AIM: Energy-Efficient Aggregation Inside the Memory Hierarchy

JUNWHAN AHN, SUNGJOO YOO, and KIYOUNG CHOI, Seoul National University

In this article, we propose Aggregation-in-Memory (AIM), a new processing-in-memory system designed for energy efficiency and near-term adoption. In order to efficiently perform aggregation, we implement simple aggregation operations in main memory and develop a locality-adaptive host architecture for in-memory aggregation, called cache-conscious aggregation. Through this, AIM executes aggregation at the most energy-efficient location among all levels of the memory hierarchy. Moreover, AIM minimally changes existing sequential programming models and provides fully automated compiler toolchain, thereby allowing unmodified legacy software to use AIM. Evaluations show that AIM greatly improves the energy efficiency of main memory and the system performance.

CCS Concepts: • **Hardware** → **Dynamic memory**;

Additional Key Words and Phrases: Processing-in-memory, near-data processing, aggregation, locality-adaptive execution

ACM Reference Format:

Junwhan Ahn, Sungjoo Yoo, and Kiyoungh Choi. 2016. AIM: Energy-efficient aggregation inside the memory hierarchy. *ACM Trans. Archit. Code Optim.* 13, 4, Article 34 (October 2016), 24 pages.

DOI: <http://dx.doi.org/10.1145/2994149>

1. INTRODUCTION

The main memory system is a performance and energy efficiency bottleneck of modern computer systems. This trend has been exacerbated in the past few years with (1) architectural innovations for improving the efficiency of computation units (e.g., chip multiprocessors), which shift the major cause of inefficiency from processors to memory, and (2) the emergence of data-intensive workloads, which demands a large capacity of main memory and an excessive amount of memory bandwidth to efficiently handle such workloads.

One promising direction to overcome this performance/energy gap between processors and memory is *Processing-in-Memory (PIM)*. By moving computation to where data resides, the PIM concept enables low-latency and high-bandwidth accesses from computation units to memory-resident data without being limited by narrow off-chip channels. Moreover, it reduces the energy consumption of data transfer by shortening the distance between computation units and data storage. Fortunately, such advantages can now be achieved in a practical manner with 3D die stacking

This work is supported by Research Resettlement Fund for the new faculty of Seoul National University and the IT R&D program of MKE/KEIT (No. 10041608, Embedded System Software for New Memory based Smart Devices).

Authors' addresses: J. Ahn and K. Choi, Department of Electrical and Computer Engineering, Seoul National University, Seoul, Republic of Korea; emails: {junwhan, kchoi}@snu.ac.kr; S. Yoo, Department of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea; email: sungjoo.yoo@gmail.com. Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/10-ART34 \$15.00

DOI: <http://dx.doi.org/10.1145/2994149>

[Balasubramonian et al. 2014], unlike in the first PIM era in the 1990s [Kogge 1994; Patterson et al. 1997; Oskin et al. 1998; Elliott et al. 1999; Hall et al. 1999; Kang et al. 1999; Draper et al. 2002; Kim et al. 2002; Sterling and Zima 2002], which diminished due to the lack of cost-effective technologies for tight integration of logic and memory. This reignites the research on developing PIM systems based on 3D-stacked memory [Zhu et al. 2013; Pugsley et al. 2014; Zhang et al. 2014; Farmahini-Farahani et al. 2015; Nair et al. 2015; Ahn et al. 2015a, 2015b].

While the PIM concept itself gives the performance and energy efficiency benefits to existing architectures by reducing the amount of data transfer, little has been known about *optimizing* PIM architectures toward higher energy efficiency of main memory. More specifically, existing PIM proposals are suboptimal in terms of energy efficiency because (1) they are unaware of the DRAM energy characteristics (e.g., row activation energy) and (2) they always offload the target computation to memory regardless of the data locality, which can increase the energy consumption of main memory compared to conventional systems based on large on-chip caches.

