

PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture

Junwhan Ahn Sungjoo Yoo Onur Mutlu[†] Kiyoung Choi
junwhan@snu.ac.kr, sungjoo.yoo@gmail.com, onur@cmu.edu, kchoi@snu.ac.kr

Seoul National University

[†]Carnegie Mellon University

Abstract

Processing-in-memory (PIM) is rapidly rising as a viable solution for the memory wall crisis, rebounding from its unsuccessful attempts in 1990s due to practicality concerns, which are alleviated with recent advances in 3D stacking technologies. However, it is still challenging to integrate the PIM architectures with existing systems in a seamless manner due to two common characteristics: unconventional programming models for in-memory computation units and lack of ability to utilize large on-chip caches.

In this paper, we propose a new PIM architecture that (1) does not change the existing sequential programming models and (2) automatically decides whether to execute PIM operations in memory or processors depending on the locality of data. The key idea is to implement simple in-memory computation using compute-capable memory commands and use specialized instructions, which we call PIM-enabled instructions, to invoke in-memory computation. This allows PIM operations to be interoperable with existing programming models, cache coherence protocols, and virtual memory mechanisms with no modification. In addition, we introduce a simple hardware structure that monitors the locality of data accessed by a PIM-enabled instruction at runtime to adaptively execute the instruction at the host processor (instead of in memory) when the instruction can benefit from large on-chip caches. Consequently, our architecture provides the illusion that PIM operations are executed as if they were host processor instructions.

We provide a case study of how ten emerging data-intensive workloads can benefit from our new PIM abstraction and its hardware implementation. Evaluations show that our architecture significantly improves system performance and, more importantly, combines the best parts of conventional and PIM architectures by adapting to data locality of applications.

1. Introduction

Performance and energy consumption of modern computer systems are largely dominated by their memory hierarchy. This memory bottleneck is expected to be aggravated due to two trends. First, computational power has been continuously increasing through architectural innovations (e.g., chip multiprocessors, specialized accelerators, etc.), whereas the memory bandwidth cannot be easily increased due to the pin count limitation. Second, emerging data-intensive workloads require large volumes of data to be transferred fast enough to keep computation units busy, thereby putting even higher pressure on the memory hierarchy.

The widening discrepancy between computation speed and data transfer speed, commonly known as the *memory wall* [49], motivates the need for a different computing paradigm. In particular, processing-in-memory (PIM) is regaining attention because it can minimize data movement by placing the computation close to where data resides. Although the PIM concept itself was already studied by many researchers decades ago, it is worth revisiting it in a new context today: (1) 3D stacking technology now enables cost-effective integration of logic and memory and (2) many new applications are data-intensive and demand great amounts of memory bandwidth [1, 2, 32].

To date, most of the existing PIM architectures are based on general-purpose computation units inside memory for flexibility across different applications [14–16, 25, 28, 35, 39–41, 46, 48, 50]. However, this introduces two major challenges in seamlessly integrating such architectures into conventional systems in the near term. First, prior proposals require new programming models for in-memory computation units, which is often significantly different from what is used today. Main memory products with integrated full-fledged processors with new programming models may also not be available in the near future because of the associated design complexity and changes required across the hardware/software stack.

Second, prior proposals do not utilize the benefits of on-chip caches and virtual memory provided by host processors. Specifically, they offload computation to memory with no consideration of on-chip cache locality, thereby significantly degrading performance when the applications exhibit high data locality. Moreover, most prior approaches perform in-memory computation on noncacheable, physically addressed memory regions, which inevitably sacrifices efficiency and safety of all memory accesses from host processors to memory regions that can potentially be accessed by PIM. In particular, the lack of interoperability with on-chip caches is critical considering that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ISCA '15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. ISBN 978-1-4503-3402-0/15/06\$15.00
DOI: <http://dx.doi.org/10.1145/2749469.2750385>

commercial processors already integrate large last-level caches on chip (e.g., Intel Ivytown has a 37.5 MB L3 cache [42]).

To overcome these two major limitations, this paper proposes to enable simple PIM operations by extending the ISA of the host processor with *PIM-enabled instructions (PEIs)*, without changing the existing programming model. We define a PEI as an instruction that can be executed either on the host processor or on the in-memory computation logic. By providing PIM capability as part of the existing programming model, conventional architectures can exploit the PIM concept with no changes in their programming interface. The approach of using PEIs also facilitates our architecture to support cache coherence and virtual memory seamlessly: existing mechanisms in place for cache coherence and virtual memory can be used without modification, unlike other approaches to PIM. We develop a hardware-based scheme that can adaptively determine the location to execute PEIs by considering data locality: a PEI can be selectively executed on the host processor (instead of on the in-memory computation logic) when large on-chip caches are beneficial for its execution.

This paper makes the following major contributions:

- We introduce the concept of PIM-enabled instructions (PEIs), which enable simple PIM operations to be interfaced as simple ISA extensions. We show that the judicious usage of PEIs can achieve significant speedup with minimal programming effort and no changes to the existing programming model.
- We design an architecture that supports implementing PEIs as part of the host-PIM interface in order to provide the illusion that PIM operations are executed as if they were host processor instructions. Unlike previous PIM proposals, PEIs are fully interoperable with existing cache coherence and virtual memory mechanisms.
- We propose a mechanism to dynamically track locality of data accessed by PEIs and execute PEIs with high data locality on host processors instead of offloading them to memory, in order to exploit large on-chip caches.
- We evaluate our architecture using ten emerging data-intensive workloads and show significant performance improvements over conventional systems. We also show that our architecture is able to adapt to data locality at runtime, thereby outperforming PIM-only systems as well.

2. Background and Motivation

2.1. 3D-Stacked DRAM and Processing-in-Memory

Recent advances in die stacking technologies facilitate low-cost integration of logic and memory dies in a single package. Considering the Hybrid Memory Cube (HMC) [23] as a concrete example: each cube consists of multiple DRAM dies and one logic die connected via high-bandwidth links called through-silicon vias (TSVs). Such a 3D-stacked organization of logic and memory brings multiple opportunities in optimizing main memory subsystems. First, main memory density can be improved over conventional 2D designs by stacking multiple DRAM dies on a single chip. Second, the base logic die

can be utilized to integrate memory controllers and high-speed signaling circuits inside memory. This enables the realization of an abstract, packetized memory interface (instead of low-level DRAM commands) and higher off-chip memory bandwidth [20]. Third, TSV-based vertical interconnect facilitates low-latency, high-bandwidth, and energy-efficient data transfer between different dies in a package [8].

Among these three major benefits of 3D-stacked DRAM, this paper concentrates on exploiting especially the last two. When 3D-stacked DRAM is used as off-chip main memory, it provides much higher memory bandwidth inside a DRAM chip (between the logic die and the DRAM die) than between processors and DRAM chips because TSVs are much more cost-effective than off-chip wires and package pins. Moreover, vertical data transfer inside a 3D-stacked DRAM chip between logic and memory is more energy-efficient than off-chip transfer due to the much shorter wire length.

Our approach to leveraging such advantages is to move computation inside its logic die, which is called processing-in-memory (PIM). This reduces the latency and energy overhead caused by off-chip data transfer and, more importantly, enables utilizing high internal memory bandwidth. Moreover, unlike PIM architectures in 1990s that required significant modification to DRAM dies (e.g., [14, 16, 25, 28, 39, 40]), such integration can now be realized in a much more cost-effective manner due to the existence of logic dies in 3D-stacked DRAM. These benefits can potentially improve system performance and energy efficiency in a practical manner, but only with careful design of PIM architectures.

Past work on PIM, including early proposals [16, 25, 35, 39, 40, 46] and more recent ones [1, 15, 41, 50], mostly relies on fully programmable computation units (e.g., general-purpose processors, programmable logic, etc.) in memory. While programmability gives the benefit of broad applicability across different workloads, such approaches may not be suitable for a near-term solution due to (1) high design effort required to integrate such complex modules into memory and (2) new programming models for in-memory computation units, which require significant changes to software code to exploit PIM.

Motivated by these difficulties, we explore possibilities of *integrating simple PIM operations by minimally extending the ISA of host processors*. Compared to PIM architectures based on fully programmable computation units, our approach improves the practicality of the PIM concept by reducing the implementation overhead of in-memory computation units and facilitating the use of existing programming models.¹ However, despite these advantages, no prior work, to our knowledge, explored the methods and their effectiveness of utilizing *simple* in-memory computation.

