LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory

Amirali Boroumand[†], Saugata Ghose[†], Minesh Patel[†], Hasan Hassan[†§], Brandon Lucia[†], Kevin Hsieh[†], Krishna T. Malladi[⋆], Hongzhong Zheng[⋆], and Onur Mutlu[‡†]

[†] *Carnegie Mellon University* [⋆] *Samsung Semiconductor, Inc.* [§] *TOBB ETÜ* [‡] *ETH Zürich*

Abstract—*Processing-in-memory* (PIM) architectures cannot use traditional approaches to cache coherence due to the high off-chip traffic consumed by coherence messages. We propose *LazyPIM*, a new hardware cache coherence mechanism designed specifically for PIM. LazyPIM uses a combination of speculative cache coherence and compressed coherence signatures to greatly reduce the overhead of keeping PIM coherent with the processor. We find that LazyPIM improves average performance across a range of PIM applications by 49.1% over the best prior approach, coming within 5.5% of an ideal PIM mechanism.

1 Introduction

Memory bandwidth continues to be a limiting factor for many applications today. In 3D-stacked DRAM architectures, vertical *through-silicon vias* provide very high bandwidth *within* memory, much higher than the bandwidth available to the processor. To exploit the high *internal* bandwidth, several recent works explore *processing-in-memory* (PIM), also known as *near-data processing* (e.g., [1, 2, 9, 10, 11, 13, 26, 27, 35]), where the processor dispatches parts of the application (which we refer to as *PIM kernels*) for execution at compute units (*PIM cores*) within DRAM. PIM architectures have been proposed for decades (e.g., [17,23,28,31]), but the greater bandwidth in 3D-stacked DRAM has made PIM much more appealing and easier to adopt.

Cache coherence is a major system challenge for PIM architectures. If PIM cores are coherent with the processor, the PIM programming model remains similar to conventional multithreaded programming, facilitating the wide adoption of PIM. However, it is impractical for PIM to perform traditional coherence, as this forces a large number of coherence messages to traverse a narrow off-chip bus, potentially undoing the benefits of high-bandwidth PIM execution. 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., [1, 9, 35]) or force processor cores to flush or not access any data that could *potentially* be used by PIM (e.g., [2, 9, 10, 11, 26, 27]).

We make two *key observations* based on our analysis of several data-intensive applications: (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 Sec. 3). **Our goal** in this work is to propose 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 cores. 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 Sec. 4.1), the dirty cache lines in the processor are flushed, and the PIM core rolls back and re-executes the kernel. Our execution model *for the PIM cores* is similar to *chunk-based execution* [6] (i.e., each *batch* of consecutive instructions executes atomically), which prior work has harnessed for various purposes [6, 7, 12]. Unlike past works, however, the processor in LazyPIM executes conventionally and *never rolls back*, which can make it easier to enable PIM.

Manuscript submitted 15 February 2016; revised 25 April 2016; accepted 18 May 2016. Final manuscript received 3 June 2016.

This work was supported by Google, Intel, NVIDIA, Samsung, Seagate, the Intel Science and Technology Center for Cloud Computing, the Semiconductor Research Corporation, and NSF grants 1212962, 1320531, and 1409723.

We make the following key contributions in this work:

- 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.

2 Baseline PIM Architecture

A number of 3D-stacked DRAM architectures [18, 19], such as HBM [15] and HMC [14], include a *logic layer*, where architects can implement functionality that interacts with both the processor and the DRAM cells [14]. Recent PIM proposals (e.g., [1, 2, 9, 10, 11, 35]) add compute units to the logic layer to exploit the high bandwidth available. In our evaluation, we assume that the compute units consist of simple *in-order* cores. These 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.

Each PIM core has private L1 I/D caches, which are kept coherent using a MESI directory 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 [1, 9], we assume that direct segments [3] are used for PIM data, and that PIM kernels operate only on physical addresses.

3 Motivation

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 the PIM cores. On the other hand, compute-intensive parts or those parts that exhibit high locality *must remain on the processor cores* to maximize performance [2, 13].

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 [29, 34]. We study a number of these algorithms [29], 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 [20,24,32]. The analytical portions of these databases are well suited for PIM execution [16,21,33]. In contrast, even though transactional queries access the *same* data, they perform better if they are executed on the processor, 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) 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 bus (we call this *PIM coherence traffic*). Fig. 1 shows the speedup of PIM with different coherence mechanisms for certain graph workloads, normalized to a *CPU-only* baseline (where the whole application runs on the processor).[1] To illustrate the impact of inefficient mechanisms, we also show the performance of an *ideal* mechanism where there is no performance penalty for coherence (*Ideal-PIM*). As shown in Fig. 1, PIM with fine-grained coherence (*FG*) always performs worse than CPU-only execution.

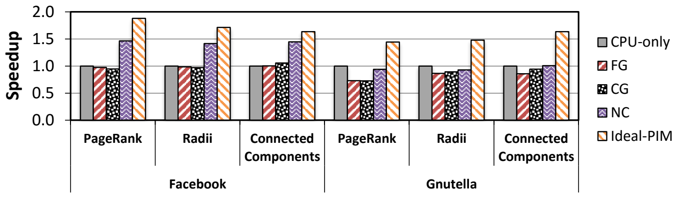

Fig. 1. PIM speedup with 16 threads, normalized to CPU-only.[1]

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.