In this article, we propose *Aggregation-in-Memory (AIM)*, a new PIM architecture designed for energy efficiency. AIM targets *aggregation operations*, which are defined as commutative and associative read-modify-write operations to memory-resident data. For instance, adding a given value, which we call *aggregation operand*, to data stored in memory belongs to aggregation operations. Such a type of operations is extensively used in many important data-intensive applications, including graph analysis, machine learning, histogram computation, and so on. In this article, we argue that offloading aggregation operations to memory greatly improves the energy efficiency of main memory in the following three ways:

- Fewer main memory accesses*: While conventional architectures issue one read and one write to memory per aggregation, in-memory aggregation sends only one aggregation operation to memory.
- Fewer row buffer misses*: An in-memory aggregation operation is processed as a single Dynamic Random Access Memory (DRAM) command, and thus, requires at most one row activation. On the other hand, conventional systems issue up to two row activations per aggregation since writeback caches hide the temporal locality between the read and the write of an aggregation operation from DRAM row buffers.
- Fewer bit-flips on off-chip channels*: In-memory aggregation sends only the aggregation operands, which usually have much smaller values (i.e., lower entropy in bits) than the target data, to memory.

However, the aforementioned benefits diminish when the target data of aggregation operations can be served by on-chip caches, in which case conventional architectures do not access main memory at all. Hence, we develop a host architecture that can *adaptively* exploit in-memory aggregation by considering data locality of applications at runtime. Our hardware mechanism, called *cache-conscious aggregation*, coalesces multiple aggregation operations to the same data into a single in-memory aggregation operation by utilizing on-chip caches. This further reduces main memory accesses by introducing locality awareness into PIM execution.

Moreover, AIM facilitates near-term adoption of PIM into existing systems. First, aggregation operations in AIM have the identical interface to the equivalent host processor instructions including full support for cache coherence and virtual memory, which realizes an intuitive programming model for PIM. Second, since a tiny Arithmetic Logic Unit (ALU) would suffice to support aggregation operations, in-memory aggregation can be implemented either on the existing logic die of 3D-stacked DRAM or on the DRAM die of commodity DDRx modules, thereby providing more options for practical PIM implementation.

This article makes the following contributions:

- We propose *AIM* for near-term adoption of energy-efficient PIM. *AIM* implements aggregation capability into every level of the memory hierarchy (e.g., on-chip caches, memory controllers, and main memory) and coordinates it in a locality-aware manner, which we call *processing-in-memory-hierarchy*.
- We develop a simple programming model and compiler support for *AIM*, which allow existing software to automatically exploit our mechanism with no modifications.
- We show that *AIM* improves the energy efficiency of DRAM by reducing off-chip data transfer, improving row buffer locality, and minimizing bit-flips on memory channels. In contrast, existing PIM proposals usually provide only the first benefit under scarce data locality.
- We quantitatively analyze the effectiveness of *AIM* using 10 important data-intensive workloads and show that *AIM* greatly improves energy efficiency and performance over both conventional systems and PIM-only systems.

2. MOTIVATION

2.1. Rethinking PIM for Energy Efficiency

Many recent PIM architectures reduce the energy consumption of the memory hierarchy by reducing off-chip traffic and/or improving system performance [Pugsley et al. 2014; Zhang et al. 2014; Farmahini-Farahani et al. 2015; Nair et al. 2015; Ahn et al. 2015b]. However, we argue that they are not fully optimized yet for energy efficiency of main memory in the following two aspects.

First, while moving computation to memory reduces off-chip I/O energy consumption by avoiding off-chip data transfer, it does not necessarily improve the energy efficiency of DRAM itself. In particular, simply performing computation in memory does not improve row buffer locality, which is a key factor in determining the DRAM energy efficiency (i.e., row activation energy) [Volos et al. 2014]. Therefore, there is plenty of room for further energy optimization in PIM system design.

Second, if the target applications exhibit enough data locality, utilizing PIM may even increase the *internal* energy consumption of main memory (e.g., row activation energy, read/write energy, etc.) compared to conventional architectures. This is because, contrary to the traditional memory hierarchy with large on-chip caches, in-memory computation units are usually equipped with a shallow cache hierarchy, thereby providing limited ability to exploit data locality for reducing internal main memory accesses. What is worse, data locality information of an application is often available only at runtime due to its input and/or system load dependence, which implies that static decision of whether to use PIM or not is impractical in many cases. Thus, PIM architectures should be able to dynamically adapt to data locality in order to be robust against dynamic characteristics of workloads.