2.2. Potential of ISA Extensions as the PIM Interface

To evaluate the potential of introducing simple PIM operations as ISA extensions, let us consider a parallel implementation

¹Due to these advantages, simple in-memory computation mechanisms are already starting to become available from the industry, e.g., the in-memory 8/16-byte arithmetic/bitwise/boolean/comparison atomics in HMC 2.0 [21].

```

1  parallel_for (v: graph.vertices) {
2    v.pagerank = 1.0 / graph.num_vertices;
3    v.next_pagerank = 0.15 / graph.num_vertices;
4  }
5  count = 0;
6  do {
7    parallel_for (v: graph.vertices) {
8      delta = 0.85 * v.pagerank / v.out_degree;
9      for (w: v.successors) {
10       atomic w.next_pagerank += delta;
11     }
12   }
13   diff = 0.0;
14   parallel_for (v: graph.vertices) {
15     atomic diff += abs(v.next_pagerank - v.pagerank);
16     v.pagerank = v.next_pagerank;
17     v.next_pagerank = 0.15 / graph.num_vertices;
18   }
19 } while (++count < max_iteration && diff > e);

```

Figure 1: Pseudocode of parallel PageRank computation.

of the PageRank algorithm [17], shown in Figure 1, as an example. PageRank [5] is widely used in web search engines, spam detection, and citation ranking to compute the importance of nodes based on their relationships. For large graphs, the performance bottleneck of this workload is in updating `next_pagerank` of successor vertices (line 10) since it generates a very large amount of random memory accesses across the entire graph [34]. Due to its high memory intensity and computational simplicity, this operation in PageRank is a good candidate for implementation as a simple PIM operation.

Figure 2 shows speedup achieved by implementing an *atomic addition* command inside memory (see Section 6 for our evaluation methodology). The evaluation is based on nine real-world graphs [29, 45], which have 62 K to 5 M vertices (y-axis is sorted in ascending order of the number of vertices).

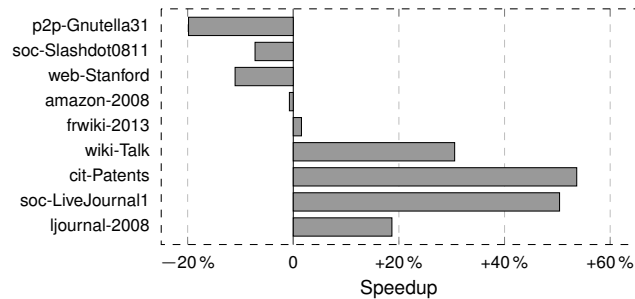


Figure 2: Performance improvement with an in-memory atomic addition operation used for the PageRank algorithm.

We observe that employing only one simple PIM operation can improve system performance by up to 53%. The main benefit is due to the greatly reduced memory bandwidth consumption of the memory-side addition compared to its host-side counterpart. While host-side addition moves the entire cache block back and forth between memory and the host processor for each update, memory-side addition simply sends the 8-byte `delta` to memory to update `w.next_pagerank`. Assuming 64-byte cache blocks and no cache hits, the host-side addition transfers 128 bytes of data from/to off-chip memory for every single update, whereas the memory-side addition

requires only 8 bytes of off-chip communication per update.

Unfortunately, memory-side addition sometimes incurs significant performance degradation as well (up to 20%). This happens when most of `next_pagerank` updates can be served by on-chip caches, in which case host-side addition does *not* consume off-chip memory bandwidth at all. In such a situation, memory-side addition degrades both performance and energy efficiency since it *always* accesses DRAM to update data (e.g., memory-side addition causes 50x DRAM accesses over host-side addition in p2p-Gnutella31). Thus, one needs to be careful in using memory-side operations as their benefit greatly depends on the locality of data in on-chip caches.

In summary, simple PIM operations interfaced as ISA extensions have great potential for accelerating memory-intensive workloads. However, in order to maximize the effectiveness of simple PIM operations, a host processor should be smart enough in utilizing these operations (e.g., importantly, by considering data locality of applications). Based on these observations, in the following sections, we describe our new, simple PIM interface consisting of simple ISA extensions and the architectural support required to integrate such simple PIM operations into conventional systems.

3. PIM Abstraction: PIM-Enabled Instructions

In this section, we explain our abstraction model for simple PIM operations, called PIM-enabled instructions (PEIs). Our goal is to *provide the illusion that PIM operations are executed as if they were host processor instructions*. This section describes our design choices to achieve this new PIM abstraction. Section 4 will describe our detailed implementation to realize this abstraction.

3.1. Operations

In order to integrate PIM capability into an existing ISA abstraction, PIM operations are expressed as specialized instructions (PEIs) of host processors. For example, if the main memory supports an in-memory *atomic add* command (PIM operation), we add a PIM-enabled *atomic add* instruction (PEI) to the host processor. This facilitates effortless and gradual integration of PIM operations into existing software through replacement of normal instructions with PEIs.

When a PEI is issued by a host processor, our hardware mechanism dynamically decides the best location to execute it between memory and the host processor on a per-operation basis. Software does not provide any information to perform such a decision and is unaware of the execution location of the operation, which is determined by hardware.

For memory operands of PIM operations, we introduce an important design choice, called the *single-cache-block restriction*: the memory region accessible by a single PIM operation is limited to a single last-level cache block. Such a restriction brings at least three important benefits in terms of efficiency and practicality of PIM operations, as described below.

- **Localization:** It ensures that data accessed by a PIM operation are always bounded to a single DRAM module. This implies that PIM operations always use only vertical links

(without off-chip data transfer) in transferring the target data between DRAM and in-memory computation units.²

- **Interoperability:** Since PIM operations and normal last-level cache accesses now have the same memory access granularity (i.e., one last-level cache block), hardware support for coherence management and virtual-to-physical address translation for PIM operations becomes greatly simplified (see Section 4).
- **Simplified Locality Profiling:** Locality of data accessed by PIM operations can be easily identified by utilizing the last-level cache tag array or similar structures. Such information is utilized in determining the best location to execute PEIs.

In addition to memory operands, PIM operations can also have input/output operands. An example of this is the `delta` operand of the atomic addition at line 10 of Figure 1. When a PIM operation is executed in main memory, its input/output operands are transferred between host processors and memory through off-chip links. The maximum size of input/output operands is restricted to the size of a last-level cache block because, if input/output operands are larger than a last-level cache block, offloading such PIM operations to memory increases off-chip memory bandwidth consumption compared to host-side execution due to the single-cache-block restriction.

3.2. Memory Model

Coherence. Our architecture supports hardware cache coherence for PIM operations so that (1) PIM operations can access the latest versions of data even if the target data are stored in on-chip caches and (2) normal instructions can see the data modified by PIM operations. This allows programmers to mix normal instructions and PEIs in manipulating the same data without disabling caches, in contrast to many past PIM architectures.

Atomicity. Our memory model ensures atomicity between PEIs. In other words, a PEI that reads from and writes to its target cache block is not interrupted by other PEIs (possibly from other host processors) that access the same cache block. For example, if the addition at line 10 of Figure 1 is implemented as a PEI, hardware preserves its atomicity without any software concurrency control mechanism (e.g., locks).

On the contrary, atomicity of a PEI is not guaranteed if a normal instruction accesses the target cache block. For example, if a normal store instruction writes a value to `w.next_pagerank` (in Figure 1), this *normal* write may happen between reading `w.next_pagerank` and writing it inside the atomic PEI addition (line 10), thereby breaking the atomicity of the PEI. This is because, in order to support atomicity between a PEI and a normal instruction, every memory access needs to be checked, which incurs overhead even for programs that do not use PEIs.

²Without the single-cache-block restriction, PIM operations require special data mapping to prevent off-chip transfer between multiple DRAM modules. This comes with many limitations in that (1) it introduces significant modification to existing systems to expose the physical location of data to software and (2) the resulting design may not be easily adaptable across different main memory organizations.

Instead, host processors provide a PIM memory fence instruction called *pfence* to enforce memory ordering between normal instructions and PEIs. The *pfence* instruction blocks host processor execution until all PEIs issued before it (including those from other host processors) are completed. In Figure 1, a *pfence* instruction needs to be inserted after line 12 since normal instructions in the third loop access data modified by PEIs in the second loop (i.e., the `w.next_pagerank` fields).

It should be noted that, although *pfence* itself might introduce some performance overhead, its overhead can generally be amortized over numerous PEI executions. For example, the PageRank algorithm in Figure 1 issues one PEI per edge before each *pfence*, which corresponds to millions or even billions of PEIs per *pfence* for large real-world graphs.

Virtual Memory. PEIs use virtual addresses just as normal instructions. Supporting virtual memory for PEIs therefore does not require any modification to existing operating systems and applications.