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*.[1] Coherence at a smaller granularity, such as page-granularity [10], 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 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 applicaton 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 Fig. 1) 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 [1], so that DRAM always contains up-to-date data. For applications where PIM data is almost exclusively accessed by the PIM cores, this incurs little penalty, but for many applications, the processor also accesses PIM data often. For our graph applications with a representative input (arXiV),[1] 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 high performance and bandwidth overhead. As shown in Fig. 1, 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 when 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 LazyPIM Mechanism

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 enabling easier adoption of PIM.

LazyPIM avoids the pitfalls of the mechanisms discussed in Sec. 3. With its compressed signatures, LazyPIM causes much less PIM coherence traffic than traditional 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. 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 and allowing processor threads to continue using their caches.

4.1 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 without violating the memory consistency model, 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 Sec. 4.2.) At this point, a *conflict* may be detected on a cache line read by the PIM kernel, where the processor cache contains an *newer* copy of the line.

Fig. 2 (left) 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. Fig. 2 (left) 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 after the kernel finishes (see Sec. 4.2.2).

1. See Sec. 5 for our methodology.

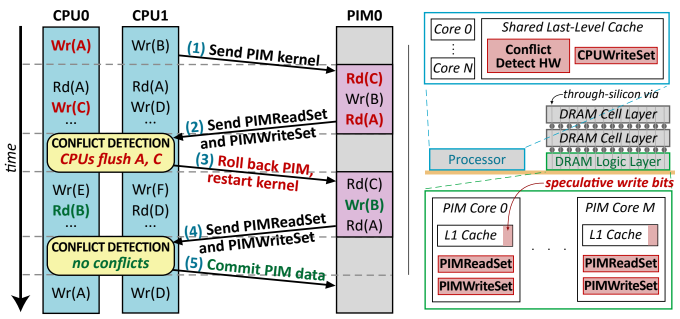

Fig. 2. Example timeline of LazyPIM coherence sequence (left); high-level additions (in bold) to PIM architecture to support LazyPIM (right).

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 Fig. 2), 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 kernel's writes, by marking them as speculative until the kernel finishes (see Sec. 4.2.2). 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.

4.2 LazyPIM Architectural Support

4.2.1 Program Interface

We provide a simple interface to port applications to LazyPIM. First, 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. Second, we assume either the programmer or the compiler can annotate all of the PIM data. This information is saved in the page table using per-page flag bits.

4.2.2 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 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 Fig. 2 (right): (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 [25]). The processor uses the page table flag bits from Sec. 4.2.1 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 Sec. 4.2.3)

To reduce coherence overheads, the signatures use parallel Bloom filters [5], which employ simple Boolean logic to hash multiple addresses into a single (256B) fixed-length register. Addresses can be extracted and compared from the register [5, 6]. The hashing introduces a limited number of false positives. To store more addresses, we use multiple filters.

4.2.3 Handling Conflicts

As Fig. 2 (left) 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, all coherence directory entries for the PIM data region are locked to ensure atomicity. Finally, all signatures are erased.

If an overlap is found between the PIMReadSet and the CPUWriteSet, a conflict may have occurred. 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 finishes, conflict detection is performed again. LazyPIM guarantees forward progress by acquiring a lock for each line in the PIMReadSet after a number of rollbacks (we set this number, empirically, to 3 rollbacks).

4.2.4 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 (Sec. 4.2.1); (2) a 0.2% increase in PIM core L1 size to mark speculative data (Sec. 4.2.2); and (3) in the worst case, 12KB for the signatures per PIM core (Sec. 4.2.2). 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 784B.

5 Methodology

We study two types of data-intensive applications: graph workloads and databases. We use three Ligra [29] graph applications (PageRank, Radii, Connected Components), with input graphs constructed from real-world network datasets [30]: 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) [20,24,32] 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 [22]. We conservatively select candidate PIM kernels, choosing hotspots 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 cores. We modify each application to ship the hotspots to the PIM cores. We manually annotate the PIM data set.

For our evaluations, we modify the gem5 simulator [4]. We use the x86-64 architecture in full-system mode, and use DRAMSim2 [8] to perform detailed timing simulation of DRAM. Table 1 shows our system configuration.

TABLE 1
Evaluated system configuration.

Processor	4–16 cores, 3-wide, 64kB IL1/DL1, 2MB L2, MESI coherence
PIM	4–16 cores, 64kB IL1/DL1, MESI coherence
Memory	HMC [14], 1 channel, 16 vaults per cube, 16 banks per vault

6 Evaluation

We first analyze the off-chip traffic reduction of LazyPIM, which leads to bandwidth and energy savings. We then analyze LazyPIM's performance. We show results normalized to a processor-only baseline (*CPU-only*, as defined in Sec. 3), and compare with using fine-grained coherence (*FG*), coarse-grained locks (*CG*), or non-cacheable data (*NC*) for PIM data.

