provide general purpose support for translation and coherence in PIM architectures, (2) maintain the conventional multithreaded programming model, and (3) do not incur high communication overheads. The first solution, IMPICA, provides an efficient in-memory accelerator for pointer chasing that can perform efficient address translation from within memory. The second solution, LazyPIM, provides an efficient cache coherence protocol that does not restrict how PIM processing logic and the CPU share data, by using speculation and coherence message compression to minimize the overhead of PIM coherence requests.

We hope that our solutions to the address translation and cache coherence challenges can ease the adoption of PIM-based architectures, by easing both the design and programmability of such systems. We also hope that the

challenges and ideas discussed in this chapter can inspire other researchers to develop other novel solutions that can ease the adoption of PIM architectures.

ACKNOWLEDGMENTS

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 chapter. Thanks also goes to our research group's industrial sponsors over the past nine years, especially 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, and NSF grants 1212962, 1320531, and 1409723.

REFERENCES

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *HPCA*, 2017.

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

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

[4] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.

[5] C. Alkan *et al.*, "Personalized Copy Number and Segmental Duplication Maps Using Next-Generation Sequencing," *Nature Genetics*, 2009.

[6] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: A New Hardware Architecture for Accelerating Pre-Alignment in DNA Short Read Mapping," *Bioinformatics*, 2017.

[7] ARM Holdings, "ARM Cortex-A57," http://www.arm.com/products/processors/cortex-a/cortex-a57-processor.php.

[8] ARM Holdings, "ARM Cortex-R4," http://www.arm.com/products/processors/cortex-r/cortex-r4.php.

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

[10] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes," in *MICRO*, 2017.

[11] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency," in *ASPLOS*, 2018.

[12] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.

[13] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *ISCA*, 2013.

[14] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory: Concepts and Design," *CACM*, 1972.

[15] A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," in *MICRO*, 2013.

[16] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," in *HPCA*, 2011.

[17] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB for Chip Multiprocessors," in *ASPLOS*, 2010.

[18] N. Binkert, B. Beckman, A. Saidi, G. Black, and A. Basu, "The gem5 Simulator," *CAN*, 2011.

[19] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Commun. ACM*, 1970.

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

[21] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, N. Hajinazar, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: Efficient Support for Cache Coherence in Processing-in-Memory Architectures," arXiv:1706.03162 [cs:AR], 2017.

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

[23] L. M. Censier and P. Feutrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE TC*, 1978.

[24] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, 2007.

[25] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.

[26] K. K. Chang, "Understanding and Improving the Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon University, 2017.

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

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

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

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

[31] L. Chua, "Memristor—The Missing Circuit Element," *IEEE TCT*, 1971.

[32] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big Data on Little Clients," in *ISCA*, 2013.

[33] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer Cache Assisted Prefetching," in *MICRO*, 2002.

[34] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. Lee, D. M. Lavery, and J. P. Shen, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," in *ISCA*, 2001.

[35] R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-Directed Data Prefetching Mechanism," in *ASPLOS*, 2002.

[36] N. C. Crago and S. J. Patel, "OUTRIDER: Efficient Memory Latency Tolerance with Decoupled Strands," in *ISCA*, 2011.
[37] J. Dean and L. A. Barroso, "The Tail at Scale," *CACM*, 2013.
[38] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic Shared Memory Multiprocessing," in *ASPLOS*, 2009.
[39] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The Architecture of the DIVA Processing-in-Memory Chip," in *SC*, 2002.
[40] E. Ebrahimi, O. Mutlu, and Y. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *HPCA*, 2009.
[41] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-Aware Shared Resource Management for Multi-core Systems," in *ISCA*, 2011.
[42] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-Core Systems," in *MICRO*, 2009.
[43] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal Feasibility of Die-Stacked Processing in Memory," in *WoNDP*, 2014.
[44] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational RAM: Implementing Processors in Memory," *IEEE Design & Test*, 1999.
[45] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational RAM: A Memory-SIMD Hybrid and Its Application to DSP," in *CICC*, 1992.
[46] R. Elmasri, *Fundamentals of Database Systems*. Pearson, 2007.
[47] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.
[48] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," in *ASPLOS*, 2012.
[49] M. Filippo, "Technology Preview: ARM Next Generation Processing," *ARM TechCon*, 2012.
[50] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, 2004.
[51] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[52] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.
[53] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP*, 2003.
[54] D. Giampaolo, *Practical File System Design with the BE File System*. Morgan Kaufmann Publishers Inc., 1998.
[55] A. Glew, "MLP Yes! ILP No!" in *ASPLOS WACI*, 1998.
[56] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *IEEE Computer*, 1995.
[57] J. R. Goodman, "Using Cache Memory to Reduce Processor-memory Traffic," in *ISCA*, 1983.
[58] 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, and D. Chang, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *ISCA*, 2016.
[59] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
[60] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of Error in Full-System Simulation," in *ISPASS*, 2014.
[61] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *ISCA*, 2004.
[62] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *MICRO*, 2016.
[63] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.
[64] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
[65] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
[66] S. M. Hassan, S. Yalamanchili, and S. Mukhopadhyay, "Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore," in *MEMSYS*, 2015.
[67] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[68] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Conner, N. Vijaykumar, O. Mutlu, and S. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[69] Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: Tag Correlating Prefetchers," in *HPCA*, 2003.
[70] C. J. Hughes and S. V. Adve, "Memory-Side Prefetching for Linked Data Structures for Processor-in-Memory Systems," *JPDC*, 2005.
[71] Hybrid Memory Cube Consortium, "HMC Specification 1.1," 2013.
[72] Hybrid Memory Cube Consortium, "HMC Specification 2.0," 2014.
[73] Intel, "Intel Xeon Processor W3550," 2009.