3.3. Software Modification

In this paper, we assume that programmers modify source code of target applications to utilize PEIs. This is similar to instruction set extensions in commercial processors (e.g., Intel SSE/AVX [12]), which are exploited by programmers using intrinsics and are used in many real workloads where they provide performance benefit. However, we believe that the semantics of the PEIs is simple enough (e.g., atomic add) for modern compilers to automatically recognize places to emit them without requiring hints from programmers and/or complex program analysis. Moreover, our scheme reduces the burden of compiler designers since it *automatically* determines the best location to execute each PEI between memory and the host processor, which allows compilers to be less accurate in estimating performance/energy gain of using PEIs.

4. Architecture

In this section, we describe our architecture that implements our PIM abstraction with minimal modification to existing systems. Our architecture is not limited to specific types of in-memory computation, but provides a *substrate* to implement simple yet general-purpose PIM operations in a practical and efficient manner.

4.1. Overview

Figure 3 gives an overview of our architecture. We choose the Hybrid Memory Cube (HMC) [23] as our baseline memory technology. An HMC consists of multiple vertical DRAM partitions called *vaults*. Each vault has its own DRAM controller placed on the logic die. Communication between host processors and HMCs is based on a packet-based abstract protocol supporting not only read/write commands, but also compound commands such as add-immediate operations, bit-masked writes, and so on [20]. Note that our architecture can be easily adopted to other memory technologies since it does not depend on properties specific to any memory technology.

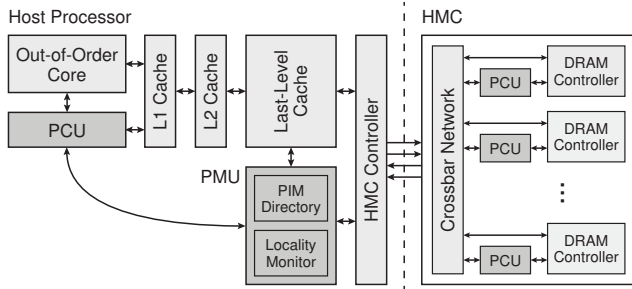


Figure 3: Overview of the proposed architecture.

The key features of our architecture are (1) to support PIM operations as part of the host processor instruction set and (2) to identify PEIs with high data locality and execute them in the host processor. To realize these, our architecture is composed of two main components. First, a PEI Computation Unit (PCU) is added into each host processor and each vault to facilitate PEIs to be executed in either the host processor or main memory. Second, in order to coordinate PEI execution in different PCUs, the PEI Management Unit (PMU) is placed near the last-level cache and is shared by all host processors. In the following subsections, we explain the details of these two components and their operation sequences in host-/memory-side PEI execution.

4.2. PEI Computation Unit (PCU)

Architecture. A PCU is a hardware unit that executes PEIs. Each PCU is composed of computation logic and an operand buffer. The computation logic is a set of circuits for computation supported by main memory (e.g., adders). All PCUs in the system have the same computation logic so that any PEI can be executed by any PCU.

The operand buffer is a small SRAM buffer that stores information of in-flight PEIs. For each PEI, an operand buffer entry is allocated to store its type, target cache block, and input/output operands. When the operand buffer is full, future PEIs are stalled until space frees up in the buffer.

The purpose of the operand buffer is to exploit memory-level parallelism during PEI execution. In our architecture, when a PEI obtains a free operand buffer entry, the PCU immediately sends a read request for the target cache block of the PEI to memory even if the required computation logic is in use. Then, the fetched data are buffered in the operand buffer until the computation logic becomes available. As such, memory accesses from different PEIs can be overlapped by simply increasing the number of operand buffer entries. This is especially useful in our case since simple PIM operations usually underutilize the computation logic of PCUs due to the small amount of computation they generate.

Interface. A host processor controls its host-side PCU by manipulating memory-mapped registers inside it (see Section 4.5). Assemblers can provide pseudo-instructions for PCU control, which are translated to accesses to those memory-mapped registers, in order to abstract the memory-mapped registers away from application software. Although we choose

this less invasive style of integration to avoid modification to out-of-order cores, one can add actual instructions for PCU control by modifying the cores for tighter integration.

Memory-side PCUs are interfaced with the HMC controllers using special memory commands. It is relatively easy to add such commands because communication between HMCs and HMC controllers is based on a packet-based abstract protocol, which allows the flexible addition of new commands.

4.3. PEI Management Unit (PMU)

In order to coordinate all PCUs in the system, the PMU performs three important tasks for PEI execution: (1) atomicity management of PEIs, (2) cache coherence management for PIM operation execution, and (3) data locality profiling for locality-aware execution of PIM operations. We explain in detail the proposed hardware structures to handle these tasks.

PIM Directory. As explained in Section 3.2, our architecture guarantees the atomicity of PEIs. If we consider memory-side PEI execution only, atomicity of PEIs can be maintained simply by modifying each DRAM controller inside HMCs to schedule memory accesses (including reads and writes) from a single PEI as an inseparable group. However, since our architecture executes PEIs in both host-side PCUs and memory-side PCUs, this is not enough to guarantee the atomicity of host-side PEI execution.

In order to guarantee the atomicity of both host-side and memory-side PEI execution, our architecture introduces a hardware structure that manages atomicity of PEI execution at the host side. Ideally, this structure would track *all* in-flight PEIs to ensure that each cache block has either only one writer PEI (i.e., a PEI that modifies its target cache block) or multiple reader PEIs (i.e., PEIs that only read their target cache block). However, this incurs a very large overhead since exact tracking of such information requires a fully associative table having as many entries as the total number of operand buffer entries of all PCUs in the system (which is equal to the maximum number of in-flight PEIs, as discussed in Section 4.2).

We develop a special hardware unit called *PIM directory* to manage atomicity of in-flight PEIs in a cost-effective manner. The key idea of the PIM directory is to allow rare false positives in atomicity management (i.e., serialization of two PEIs with different target cache blocks) for storage overhead reductions. This does not affect the atomicity of PEIs as long as there are no false negatives (e.g., simultaneous execution of two writer PEIs with the same target cache block).³ In order to exploit this idea, the PIM directory is organized as a direct-mapped, tag-less table indexed by XOR-folded addresses of target cache blocks. Each entry implements a reader-writer lock with four fields: (1) a readable bit, (2) a writeable bit, (3) an n -bit reader counter where $n = \lceil \log_2(\text{total number of operand buffer entries}) \rceil$, and (4) a 1-bit writer counter.

³Although too frequent false positives may incur a performance overhead due to unnecessary serialization, our evaluation shows that our mechanism achieves similar performance to its ideal version (an infinite number of entries) while incurring only a small storage overhead (see Section 7.6).

When a reader PEI arrives at the PIM directory, it is blocked until the corresponding PIM directory entry is in the readable state. After that, the entry is marked as non-writeable in order to block future writer PEIs during the reader PEI execution. At this moment, the reader PEI can be executed atomically. After the reader PEI execution, if there are no more in-flight reader PEIs to the entry, the entry is marked as writeable.

When a writer PEI arrives, it first needs to ensure that there are no in-flight writer PEIs for the corresponding PIM directory entry since atomicity allows only a single writer PEI for each cache block. Then, the entry is marked as non-readable to avoid write starvation by future reader PEIs. After that, the writer PEI waits until all in-flight reader PEIs for the PIM directory entry are completed (i.e., until the PIM directory entry becomes writeable), in order to prevent the reader PEIs from reading the cache block in the middle of the writer PEI execution. Finally, the write PEI is executed atomically and, upon its completion, the state of the entry is set to readable.

In addition to atomicity management, the PIM directory also implements the pfence instruction explained in Section 3.2. When a pfence instruction is issued, it waits for each PIM directory entry to become readable. This ensures that all in-flight writer PEIs issued before the pfence are completed when the pfence returns.

Cache Coherence Management. In our PIM abstraction, PEIs should be interoperable with existing cache coherence protocols. This is easily achievable for host-side PEI execution since host-side PCUs share L1 caches with their host processors. On the other hand, PEIs offloaded to main memory might read stale values of their target cache blocks if on-chip caches have modified versions of these blocks.

Our solution to this problem is simple. Due to the single-cache-block restriction, when the PMU receives a PEI, it knows exactly which cache block the PEI will access. Thus, it simply requests back-invalidation (for writer PEIs) or back-writeback⁴ (for reader PEIs) for the target cache block to the last-level cache before sending the PIM operation to memory. This ensures that neither on-chip caches nor main memory has a stale copy of the data before/after PIM operation execution, without requiring complex hardware support for extending cache coherence protocols toward the main memory side. Note that back-invalidation and back-writeback happen infrequently in practice since our architecture offloads PEIs to memory *only if* the target data are not expected to be present in on-chip caches.