2.2. Aggregation as PIM Operations

The goal of this work is to maximize the energy efficiency benefit of PIM while minimizing the implementation cost. Thus, it is favorable to choose the computation that is difficult for conventional architectures to handle in an energy-efficient manner but at the same time computationally simple.

One type of computation that satisfies these two criteria is *aggregation operations* over a large memory region. Formally, we define an aggregation operation $v \leftarrow x$ as a read-modify-write operation that computes $v * x$ for a commutative and associative operation “ $*$ ” and stores its value back to v . For example, an increment-by-one operation can be expressed as an aggregation operation $v \leftarrow v + 1$. Aggregation operations are importantly used in many data-intensive applications (see Section 6). A

```

1  for (v: graph.vertices) {
2    v.pagerank = 1 / graph.num_vertices;
3    v.next_pagerank = 0.15 / graph.num_vertices;
4  }
5  count = 0;
6  do {
7    diff = 0;
8    for (v: graph.vertices) {
9      value = 0.85 * v.pagerank / v.out_degree;
10     for (w: v.successors) {
11       w.next_pagerank += value;
12     }
13   }
14   for (v: graph.vertices) {
15     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 (diff > e && ++count < max_iteration);

```

Fig. 1. PageRank algorithm.

Fig. 2. Performing aggregation in (a) conventional, (b) PIM-only, and (c) AIM systems. Each arrow indicates one access.

representative example of them is the PageRank algorithm [Brin and Page 1998; Malewicz et al. 2010; Hong et al. 2012, 2014], which is widely used in web search engines and citation ranking. For large real-world graphs with millions or billions of vertices, the bottleneck of the algorithm is in updating `next_pagerank` of neighbor vertices (at line 11 of Figure 1), which belongs to an aggregation operation. This is because neighbor traversal randomly accesses the entire set of vertices with very small amounts of computation, and thus, demands very high memory bandwidth [Malewicz et al. 2010; Ahn et al. 2015a].

Despite the simplicity of aggregation operations, the conventional memory hierarchy is not optimized for processing them in an energy-efficient manner due to two reasons. First, one aggregation operation requires two memory accesses (read and write), which eventually become two main memory accesses under low data locality. Second and more importantly, one aggregation operation often incurs *two DRAM row activations*, one for read and the other for write, since on-chip caches postpone the write of the operation until the cache block eviction, which hides temporal locality between the read and the write from the row buffer.

Performing aggregation in memory solves the aforementioned inefficiencies of conventional architectures. Instead of transferring the target data back and forth between the host processor and main memory, in-memory aggregation simply sends the aggregation operand to main memory and lets the memory perform the aggregation *inside* it. This reduces the number of main memory accesses (as indicated by thick arrows in Figure 2(b)) under low data locality. Moreover, since in-memory aggregation is implemented as a single DRAM command, an aggregation operation incurs at most *one* row activation, instead of two as in conventional architectures.

Fig. 3. Overview of the AIM Architecture.

In order to fully exploit these benefits of in-memory aggregation, it is very important to design an intelligent host architecture as motivated previously. In particular, when the target data is stored in on-chip caches, executing all aggregation operations in memory not only increases main memory accesses but also generates extra cache traffic to flush any stale copy of the target data in on-chip caches before issuing in-memory aggregation. Ideally, the host architecture should be able to adaptively utilize both on-chip caches and in-memory aggregation capability in a way to minimize DRAM accesses, as conceptually shown in Figure 2(c). This motivates the design of our PIM system called *AIM*.

3. ARCHITECTURE

3.1. Overview

Figure 3 gives an overview of the AIM architecture. It also conceptually exemplifies the contents of cache blocks (white boxes in the caches) after executing an aggregation operation $v + \leftarrow 1$ to an 8-byte integer v stored in main memory (white box in the main memory bank). Although one cache block generally contains multiple 8-byte values, other parts of the block are omitted from the figure for better visibility.