[74] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube: New DRAM Architecture Increases Density and Performance," in *VLSIT*, 2012.
[75] JEDEC, "High Bandwidth Memory (HBM) DRAM," Standard No. JESD235, 2013.
[76] J. Joao, O. Mutlu, and Y. N. Patt, "Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection," in *ISCA*, 2009.
[77] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., 1996.
[78] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," in *ISCA*, 1997.
[79] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-Scale Computer," in *ISCA*, 2015.
[80] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM," in *ICASSP*, 2014.
[81] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. Choi, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
[82] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 1999.
[83] M. Karlsson, F. Dahlgren, and P. Stenström, "A Prefetching Technique for Irregular Accesses to Linked Data Structures," in *HPCA*, 2000.
[84] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[85] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data Dependent Failures in DRAM," in *DSN*, 2016.
[86] S. Khan, C. Wilkerson, D. Lee, A. R. Alameldeen, and O. Mutlu, "A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM," *CAL*, 2016.
[87] S. Khan, C. Wilkerson, Z. Wang, A. Alameldeen, D. Lee, and O. Mutlu, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
[88] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-Level Storage System," *IRE Trans. Elec. Computers*, 1962.
[89] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.
[90] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh, "Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs," in *SC*, 2017.
[91] J. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
[92] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Filtering in Read Mapping Using Emerging Memory Technologies," arXiv:1708.04329 [q-bio.GN], 2017.
[93] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
[94] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "RowHammer: Reliability Analysis and Security Implications," arXiv:1603.00747 [cs:AR], 2016.
[95] Y. Kim, "Architectural Techniques to Enhance DRAM Scaling," Ph.D. dissertation, Carnegie Mellon University, 2015.
[96] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[97] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[98] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
[99] Y. O. Koçberber, B. Grot, J. Picorel, B. Falsafi, K. T. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases," in *MICRO*, 2013.
[100] P. M. Kogge, "EXECUBE–A New Architecture for Scaleable MPPs," in *ICPP*, 1994.
[101] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *ISPASS*, 2013.
[102] L. Kurian, P. T. Hulina, and L. D. Coraor, "Memory Latency Effects in Decoupled Architectures With a Single Data Memory Module," in *ISCA*, 1992.
[103] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC—Memristor-Aided Logic," *IEEE TCAS II: Express Briefs*, 2014.
[104] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Memristor-Based IMPLY Logic Design Procedure," in *ICCD*, 2011.
[105] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *TVLSI*, 2014.
[106] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE TC*, 1979.
[107] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, 2010.

[108] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, 2010.
[109] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
[110] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware Memory Controllers," *IEEE TC*, 2011.
[111] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.
[112] D. Lee, F. Hormozdiari, H. Xin, F. Hach, O. Mutlu, and C. Alkan, "Fast and Accurate Mapping of Complete Genomics Reads," *Methods*, 2014.
[113] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[114] D. Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," Ph.D. dissertation, Carnegie Mellon University, 2016.
[115] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
[116] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
[117] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[118] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[119] J. Lee, Y. Solihin, and J. Torrettas, "Automatically Mapping Code on an Intelligent Memory Architecture," in *HPCA*, 2001.
[120] J. H. Lee, J. Sim, and H. Kim, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *PACT*, 2015.
[121] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky, "Logic Operations in Memory Using a Memristive Akers Array," *Microelectronics Journal*, 2014.
[122] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.
[123] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing," *TACO*, 2013.
[124] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.
[125] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated Memory for Expansion and Sharing in Blade Servers," in *ISCA*, 2009.
[126] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *ISCA*, 2013.
[127] Linaro, "64-Bit Linux Kernel for ARM," 2014.
[128] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, "SPAID: Software Prefetching in Pointer- and Call-Intensive Environments," in *MICRO*, 1995.
[129] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[130] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[131] Z. Liu, I. Calciu, M. Harlihy, and O. Mutlu, "Concurrent Data Structures for Near-Memory Computing," in *SPAA*, 2017.
[132] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *ISCA*, 2008.
[133] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A Processing in Memory Taxonomy and a Case for Studying Fixed-Function PIM," in *WoNDP*, 2013.
[134] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, Y. O. Koçberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Özer, and B. Falsafi, "Scale-Out Processors," in *ISCA*, 2012.
[135] C. Luk, "Tolerating Memory Latency Through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," in *ISCA*, 2001.
[136] C. Luk and T. C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," in *ASPLOS*, 1996.
[137] Y. Luo, S. Ghose, T. Li, S. Govindan, B. Sharma, B. Kelly, A. Boroumand, and O. Mutlu, "Using ECC DRAM to Adaptively Increase Memory Capacity," arXiv:1706.08870 [cs:AR], 2017.
[138] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
[139] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," *ACM TACO*, 2013.