Locality Monitor. One of the key features of our architecture is locality-aware PEI execution. The key idea is to decide whether to execute a PEI locally on the host or remotely in memory. This introduces a new challenge: dynamically profiling the data locality of PEIs. Fortunately, our single-cache-block restriction simplifies this problem since data locality of

⁴We use the term ‘back-writeback’ to denote a writeback request that forces writing back the target cache block from any of the L1, L2, or L3 caches it is present in to main memory (analogous to back-invalidation in inclusive cache hierarchies).

each PEI depends only on the locality of a *single* target cache block. With that in mind, the remaining issue is to monitor locality of the cache blocks accessed by the PEIs, which is done by the *locality monitor* in the PMU.

The locality monitor is a tag array with the same number of sets/ways as that of the last-level cache. Each entry contains a valid bit, a 10-bit partial tag (constructed by applying folded-XOR to the original tags), and replacement information bits. Each last-level cache access leads to hit promotion and/or block replacement for the corresponding locality monitor entry (as in the last-level cache).

The key difference between the locality monitor and the tag array of the last-level cache is that the former is also updated *when a PIM operation is issued to memory*. More specifically, when a PIM operation is sent to main memory, the locality monitor is updated as if there is a last-level cache access to its target cache block. By doing so, the locality of PEIs is properly monitored regardless of the location of their execution.

In our locality monitor, the data locality of a PEI can be identified by checking to see if its target cache block address hits in the locality monitor. The key idea behind this is that, if some cache blocks are accessed frequently (i.e., high locality), they are likely to be present in the locality monitor. However, according to our evaluations, if a locality monitor entry is allocated by a PIM operation, it is often too aggressive to consider it as having high locality on its first hit in the locality monitor. Therefore, we add a 1-bit ignore flag per entry to ignore the first hit to an entry allocated by a PIM operation (those allocated by last-level cache accesses do not set this ignore flag).

4.4. Virtual Memory Support

Unlike many other PIM architectures, virtual memory is easily supported in our architecture since PEIs are part of the conventional ISA. When a host processor issues a PEI, it simply translates the virtual address of the target cache block by accessing its own TLB. By doing so, all PCUs and the PMU in the system handle PEIs with physical addresses only.

This scheme greatly improves the practicality of our architecture. First, it avoids the overhead of adding address translation capabilities in memory. Second, existing mechanisms for handling page faults can be used without modification because page faults are still handled only at the host processor (i.e., no need to handle page faults in memory). Third, it does not increase TLB pressure since the single-cache-block restriction guarantees that only one TLB access is needed for each PEI just as a normal memory access operation.

4.5. PEI Execution

Host-side PEI Execution. Figure 4 illustrates the execution sequence of a PEI with high data locality. First, the host processor sends the input operands of the PEI to the PCU and issues it ①. The host-side PCU has a set of memory-mapped registers for control and temporary storage of input operands. Next, the host-side PCU accesses the PMU to (1) obtain a reader-writer lock for the target cache block from the PIM

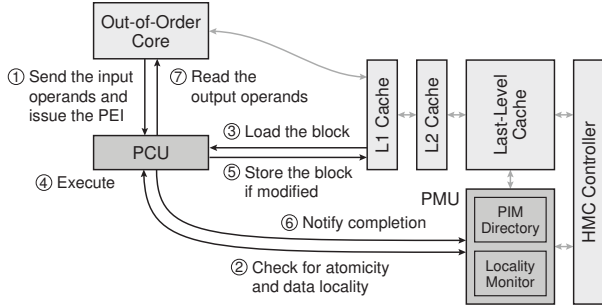


Figure 4: Host-side PEI execution.

directory and (2) consult the locality monitor to decide the best location to execute the PEI (2). In this case, the locality monitor advises the execution of the PEI on the host-side PCU as the target block is predicted to have high data locality. After that, the host-side PCU allocates a new operand buffer entry, copies the input operands from the memory-mapped register to this entry, and loads the cache block from the L1 cache (3). Once the target cache block is loaded, the PCU executes the PEI (4) and initiates a store request to the L1 cache if the PEI modifies the target cache block data (5). When the PEI execution is complete, the host-side PCU notifies the completion to the PMU in background to release the corresponding PIM directory entry (6). Finally, the host processor reads the output operands through memory-mapped registers inside the PCU and the operand buffer entry is deallocated (7).

Memory-side PEI Execution. Figure 5 shows the procedure of executing a PEI in main memory. Steps (1) and (2) are the same as in the above case, except that the locality monitor advises the execution of the PEI in memory. While this decision is being returned to the host-side PCU, the PMU sends a back-invalidation/back-writeback signal to the last-level cache to clean any local copy of the target cache block (3). At the same time, the host-side PCU transfers the input operands stored in its memory-mapped registers to the PMU (4). Once both steps are complete, the PMU sends the PIM operation to main memory by packetizing the operation type, the target cache block address, and the input operands (5). Upon receiving the response from the main memory (6), the PMU releases the corresponding PIM directory entry and sends the output operands back to the host-side PCU (7) so that the host processor can read them (8).

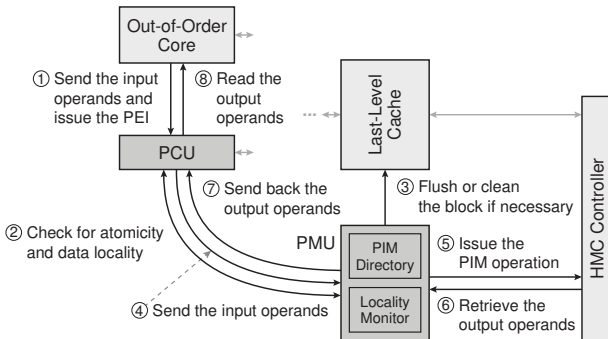


Figure 5: Memory-side PEI execution.

5. Target Applications for Case Study

The primary target of our architecture is applications with large memory footprint and very large memory bandwidth consumption. This makes it difficult to use standard benchmarks for evaluation as many standard benchmarks do not stress off-chip memory. Instead, we perform a case study on ten emerging data-intensive workloads from three important domains, which are often classified as “big-data” workloads. Evaluating such big-data applications on PIM architectures is very important since they are envisioned as the principal target for PIM due to their importance, broad applicability, and enormous memory bandwidth demand [2, 41]. Note that other applications can also be accelerated under our general-purpose framework by implementing (possibly) different types of PEIs.

5.1. Large-Scale Graph Processing

Average Teenage Follower (ATF) [19] is an example kernel of social network analysis. It counts for each vertex the number of its teenage followers by iterating over all teenager vertices and incrementing the ‘follower’ counters of their successor vertices. Since this generates a very large amount of random memory accesses over the entire graph (pointer chasing over edges), we implement *8-byte atomic integer increment* as a PIM operation to accelerate it.

Breadth-First Search (BFS) [19] is a graph traversal algorithm, which visits vertices closer to a given source vertex first. Our implementation is based on the parallel level-synchronous BFS algorithm [18, 19], where each vertex maintains a ‘level’ field to track the progress of traversal. Since each vertex updates the level fields of its neighbors by a min function, we implement *8-byte atomic integer min* as a PIM operation, which accepts an 8-byte input operand that replaces the target 8-byte word in memory if the input operand is smaller than the target word in memory.

PageRank (PR) [18] is a well-known algorithm that calculates the importance of vertices in a graph (see Figure 1). We implement *double-precision floating point addition* as a PIM operation for this application as discussed in Section 2.2.

Single-Source Shortest Path (SP) [19] finds the shortest path from a given source vertex to other vertices in a graph. Our application uses the parallel Bellman-Ford algorithm, which repeatedly iterates over vertices with distance changes and relaxes their outgoing edges with min functions. Therefore, our implementation uses the *atomic integer min operation* that we already discussed for BFS.

Weakly Connected Components (WCC) [24] identifies weakly connected components of a graph (a set of vertices that are reachable from each other when edge direction is ignored). The algorithm assigns a unique label for each vertex and collapses the labels of connected vertices into the smallest one by propagating labels through edges. Thus, this workload can also benefit from the *atomic integer min operation*.

5.2. In-Memory Data Analytics

Hash Join (HJ) [3] joins two relations from an in-memory database by building a hash table with one relation and probing

it with keys from the other. For this application, we implement a PIM operation for hash table probing, which checks keys in a given bucket for a match and returns the match result and the next bucket address. The host processor issues this PIM operation for the next bucket by performing pointer chasing with the returned next bucket address. We also modify the software to unroll multiple hash table lookups for different rows to be interleaved in one loop iteration. This allows out-of-order cores to overlap the execution of multiple PIM operations with the use of out-of-order execution.