Programming Model (Section 3.2). AIM interfaces aggregation operations to software as cache-coherent, virtually addressed host processor instructions. When the host processor issues an aggregation instruction, our hardware mechanism performs in-memory aggregation for the target data behind the scenes. This simplifies the programming model of AIM because the interface of in-memory aggregation is *identical* to the normal instruction for the equivalent operation.

Cache-Conscious Aggregation (Section 3.3). AIM avoids main memory accesses for aggregation operations with high data locality by *coalescing aggregation operations to the same data at on-chip caches*. The key idea is to store only the aggregation operands in caches without fetching the original data (e.g., “+1” in Figure 3). Cache-resident aggregation operands are later merged with the original data in main memory by performing in-memory aggregation.¹ To support aggregation coalescing in caches, all caches are equipped with *Aggregation Computation Unit (ACU)*, which is a tiny ALU for aggregation.

In-Memory Aggregation (Section 3.5). We implement in-memory aggregation by adding one ACU per DRAM bank. At the protocol level, in-memory aggregation is interfaced as a special write command of DRAM because both aggregation commands and write commands send cache-block-sized data to DRAM. Note that, although this article assumes commodity DDR3-based main memory (which eases the near-future integration of our idea into existing systems), AIM can also be implemented based on

¹While this may change the execution order of aggregation operations, it does not affect the program semantics because of the commutativity and associativity of aggregation operations.

3D-stacked DRAM (e.g., Hybrid Memory Cube [HMC 2014]), similar to other recent PIM proposals.

To support in-memory aggregation, we introduce two modifications to the memory controllers. First, we add one ACU per memory controller to correctly handle data dependence between read commands and in-flight aggregation commands. Second, we slightly change the data placement across DRAM chips in a DIMM to ensure that in-memory aggregation can be performed locally inside each DRAM chip. Such modifications are transparent from other system components.

3.2. Programming Model

Our architecture exposes aggregation operations to software as host processor instructions. To simplify the hardware design, we ensure that aggregation operands are always aligned in memory (e.g., 8-byte aggregation operands are aligned to 8-byte boundaries), just as modern compilers guarantee aligned memory access by default. By simply using the new instructions, existing software can benefit from the in-memory aggregation capability. Also, our cache-conscious aggregation automatically adapts to data locality with no software hints (Section 3.3), thereby letting compilers aggressively emit aggregation instructions without accurate estimation of data locality. This reduces the burden of developing compilers for our architecture (Section 4).

In addition, the simplicity of our programming model helps seamless integration of our idea into existing systems. In particular, AIM supports virtual memory for in-memory aggregation without in-memory Memory Management Units (MMUs) since the target address of an aggregation instruction (which is a virtual address) can be translated into a physical address by using the Table Lookaside Buffer (TLB) of the host processor, just as normal load/store instructions.

3.3. On-Chip Caches

In our architecture, the host processor executes an aggregation instruction by issuing an aggregation operation, which consists of an aggregation type (e.g., $+$, $\times$, etc.) and an aggregation operand, to the L1 cache. The following describes our cache architecture that supports aggregation operation execution.

Organization. In order to distinguish cache blocks that contain aggregation operands from ordinary blocks, the tag of each block is extended with an Aggregation Type (AT) field. If the AT field of a block is set to a nonzero value t , the block is called *aggregated block* and should contain *only* the aggregation operands for the aggregation operation of type t . In other words, a single cache block is not allowed to contain aggregation operands from multiple types of aggregation operations. The AT field is set to zero for ordinary cache blocks.

L1 Cache Miss. When an aggregation operation misses in the L1 cache, the L1 cache does *not* read the data from the L2 cache, unlike normal loads/stores. Instead, it simply inserts a new L1 cache block filled with *identity elements*² of the given aggregation type t and sets the AT field of the block to t . Since applying an aggregation operation whose operand is an identity element does not change the target data at all, when an aggregated block is merged back to the original data in memory (explained later in this subsection), the portions that are not updated by the host processor are left intact.