[140] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA*, 2000.
[141] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, "Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM)," *IBM JRD*, 2002.
[142] Y. Mao, E. Kohler, and R. T. Morris, "Cache Craftiness for Fast Multicore Key-Value Storage," in *EuroSys*, 2012.
[143] S. A. McKee, "Reflections on the Memory Wall," in *CF*, 2004.
[144] MemSQL, Inc., "MemSQL," http://www.memsql.com.
[145] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for mixing Die-Stacked and Off-Package Memories," in *HPCA*, 2015, pp. 126–136.
[146] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
[147] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
[148] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot, "Sort vs. Hash Join Revisited for Near-Memory Execution," in *ASBD*, 2007.
[149] A. Morad, L. Yavits, and R. Ginosar, "GP-SIMD Processing-in-Memory," *ACM TACO*, 2015.
[150] J. Mukundan, H. Hunter, K. H. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems," in *ISCA*, 2013.
[151] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *IMW*, 2013.
[152] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
[153] O. Mutlu, H. Kim, and Y. N. Patt, "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," in *MICRO*, 2005.
[154] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.
[155] O. Mutlu, H. Kim, and Y. N. Patt, "Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses," *TC*, 2006.
[156] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," *IEEE Micro*, 2006.
[157] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[158] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.
[159] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Effective Alternative to Large Instruction Windows," *IEEE Micro*, 2003.
[160] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
[161] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, "SigRace: Signature-Based Data Race Detection," in *ISCA*, 2009.
[162] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "STT-RAM Scaling and Retention Failure," *Intel Technol. J.*, 2013.
[163] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[164] B. Naylor, J. Amanatides, and W. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations," in *SIGGRAPH*, 1990.
[165] OProfile, http://oprofile.sourceforge.net/.
[166] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.
[167] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private. Cache Memories," in *ISCA*, 1984.
[168] M. Patel, J. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
[169] Y. N. Patt, W.-M. Hwu, and M. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," in *MICRO*, 1985.
[170] Y. N. Patt, S. W. Melvin, W.-M. Hwu, and M. C. Shebanow, "Critical Issues Regarding HPS, a High Performance Microarchitecture," in *MICRO*, 1985.
[171] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
[172] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.
[173] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ASPLOS*, 2014.
[174] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai, "Architecting a Chunk-Based Memory Race Recorder in Modern CMPs," in *MICRO*, 2009.
[175] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *HPCA*, 2014.