Histogram (HG) builds a histogram with 256 bins from 32-bit integer data. In order to reduce memory bandwidth consumption of reading the input data, we implement a PIM operation that calculates the bin indexes of data in memory by shifting each 4-byte word in a 64-byte cache block with a given shift amount, truncating each word into a 1-byte value, and returning all 16 of them as a 16-byte output operand.

Radix Partitioning (RP) [3] is a data partitioning algorithm for an in-memory database, which acts as a preprocessing step for many database operations. Since it internally builds a histogram of data before partitioning the data, it can be accelerated by using the PIM operation for HG. However, unlike HG where input data are read only once in a streaming manner, radix partitioning accesses the original data again after building the histogram to move the data to corresponding partitions. We simulate a usage scenario where this algorithm is applied to given input data 100 times, which resembles access patterns of database servers continuously receiving queries to the same relation.

5.3. Machine Learning and Data Mining

Streamcluster (SC) [4] is an online clustering algorithm for n -dimensional data. The bottleneck of this algorithm is in computing the Euclidean distance of two points. To accelerate this computation, we implement a PIM operation that computes the distance between two 16-dimensional single-precision floating-point vectors, one stored in its target cache block (A) and the other passed as its input operand (B). Since the application uses this kernel to calculate distance from *few* cluster centers to *many* data points, we use the PIM operation by passing a cluster center as B and a data point as A , to preserve the locality of the cluster centers.

Support Vector Machine Recursive Feature Elimination (SVM) [38] selects the best set of features that describe the data in support vector machine (SVM) classification. It is extensively used in finding a compact set of genes that are correlated with disease. The SVM kernel inside it heavily computes dot products between a single hyperplane vector (w) and a very large number of input vectors (x). Thus, we implement a PIM operation that computes the dot product of two 4-dimensional double-precision floating-point vectors, similar to the PIM operation for SC.

5.4. Operation Summary

Table 1 summarizes the PIM operations implemented in this paper. It also shows reader/writer flags (e.g., if the ‘W’ column

of an operation is marked as ‘O’, it indicates that the operation modifies the target cache block) and the sizes of input/output operands. All these operations are supported by both the host-side and the memory-side PCUs in our system.

Table 1: Summary of Supported PIM Operations

Operation	R	W	Input	Output	Applications
8-byte integer increment	O	O	0 bytes	0 bytes	AT
8-byte integer min	O	O	8 bytes	0 bytes	BFS, SP, WCC
Floating-point add	O	O	8 bytes	0 bytes	PR
Hash table probing	O	X	8 bytes	9 bytes	HJ
Histogram bin index	O	X	1 byte	16 bytes	HG, RP
Euclidean distance	O	X	64 bytes	4 bytes	SC
Dot product	O	X	32 bytes	8 bytes	SVM

6. Evaluation Methodology

6.1. Simulation Configuration

We use an in-house x86-64 simulator whose frontend is based on Pin [33]. Our simulator models microarchitectural components in a cycle-level manner, including out-of-order cores, multi-bank caches with MSHRs, on-chip crossbar network, the MESI cache coherence protocol, off-chip links of HMCs, and DRAM controllers inside HMCs. Table 2 summarizes the configuration for the baseline system used in our evaluations, which consists of 16 out-of-order cores, a three-level inclusive cache hierarchy, and 32 GB main memory based on HMCs.

Table 2: Baseline Simulation Configuration

Component	Configuration
Core	16 out-of-order cores, 4 GHz, 4-issue
L1 I/D-Cache	Private, 32 KB, 4/8-way, 64 B blocks, 16 MSHRs
L2 Cache	Private, 256 KB, 8-way, 64 B blocks, 16 MSHRs
L3 Cache	Shared, 16 MB, 16-way, 64 B blocks, 64 MSHRs
On-Chip Network	Crossbar, 2 GHz, 144-bit links
Main Memory	32 GB, 8 HMCs, daisy-chain (80 GB/s full-duplex)
HMC	4 GB, 16 vaults, 256 DRAM banks [20]
– DRAM	FR-FCFS, tCL = tRCD = tRP = 13.75 ns [27]
– Vertical Links	64 TSVs per vault with 2 Gb/s signaling rate [23]

In our system, each PCU has a single-issue computation logic and a four-entry operand buffer (512 bytes). Thus, a PCU can issue memory requests of up to four in-flight PEIs in parallel but executes each PEI serially. We set the clock frequency of host-side and memory-side PCUs to 4 GHz and 2 GHz, respectively. Our system has 16 host-side PCUs (one per host processor) and 128 memory-side PCUs (one per vault).

Within the PMU design, the PIM directory is implemented as a tag-less four-bank SRAM array that can keep track of 2048 reader-writer locks. The storage overhead of the PIM directory is 3.25 KB, or 13 bits per entry (a readable bit, a writeable bit, a 10-bit⁵ reader counter and a 1-bit writer counter). The locality monitor has 16,384 sets and 16 ways, which is the same as the organization of the tag array of the L3 cache.

⁵Since our system has 576 (= 16 × 4 + 128 × 4) operand buffers in total, at least 10 bits are needed to safely track the number of in-flight PEIs.

Since each entry consists of a valid bit, a 10-bit partial tag, 4-bit LRU information, and a 1-bit ignore flag, the storage overhead of the locality monitor is 512 KB (3.1% of the last-level cache capacity). The access latency of the PIM directory and the locality monitor is set to two and three CPU cycles, respectively, based on CACTI 6.5 [37].

6.2. Workloads

We simulate ten real-world applications presented in Section 5 for our evaluations. To analyze the impact of data locality, we use three input sets for each workload as shown in Table 3. All workloads are simulated for two billion instructions after skipping their initialization phases.

Table 3: Input Sets of Evaluated Applications

Application	Input Sets (Small/Medium/Large)
ATF, BFS, PR, SP, WCC	soc-Slashdot0811 (77 K vertices, 905 K edges) [45] / frwiki-2013 (1.3 M vertices, 34 M edges) [29] / soc-LiveJournal1 (4.8 M vertices, 69 M edges) [45]
HJ	$R = 128\text{ K}/1\text{ M}/128\text{ M}$ rows, $S = 128\text{ M}$ rows
HG	$10^6/10^7/10^8$ 32-bit integers
RP	128 K/1 M/128 M rows
SC	4 K/64 K/1 M of 32/128/128-dimensional points [4]
SVM	50/130/253 instances from Ovarian cancer dataset [38]

7. Evaluation Results

In this section, we evaluate four different system configurations described below. Unless otherwise stated, all results are normalized to Ideal-Host.

- **Host-Only** executes all PEIs on host-side PCUs only. This disables the locality monitor of the PMU.
- **PIM-Only** executes all PEIs on memory-side PCUs only. This also disables the locality monitor.
- **Ideal-Host** is the same as Host-Only except that its PIM directory is infinitely large and can be accessed in zero cycles. This configuration represents *an idealized conventional architecture*. In this configuration, our PEIs are implemented as normal host processor instructions since atomicity of operations is preserved exactly without incurring any cost.
- **Locality-Aware** executes PEIs on both host-side and memory-side PCUs based on our locality monitor.

7.1. Performance

Figure 6 presents a performance comparison of the four systems under different workload input set sizes. All results are normalized to Ideal-Host. The last bars labeled as ‘GM’ indicate geometric mean across 10 workloads.

First, we confirm that the effectiveness of simple PIM operations highly depends on data locality of applications. For large inputs, PIM-Only achieves 44% speedup over Ideal-Host since using PIM operations better utilizes the vertical DRAM bandwidth inside HMCs (2x the internal bandwidth utilization on average). However, for small inputs, PIM-Only degrades average performance by 20% because PIM operations *always* access DRAM even though the data set comfortably fits in on-chip caches (causing 17x the DRAM accesses on average).