However, under the inclusive cache hierarchy, installing a new L1 cache block without accessing the L2 cache violates the inclusion property if the L2 cache does not have the block. Thus, for inclusive caches, the L1 cache sends a special L2 cache request, which lets the L2 cache install an aggregated block filled with identity elements only

²In mathematics, the identity element of a binary operation “ $*$ ” is defined as x that satisfies $a * x = a$ for all a (e.g., “0” for addition).

if the L2 cache does not have the block. This is almost the same as miss handling in conventional cache architectures, except that the contents of the L2 cache block are not transferred into the L1 cache.

L1 Cache Hit. When an aggregation operation hits in the L1 cache, the cache reads the target block, computes the result of the aggregation operation by using its ACU (see Figure 3), and stores the result back to the target block, all with no preemption. If the block is an aggregated one, this coalesces the current aggregation operand with the one in the cache, so that they can be sent as a single in-memory aggregation operation on the eviction of the block (explained later). If the block is an ordinary one, the cache directly performs the aggregation operation with the original data.

L1 Cache Block Upgrade. Since aggregated blocks do not store the original data, an aggregated block cannot directly service a normal load/store or an aggregation operation with a different type (commutativity and associativity of aggregation operations hold only for the same type). In such cases, the aggregated block B of type “*” is *upgraded* to an ordinary one by (1) requesting a normal L2 cache read to fetch its original data D , (2) performing $B * \leftarrow D$ to merge the cache-resident aggregation operands with the original data, and (3) setting the AT field of B to zero (i.e., ordinary).³ At this point, all cache accesses can be serviced as in other ordinary blocks.

L1 Cache Writeback. When an aggregated block is evicted from the L1 cache, the block is written back to the L2 cache by sending an aggregation operation (instead of a normal write operation) whose type and operand are the value of the AT field and the block data itself, respectively.

Sometimes, a cache block is requested to be written back without being evicted from the cache (e.g., coherence requests, eager writeback, etc.). The L1 cache handles this request by (1) sending an aggregation operation to the L2 cache as described previously, (2) *initializing* the contents of the block with identity elements, and (3) changing the block state from dirty to clean. The last two steps are necessary to prevent the block from being aggregated multiple times.

Upper-Level Caches. All upper-level caches (e.g., L2/L3 caches) operate in the same way as the L1 cache, except for the following differences for the last-level cache. First, when an aggregated block is evicted from the last-level cache, an in-memory aggregation operation is sent to the memory controller. Second, the last-level cache does not send inclusion management requests on misses for an obvious reason.

Example. Figure 4 exemplifies an operating sequence of our cache architecture for one cache block, which is initially stored in DRAM and is loaded into two-level inclusive caches. Each set of boxes shows the contents of the block in the L1/L2 cache and the DRAM. For brevity, we assume that the block is 2 bytes long and each instruction updates a 1-byte value (x or y). White/gray boxes indicate ordinary/aggregated blocks.

At the beginning, the original data 12 34 is stored in the DRAM (a). When the host processor issues $x \leftarrow 1$, the L1 cache sends an L2 cache request that inserts a new L2 cache block filled with zeros (identity elements of addition) for inclusion management. Then, it initializes a new zero-filled L1 cache block *without accessing the DRAM* and adds one to x in the block (b). The subsequent addition to x is handled simply by adding its operand to x in the L1 cache block (c). When the aggregated L1 cache block is evicted, it is coalesced into the corresponding L2 cache block (d). An addition operation to y is performed in the same manner (e). Note that the aggregation

³According to our evaluations, such upgrades are infrequent and thus have a negligible energy overhead, that is, only 2% of the aggregated L3 cache blocks are eventually upgraded to ordinary ones in our large-input workloads.

Fig. 4. An example of cache operations.

operands of both x and y can be stored in a single cache block because their aggregation types are identical (f). The aggregated L2 cache block is later merged into the original data by using in-memory aggregation (g). When a normal instruction accesses an aggregated block, the cache upgrades the block into an ordinary one by loading the original data (h and i). An aggregation operation for an ordinary block is handled in the same way as aggregated blocks (j).