[176] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
[177] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[178] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.
[179] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines," in *HPCA*, 2007.
[180] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer, "Adaptive Insertion Policies for High-Performance Caching," in *ISCA*, 2007.
[181] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems," in *MICRO*, 2015.
[182] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
[183] O. Rodeh, C. Mason, and J. Bacik, "BTRFS: The Linux B-tree Filesystem," *TOS*, 2013.
[184] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting Dynamic Data Structures on Distributed-Memory Machines," *TOPLAS*, 1995.
[185] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *CAL*, 2011.
[186] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence Based Prefetching for Linked Data Structures," in *ASPLOS*, 1998.
[187] A. Roth and G. S. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures," in *ISCA*, 1999.
[188] SAFARI Research Group, "IMPICA (In-Memory PoInter Chasing Accelerator) – GitHub Repository," https://github.com/CMU-SAFARI/IMPICA/.
[189] SAFARI Research Group, "Ramulator: A DRAM Simulator – GitHub Repository," https://github.com/CMU-SAFARI/ramulator/.
[190] SAFARI Research Group, "SAFARI Software Tools – GitHub Repository," https://github.com/CMU-SAFARI/.
[191] SAFARI Research Group, "SoftMC v1.0 – GitHub Repository," https://github.com/CMU-SAFARI/SoftMC/.
[192] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing Signatures for Transactional Memory," in *MICRO*, 2007.
[193] SAP SE, "SAP HANA," http://www.hana.sap.com/.
[194] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," in *SIGMETRICS*, 2009.
[195] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," arXiv:1611.09988 [cs:AR], 2016.
[196] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[197] V. Seshadri, "Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems," Ph.D. dissertation, Carnegie Mellon University, 2016.
[198] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ISCA*, 2014.
[199] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.
[200] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[201] V. Seshadri, T. Mullins, A. Boroumand, O. Mutli, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
[202] V. Seshadri and O. Mutlu, "The Processing Using Memory Paradigm: In-DRAM Bulk Copy, Initialization, Bitwise AND and OR," arXiv:1610.09603 [cs:AR], 2016.
[203] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers, Volume 106*, 2017.
[204] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," *ACM TACO*, vol. 11, no. 4, pp. 51:1–51:22, 2015.
[205] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.
[206] J. S. Shapiro and J. Adams, "Design Evolution of the EROS Single-Level Store," in *USENIX ATC*, 2002.
[207] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A Fast Capability System," in *SOSP*, 1999.
[208] D. E. Shaw, S. J. Stolfo, H. Ibrahim, B. Hillyer, G. Wiederhold, and J. Andrews, "The NON-VON Database Machine: A Brief Overview," *IEEE Database Eng. Bull.*, 1981.
[209] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *PPoPP*, 2013.
[210] J. E. Smith, "Decoupled Access/Execute Computer Architectures," in *ISCA*, 1982.
[211] J. E. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," *Computer*, 1986.
[212] J. E. Smith, S. Weiss, and N. Y. Pang, "A Simulation Study of Decoupled Architecture Computers," *IEEE TC*, 1986.
[213] Y. Solihin, J. Torrellas, and J. Lee, "Using a User-Level Memory Thread for Correlation Prefetching," in *ISCA*, 2002.

[214] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, the Bad, and the Ugly," in *ASPLOS*, 2015.
[215] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors," in *MICRO*, 2010.
[216] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007.
[217] Stanford Network Analysis Project, http://snap.stanford.edu/.
[218] H. S. Stone, "A Logic-in-Memory Computer," *TC*, 1970.
[219] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS." *IEEE Data Eng. Bull.*, 2013.
[220] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The Missing Memristor Found," *Nature*, 2008.
[221] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair, "Data Access Optimization in a Processing-in-Memory System," in *CF*, 2015.
[222] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM JRD*, 1967.
[223] Transaction Processing Performance Council, "TPC Benchmarks," http://www.tpc.org.
[224] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *SIGCOMM*, 1997.
[225] 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, and B. Qiu, "BigDataBench: A Big Data Benchmark Suite From Internet Services," in *HPCA*, 2014.
[226] M. V. Wilkes, "The Memory Gap and the Future of High Performance Memories," *CAN*, 2001.
[227] P. R. Wilson, "Uniprocessor Garbage Collection Techniques," in *IWMM*, 1992.
[228] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal-Oxide RRAM," *Proc. IEEE*, 2012.
[229] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proc. IEEE*, 2010.
[230] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," in *ISCA*, 2013.
[231] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The Architecture and Design of a Database Processing Unit," in *ASPLOS*, 2014.
[232] Y. Wu, "Efficient Discovery of Regular Stride Patterns in Irregular Programs," in *PLDI*, 2002.
[233] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *CAN*, 1995.
[234] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the Wall: Near-Data Processing for Databases," in *DaMoN*, 2015.
[235] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted Hamming Distance: A Fast and Accurate SIMD-Friendly Filter to Accelerate Alignment Verification in Read Mapping," *Bioinformatics*, 2015.
[236] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating Read Mapping with FastHASH," *BMC Genomics*, 2013.
[237] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing," in *HPDC*, 2014.
[238] C. Yang and A. R. Lebeck, "Push vs. Pull: Data Movement for Linked Data Structures," in *ICS*, 2000.
[239] H. Yoon, R. A. J. Meza, R. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
[240] H. Yoon, J. Meza, N. Muralimanohar, N. P. Jouppi, and O. Mutlu, "Efficient Data Mapping and Buffering Techniques for Multilevel Cell Phase-Change Memories," *ACM TACO*, 2014.
[241] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores," *VLDB*, 2014.
[242] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect Memory Prefetcher," in *MICRO*, 2015.
[243] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.
[244] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.
[245] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *ISCA*, 2009.
[246] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.
[247] C. B. Zilles, "Benchmark Health Considered Harmful," *CAN*, 2001.
[248] C. B. Zilles and G. S. Sohi, "Execution-Based Prediction Using Speculative Slices," in *ISCA*, 2001.
[249] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," US Patent No. 5,630,096, 1997.