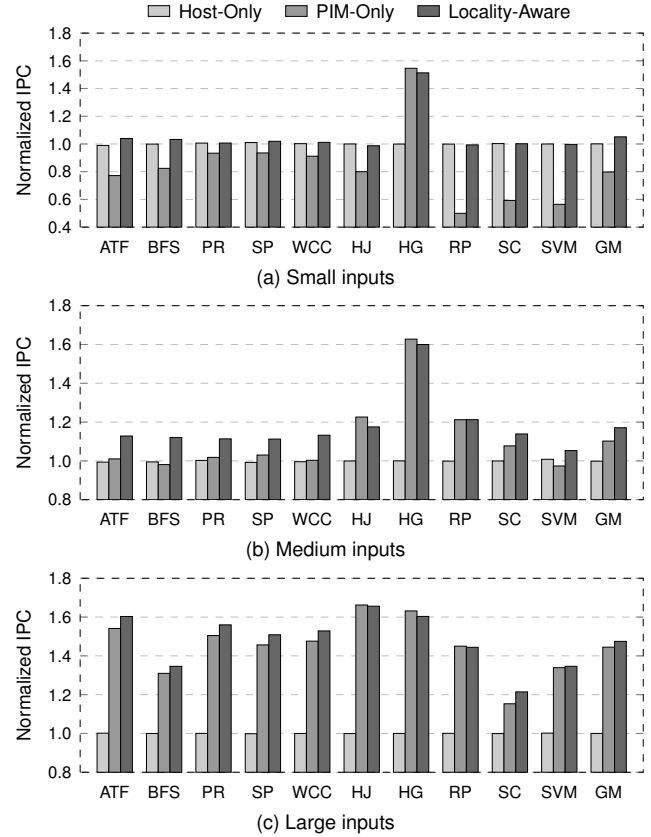


Figure 6: Speedup comparison under different input sizes.

In order to analyze the impact of PIM operations on off-chip bandwidth consumption, Figure 7 shows the total amount of off-chip transfers in Host-Only and PIM-Only, normalized to Ideal-Host. For large inputs, PIM-Only greatly reduces off-chip bandwidth consumption by performing computation in memory and bringing *only* the necessary results to the host processor.⁶ However, using PIM operations for small inputs, which usually fit in the large on-chip caches, greatly increases the off-chip bandwidth consumption (up to 502x in SC).

Our architecture exploits this trade-off between Host-Only and PIM-Only execution by adapting to the data locality of applications. As shown in Figure 6, our Locality-Aware system provides the speedup of PIM-Only in workloads with large inputs (47% improvement over Host-Only) by offloading 79% of PEIs to memory-side PCUs. At the same time, our proposal maintains the performance of Host-Only in workloads with small inputs (32% improvement over PIM-Only) by executing 86% (96% if HG is excluded) of PEIs on host-side PCUs.

More importantly, Locality-Aware often outperforms *both* Host-Only and PIM-Only by simultaneously utilizing host-side and memory-side PCUs for PEI execution. This is especially noticeable in graph processing workloads with medium inputs (i.e., ATF, BFS, PR, SP, and WCC in Figure 6b), where Locality-Aware achieves 12% and 11% speedup over Host-

⁶An exception to this trend is observed in SC and SVM with large inputs, which will be discussed in Section 7.4.

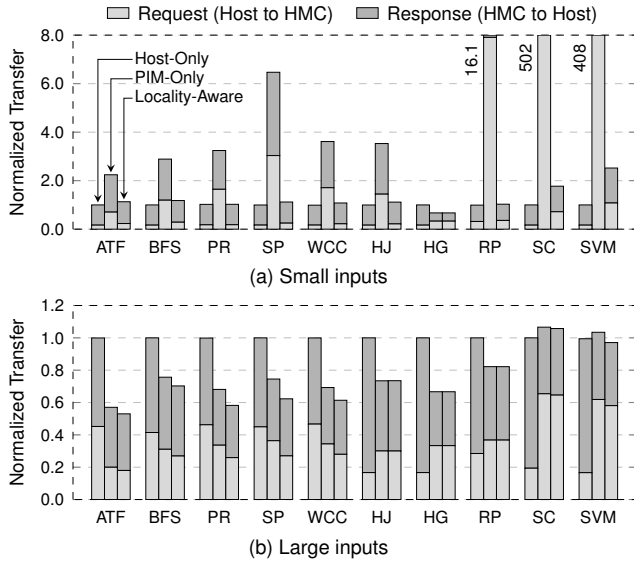


Figure 7: Normalized amount of off-chip transfer.

Only and PIM-Only, respectively. The major reason is that, in those workloads, vertices have different data locality according to the shape of the graph. For example, in the PageRank algorithm shown in Figure 1, vertices with many incoming edges (called high-degree vertices) receive more updates than those with few incoming edges since the atomic increment at line 10 is propagated through outgoing edges. Such characteristics play an important role in social network graphs like the ones used in our evaluations since they are known to show a large variation in the number of edges per vertex (often referred to as power-law degree distribution property) [36]. In this context, Locality-Aware provides the capability of automatically optimizing computation for high-degree and low-degree vertices separately without complicating software programming.

7.2. Sensitivity to Input Size

Figure 8 compares the performance of Host-Only, PIM-Only, and Locality-Aware using the PageRank workload with nine input graphs. We use the same input graphs as in Figure 2. Graphs are sorted based on their number of vertices: graphs with larger numbers of vertices appear toward the right side of the figure. Figure 8 also depicts the fraction of PEIs executed on memory-side PCUs (denoted as ‘PIM %’).

Most notably, Locality-Aware gradually shifts from host-

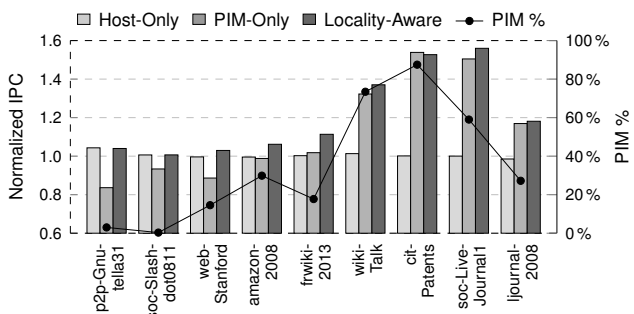


Figure 8: PageRank performance with different graph sizes.

side execution to memory-side execution as input size grows from left to right in the figure. For example, only 0.3% of the PEIs are executed on memory-side PCUs in soc-Slashdot0811, while this ratio reaches up to 87% in cit-Patents. This indicates the adaptivity of our architecture to a wide range of data locality behavior and different input sets.

We confirm that our locality monitoring scheme facilitates the use of both host-side and memory-side execution in a robust manner. For example, in amazon-2008 and frwiki-2013, which are medium-size input sets (and thus do not fully fit in on-chip caches), our technique enables a sizeable fraction of PEIs to be executed on the memory side, yet a sizeable fraction is also executed on the host side. This adaptive behavior of PEI execution shows the importance of hardware-based schemes for locality-aware PIM execution, as fine-grained (per-cache-block) information of data locality cannot be easily obtained with software-only approaches.

7.3. Multiprogrammed Workloads

To further analyze the benefit of our dynamic mechanism for locality-aware PIM execution, we evaluate our architecture with 200 multiprogrammed workloads. Each workload is constructed by randomly picking two target applications, each of which is set to spawn eight threads. We choose their input size uniformly at random from six possible combinations (small-small, medium-medium, large-large, small-medium, medium-large, and small-large). We use the sum of IPCs as a performance metric since most of our target applications are server workloads, which are throughput-oriented.

Figure 9 shows the performance of Locality-Aware and PIM-Only normalized to Host-Only, showing that our locality-aware architecture performs better than both Host-Only and PIM-Only execution for an overwhelming majority of the workloads. Hence, our architecture effectively selects the best place to execute PEIs even if applications with very different locality behavior are mixed. This is an important contribution since, without hardware-based locality monitoring mechanisms like ours, it is infeasible or very difficult for the software to determine where to execute a PEI, in the presence of multiple workloads scheduled at runtime. In many systems, a diverse spectrum of applications or workloads are run together and their locality behavior changes dynamically at a fine granularity. As such, it is critical that a technique like

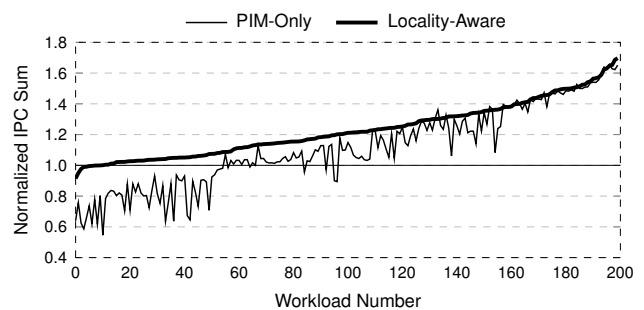


Figure 9: Performance comparison using randomly picked multiprogrammed workloads (normalized to Host-Only).

ours can dynamically profile the locality behavior and adapt PEI execution accordingly.