6.1 Off-Chip Traffic

Fig. 3 (left) shows the normalized off-chip traffic of the PIM coherence mechanisms for a 16-core architecture (with 16 processor cores and 16 PIM cores) Fig. 3 (right) shows the normalized off-chip traffic as the number of threads increases, for PageRank using the Facebook graph. LazyPIM significantly reduces *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 traffic by 58.8%, on average, over CG, the best prior approach in terms of traffic.

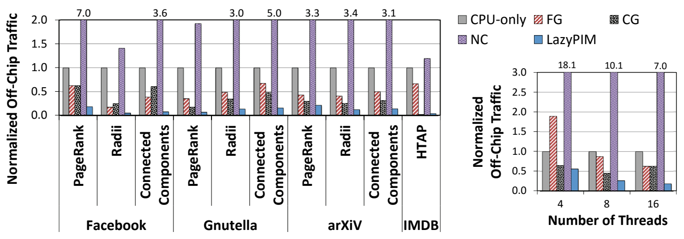

Fig. 3. 16-thread off-chip traffic (left), and off-chip traffic sensitivity to thread count (right), normalized to CPU-only.

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 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 traffic by 3.3x over CPU-only. NC off-chip 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.

6.2 Performance

Fig. 4 (left) shows the performance of PageRank using Gnutella as we increase the thread count. LazyPIM comes within 5.5% of Ideal-PIM (as defined in Sec. 3), 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 going to 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 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.

Fig. 4 (right) 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.

7 Conclusion

We propose LazyPIM, a new cache coherence mechanism for PIM architectures. Prior approaches to PIM coherence generate

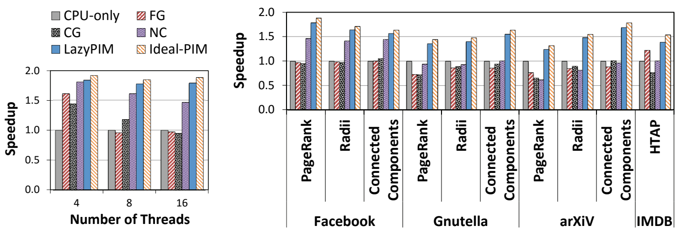

Fig. 4. Speedup sensitivity to thread count (left), and speedup for all applications with 16 threads (right), normalized to CPU-only.

very high off-chip traffic for important data-intensive applications. LazyPIM avoids this by avoiding coherence lookups *during* PIM kernel execution. Compressed coherence *signatures* are used to batch the lookups and verify correctness *after* the kernel completes. 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 same programming model.

References

[1] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[2] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[3] A. Basu *et al.*, "Efficient Virtual Memory for Big Memory Servers," in *ISCA*, 2013.
[4] N. Binkert *et al.*, "The gem5 Simulator," *Comp. Arch. News*, 2011.
[5] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, 1970.
[6] L. Ceze *et al.*, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, 2007.
[7] J. Devietti *et al.*, "DMP: Deterministic Shared Memory Multiprocessing," in *ASPLOS*, 2009.
[8] DRAMSim2, http://www.eng.umd.edu/ blj/dramsim/.
[9] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.
[10] M. Gao *et al.*, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[11] Q. Guo *et al.*, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
[12] L. Hammond *et al.*, "Transactional Memory Coherence and Consistency," in *ISCA*, 2004.
[13] K. Hsieh *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[14] Hybrid Memory Cube Specification 2.0, 2014.
[15] JEDEC, "JESD235: High Bandwidth Memory (HBM) DRAM," 2013.
[16] O. Kocberber *et al.*, "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases," in *MICRO*, 2013.
[17] P. M. Kogge, "EXECUBE: A New Architecture for Scaleable MPPs," in *ICPP*, 1994.
[18] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.
[19] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *ISCA*, 2008.
[20] MemSQL, Inc., "MemSQL," http://www.memsql.com/.
[21] N. Mirzadeh *et al.*, "Sort vs. Hash Join Revisited for Near-Memory Execution," in *ASBD*, 2007.
[22] OProfile, http://oprofile.sourceforge.net/.
[23] D. Patterson *et al.*, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
[24] SAP SE, "SAP HANA," http://www.hana.sap.com/.
[25] V. Seshadri *et al.*, "The Dirty-Block Index," in *ISCA*, 2014.
[26] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.
[27] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[28] D. E. Shaw *et al.*, "The NON-VON Database Machine: A Brief Overview," *IEEE Database Eng. Bull.*, 1981.
[29] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *PPoPP*, 2013.
[30] Stanford Network Analysis Project, http://snap.stanford.edu/.
[31] H. S. Stone, "A Logic-in-Memory Computer," *IEEE Trans. Comput.*, 1970.
[32] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS." *IEEE Data Eng. Bull.*, 2013.
[33] S. L. Xi *et al.*, "Beyond the Wall: Near-Data Processing for Databases," in *DaMoN*, 2015.
[34] J. Xue *et al.*, "Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing," in *HPDC*, 2014.
[35] D. P. Zhang *et al.*, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.