7.4. Balanced Dispatch: Idea and Evaluation

As shown in Figure 7, most of the speedup achieved by PIM-Only comes from the reduction in memory bandwidth consumption. However, we observe an intriguing behavior for some workloads: PIM-Only outperforms Host-Only in SC and SVM with large inputs even though it *increases* off-chip bandwidth consumption. This is because PIM-Only shows better balance between request and response bandwidth consumption (note that HMCs have separate off-chip links for requests and responses). For example, in SC, Host-Only reads 64-byte data per PEI, while PIM-Only sends 64-byte data to memory per PEI (see Table 1). Although the two consume nearly the same amount of memory bandwidth in total, the latter performs better because these two applications are read-dominated, which makes response bandwidth more performance-critical than request bandwidth.⁷

Leveraging this observation, we devise a simple idea called *balanced dispatch*, which relies on the host-side PEI execution capability of our architecture. In this scheme, PEIs are forced to be executed on host-side PCUs regardless of their locality if doing so achieves a better balance between request and response bandwidth. For this purpose, two counters, C_{req} and C_{res} , are added to the HMC controller to accumulate the total number of flits transferred through the request and response links, respectively (the counters are halved every 10 μ s to calculate the exponential moving average of off-chip traffic). When a PEI misses in the locality monitor, if C_{res} is greater than C_{req} (i.e., higher average response bandwidth consumption), our scheme chooses the one that consumes less response bandwidth between host-side and memory-side execution of that PEI. Similarly, if C_{req} is greater than C_{res} , the execution location with less request bandwidth consumption is chosen.

As shown in Figure 10, balanced dispatch further improves the performance of our architecture by up to 25%. We believe that this idea can be generalized to other systems with separate request/response channels such as buffer-on-board memory systems [7] (e.g., Intel SMB [22], IBM Centaur [13], etc.), the evaluation of which is left for our future work.

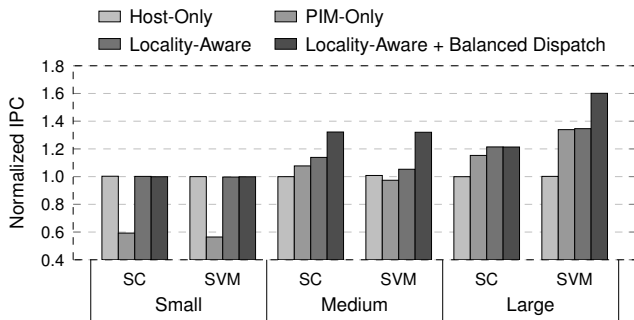


Figure 10: Performance improvement of balanced dispatch.

⁷To be more specific, a memory read consumes 16/80 bytes of request/response bandwidth and a memory write consumes 80 bytes of request bandwidth in our configuration. Thus, if an application is read-dominated, the response bandwidth is likely to be saturated first.

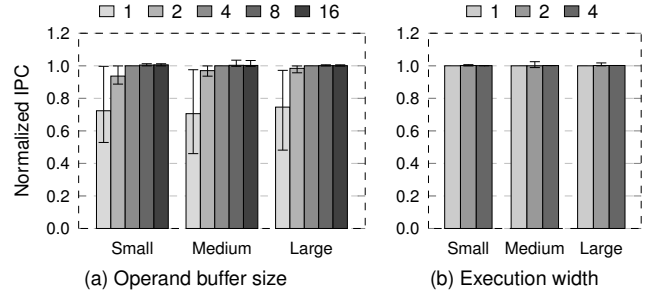


Figure 11: Performance sensitivity to different PCU designs.

7.5. Design Space Exploration for PCUs

Operand Buffer Size. Figure 11a shows the performance sensitivity of Locality-Aware to the operand buffer size in each PCU. The results are averaged over all applications and then normalized to the default configuration (four entries). Error bars show the minimum and the maximum values.

According to the results, incorporating a four-entry operand buffer per PCU (i.e., 576 in-flight PEIs) enables the PCU to exploit the maximum level of memory-level parallelism across PEIs, thereby improving the system performance by more than 30% compared to the one with a single-entry operand buffer. Having more than four operand buffer entries per PCU does not yield a noticeable difference in performance due to the saturation of instruction-level parallelism across PEIs.

Execution Width. Figure 11b depicts the impact of PCU execution width on the performance of Locality-Aware. As shown in the figure, increasing the issue width of the computation logic shows a negligible effect since execution time of a PEI is dominated by memory access latency.

7.6. Performance Overhead of the PMU

Compared to an ideal PMU with infinite storage, our PMU design can potentially degrade performance in three ways. First, the limited PIM directory size unnecessarily serializes two PEIs with different target cache blocks if the two cache blocks happen to be mapped to the same PIM directory entry. Second, partial tags of the locality monitor could potentially report false data locality if two cache blocks in a set have the same partial tag. Third, access latency of the PIM directory and the locality monitor delays PEI execution.

Fortunately, we observe that these sources of potential performance degradation have a negligible impact on system performance. According to our evaluations, idealizing the PIM directory and the locality monitor (with unlimited storage and zero latency) improves the performance of our architecture by only 0.13% and 0.31%, respectively. Therefore, we conclude that our PMU design supports atomicity and locality-aware execution of PEIs with negligible performance overhead.

7.7. Energy, Area, and Thermal Issues

Figure 12 shows the energy consumption of the memory hierarchy in Host-Only, PIM-Only, and Locality-Aware, normalized to that of Ideal-Host. We use CACTI 6.5 [37] to model energy consumption of on-chip caches, the PIM directory, and

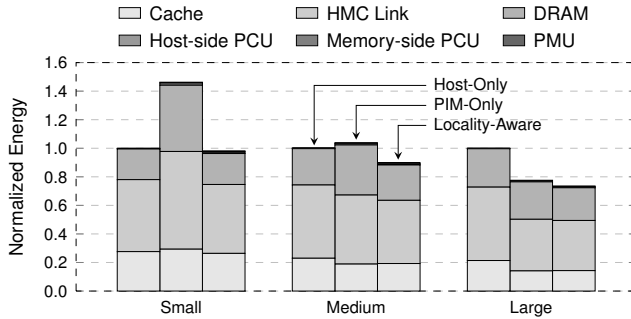


Figure 12: Energy consumption of memory hierarchy.

the locality monitor. The energy consumption of 3D-stacked DRAM, DRAM controllers, and off-chip links of an HMC is modeled by CACTI-3DD [6], McPAT 1.2 [30], and an energy model from previous work [27], respectively. The energy consumption of the PCUs is derived from synthesis results of our RTL implementation of computation logic and operand buffers based on the Synopsys DesignWare Library [47].

Among the three configurations, Locality-Aware consumes the lowest energy across all input sizes. For small inputs, it minimizes DRAM accesses by executing most of the PEIs at host-side PCUs, unlike PIM-Only, which increases energy consumption of off-chip links and DRAM by 36% and 116%, respectively. For large inputs, Locality-Aware saves energy over Host-Only due to the reduction in off-chip traffic and execution time. Hence, we conclude that Locality-Aware enables energy benefits over both Host-Only and PIM-Only due to its ability to adapt PEI execution to data locality.

Figure 12 also clearly indicates that our scheme introduces negligible energy overhead in existing systems. Specifically, the memory-side PCUs contribute only 1.4% of the energy consumption of HMCs. This implies that integration of such simple PIM operations into memory likely has a negligible impact on peak temperature, which is one of the important issues in 3D-stacked memory design from a practical viewpoint [8, 31]. Finally, our synthesis results and CACTI-3DD estimate the area overhead of memory-side PCUs to be only 1.85% of the logic die area, assuming that a DRAM die and a logic die occupy the same area.

8. Related Work

To our knowledge, this is the first work that proposes (1) a processing-in-memory execution model that is compatible with modern programming models and existing mechanisms for cache coherence and virtual memory, which is enabled by our new extensions to modern ISAs, called *PIM-enabled instructions (PEIs)*, and (2) a new mechanism, called *locality-aware PEI execution*, that adaptively determines whether a PEI should be executed in memory or in the host processor based on data locality, thereby achieving the benefits of both memory-side and host-side execution.

As we already explained in Section 1, most past works on PIM share two common limitations: (1) unconventional programming models for in-memory computation units and (2) limited interoperability with on-chip caches and virtual

memory [14, 16, 25, 39, 40, 46]. This is true even for recent studies that revisit the PIM concept using 3D-stacked DRAM technologies [1, 11, 41, 50, 51]. Although such approaches might achieve higher performance than conventional architectures, they require disruptive changes in both hardware and software, thereby hindering cost-effective and seamless integration of the PIM concept with existing systems. Our work overcomes these two major limitations.

Little has been studied on the effect of on-chip caches on PIM system design. With respect to cache coherence, most previous PIM approaches either operate on a noncacheable memory region [15, 39–41, 46], insert explicit cache block flushes into software code [16, 25, 44, 46], or require invalidations of a memory region [43], all of which introduce performance overhead and/or programming burden. Although it is possible to extend host-side cache coherence protocols to main memory [26], this incurs nontrivial overhead in implementing coherence protocols inside memory and, more importantly, tightly couples memory design to host processor design. Our architecture supports cache coherence without such drawbacks by introducing the single-cache-block restriction and managing coherence entirely from the host side.

To the best of our knowledge, no previous work considered cache locality in PIM systems from a performance perspective and proposed mechanisms to adaptively execute PIM operations either in the host processor or in memory. As shown in Section 2.2, simply offloading computation to memory all the time (as done in previous proposals) often noticeably degrades system performance when the computation exhibits high data locality. Our scheme provides adaptivity of execution location to data locality, thereby significantly improving performance and energy efficiency especially in the presence of varying dynamic factors (e.g., input sets and workload composition).

In terms of PIM operation granularity, the most relevant research to ours is active memory operations (AMOs) [9, 10], in which *on-chip memory controllers* support a limited set of simple computations. However, our approach is different from AMOs in at least three aspects. First, unlike our PIM-based architecture, computation in *on-chip memory controllers*, as opposed to in *memory*, still suffers from the off-chip bandwidth limitation, which is the bottleneck of our target applications. Second, AMOs are always executed in memory controllers, which requires cache block flushes for each AMO, thereby degrading performance compared to host-side execution under high data locality. This is not the case for our system since host-side PCUs see exactly the same memory hierarchy as the host processor. Third, the memory controller design for AMOs needs dedicated TLBs for virtual memory support, whereas our architecture achieves the same goal without such overhead by performing address translation with host processor TLBs.

9. Conclusion

In this paper, we proposed PIM-enabled instructions (PEIs), a practical model for processing-in-memory and its hardware implementation, which is compatible with existing cache coherence and virtual memory mechanisms. The key idea of

this paper is to express PIM operations by extending the ISA of the host processor with PEIs. This greatly improves the practicality of the PIM concept by (1) seamlessly utilizing the sequential programming model for in-memory computation, (2) simplifying the hardware support for interoperability with existing cache coherence and virtual memory mechanisms, and (3) minimizing area, power, and thermal overheads of implementing computation units inside memory. Importantly, our architecture is also capable of dynamically optimizing PEI execution according to the data locality of applications and PEIs. Our extensive evaluation results using emerging data-intensive workloads showed that our architecture combines the best part of conventional architectures and simple PIM operations in terms of both performance and energy consumption while minimizing the overhead of in-memory computation units and management structures for PEI execution. We conclude that our PEI abstraction and its implementation provide a practical approach to realizing high-performance and energy-efficient integration of in-memory computation capability into commodity computer systems in the near future.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in large part by the National Research Foundation of Korea (NRF) grants funded by the Korean government (MEST) (No. 2012R1A2A2A0604729 7) and the IT R&D program of MKE/KEIT (No. 10041608, Embedded System Software for New Memory-based Smart Devices). Onur Mutlu also acknowledges support from the Intel Science and Technology Center for Cloud Computing, Samsung, Intel, and NSF grants 0953246, 1065112, 1212962, and 1320531.

References

- [1] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ISCA*, 2015.
- [2] R. Balasubramonian *et al.*, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul. 2014.
- [3] C. Balkesen *et al.*, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *Proc. ICDE*, 2013.
- [4] C. Bienia *et al.*, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. PACT*, 2008.
- [5] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. WWW*, 1998.
- [6] K. Chen *et al.*, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Proc. DATE*, 2012.
- [7] E. Cooper-Balis *et al.*, "Buffer-on-board memory systems," in *Proc. ISCA*, 2012.
- [8] W. R. Davis *et al.*, "Demystifying 3D ICs: The pros and cons of going vertical," *IEEE Des. Test Comput.*, vol. 22, no. 6, pp. 498–510, Nov./Dec. 2005.
- [9] Z. Fang *et al.*, "Active memory operations," in *Proc. ICS*, 2007.
- [10] Z. Fang *et al.*, "Active memory controller," *J. Supercomput.*, vol. 62, no. 1, pp. 510–549, Oct. 2012.
- [11] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proc. HPCA*, Feb. 2015.
- [12] N. Firasta *et al.*, "Intel® AVX: New frontiers in performance improvements and energy efficiency," Intel Corporation, Tech. Rep., May 2008.
- [13] J. Friedrich *et al.*, "The POWER8™ processor: Designed for big data, analytics, and cloud environments," in *Proc. ICICDT*, 2014.
- [14] M. Gokhale *et al.*, "Processing in memory: The Terasys massively parallel PIM array," *IEEE Comput.*, vol. 28, no. 4, pp. 23–31, Apr. 1995.
- [15] Q. Guo *et al.*, "AC-DIMM: Associative computing with STT-MRAM," in *Proc. ISCA*, 2013.
- [16] M. Hall *et al.*, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proc. SC*, 1999.
- [17] S. Hong *et al.*, "Green-Marl: A DSL for easy and efficient graph analysis," in *Proc. ASPLOS*, 2012.
- [18] S. Hong *et al.*, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proc. PACT*, 2011.
- [19] S. Hong *et al.*, "Simplifying scalable graph processing with a domain-specific language," in *Proc. CGO*, 2014, pp. 208–218.
- [20] "Hybrid memory cube specification 1.0," Hybrid Memory Cube Consortium, Tech. Rep., Jan. 2013.
- [21] "Hybrid memory cube specification 2.0," Hybrid Memory Cube Consortium, Tech. Rep., Nov. 2014.
- [22] "Intel® C102/C104 scalable memory buffer datasheet," Intel, Feb. 2014.
- [23] J. Jeddelloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. VLSIT*, 2012.
- [24] U. Kang *et al.*, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. ICDM*, 2009.
- [25] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *Proc. ICCD*, 1999.
- [26] D. Keen *et al.*, "Cache coherence in intelligent memory systems," *IEEE Trans. Comput.*, vol. 52, no. 7, pp. 960–966, Jul. 2003.
- [27] G. Kim *et al.*, "Memory-centric system interconnect design with hybrid memory cubes," in *Proc. PACT*, 2013.
- [28] P. M. Kogge, "EXECUBE—a new architecture for scaleable MPPs," in *Proc. ICPP*, 1994.
- [29] Laboratory for Web Algorithmics. Available: <http://law.di.unimi.it/datasets.php>
- [30] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. MICRO*, 2009.
- [31] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *Proc. ISCA*, 2008.
- [32] G. H. Loh *et al.*, "A processing-in-memory taxonomy and a case for studying fixed-function PIM," presented at the *Workshop on Near-Data Processing*, 2013.
- [33] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.
- [34] A. Lumsdaine *et al.*, "Challenges in parallel graph processing," *Parallel Process. Lett.*, vol. 17, no. 1, pp. 5–20, Mar. 2007.
- [35] K. Mai *et al.*, "Smart memories: A modular reconfigurable architecture," in *Proc. ISCA*, 2000.
- [36] A. Mislove *et al.*, "Measurement and analysis of online social networks," in *Proc. IMC*, 2007.
- [37] N. Muralimanohar *et al.*, "CACTI 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, Apr. 2009.
- [38] R. Narayanan *et al.*, "MineBench: A benchmark suite for data mining workloads," in *Proc. IISWC*, 2006.
- [39] M. Oskin *et al.*, "Active pages: A computation model for intelligent memory," in *Proc. ISCA*, 1998.
- [40] D. Patterson *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar./Apr. 1997.
- [41] S. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory-logic devices on MapReduce workloads," in *Proc. ISPASS*, 2014.
- [42] S. Rusu *et al.*, "Ivytown: A 22nm 15-core enterprise Xeon processor family," in *International Solid-State Circuits Conference Digest of Technical Papers*, 2014.
- [43] V. Seshadri *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proc. MICRO*, 2013.
- [44] Y. Solihin *et al.*, "Automatic code mapping on an intelligent memory architecture," *IEEE Trans. Comput.*, vol. 50, no. 11, pp. 1248–1266, 2001.
- [45] Stanford Large Network Dataset Collection. Available: <http://snap.stanford.edu/data/index.html>
- [46] T. L. Sterling and H. P. Zima, "Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing," in *Proc. SC*, 2002.
- [47] Synopsys DesignWare Library – Datapath and Building Block IP. Available: <http://www.synopsys.com/dw/buildingblock.php>
- [48] S. Thoziyoor *et al.*, "PIM lite: A multithreaded processor-in-memory prototype," in *Proc. GLSVLSI*, 2005.
- [49] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.
- [50] D. P. Zhang *et al.*, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. HPDC*, 2014.
- [51] Q. Zhu *et al.*, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *Proc. 3DIC*, 2013.