Advantages. Our cache-conscious aggregation provides two major benefits. First, when the target applications have high data locality, our scheme reduces main memory accesses by coalescing multiple aggregation operations to the same data into a single in-memory aggregation operation. Since our mechanism directly uses on-chip caches as locality filters, incorporating any intelligent cache management technique will improve the effectiveness of cache-conscious aggregation.

Second, performing aggregation in caches automatically enables cache coherence support, not only among different cores in the system but also between the host processor and main memory. In a naive approach where the host processor directly sends an aggregation operation to main memory, the memory controller has to make sure that the target cache block is not stored in on-chip caches during the execution of the operation to prevent the host processor from accessing any stale copy of the data. Our architecture does not have such an issue because aggregation operations update on-chip caches first, which makes cache-resident data always up-to-date.

3.4. Coherence and Consistency

Since aggregated blocks are the same as ordinary blocks except for their contents, existing cache coherence mechanisms can manage coherence of aggregated blocks with no modifications. This can be shown in the following two steps:

First, if a specific cache block is accessed only by aggregation operations, any cache coherence protocol can maintain the coherence of the block. This is because aggregated blocks implement all types of cache operations supported by ordinary blocks (e.g., hit/miss, writeback, invalidation, etc.) with the same semantics, which allows the coherence protocol to treat aggregated blocks just as ordinary blocks.

Second, even if aggregation operations interact with normal loads/stores, it can still be supported by conventional cache coherence protocols without modifications. This is because such an interaction happens *inside a local cache* after the unmodified cache coherence protocol handles the coherence of the target cache block. For example, if a local core issues a normal load to fetch a cache block that is stored as an aggregated

Fig. 5. Data placement in a DIMM.

dirty block in a remote cache, our architecture handles this case in two orthogonal steps: (1) flushing the remote block and bringing it into the local cache *as if the local core accesses the block with an aggregation operation* and then (2) locally upgrading the block to an ordinary one. This requires no changes to the cache coherence protocol because (1) is covered by conventional coherence protocols (explained in the previous paragraph) and (2) does not involve coherence protocols at all.

Similarly, aggregation operations are fully compatible with existing consistency models because they are simply treated as a special type of memory write operations. The use of *unmodified* coherence and consistency mechanisms facilitates low-cost, low-effort integration of AIM (e.g., no need for costly in-memory hardware for cache coherence).

3.5. Main Memory

DRAM. We add one ACU per DRAM bank to implement in-memory aggregation. Since a bank is the smallest unit of memory-level parallelism in the DDR3 standard, this simplifies resource allocation between aggregation operations and in-memory ACUs. When the memory controller issues an aggregation command to the target DRAM bank, the bank reads the original data from the row buffer, computes the result of the aggregation command, and writes the result.

In addition, implementing in-memory aggregation based on standard memory modules requires modifications to data placement across DRAM chips in a DIMM. Without loss of generality, let us explain this issue with a 64-byte cache block stored in a DDR3 DIMM composed of eight $\times 8$ chips. In conventional architectures, each 8-byte subblock is interleaved across eight chips in a 1-byte granularity (Figure 5(b)). Under this data placement, performing aggregation for data larger than 1 byte requires chip-to-chip transfer (e.g., in-memory addition for chip-interleaved 8-byte data requires carry-bit propagation across chips), which is not supported by standard DIMMs. To avoid this, AIM reorganizes the data placement so that each 8-byte subblock is stored inside a single chip (Figure 5(c)). This can be done entirely by memory controllers, and thus, is transparent from other components of the system.

Memory Controllers. Memory controllers schedule aggregation commands in the same way as write commands. This is because the data bus direction of aggregation commands is identical to that of write commands (from memory controllers to DRAM), which is related to the latency of switching the bus polarity in the DDR3 standard (e.g., tWTR).

Depending on the scheduling policy, memory controllers may need to perform aggregation by themselves. This happens when the scheduler reorders a read command to a specific cache block *ahead of previous aggregation commands to the same block*. In that case, after the target block is read from the DRAM, all write/aggregation commands arriving before the read command should be applied to the loaded block to correctly handle data dependence. In reality, such a case occurs rarely since a cache block written back from the last-level cache has low chances to be read again in the near future.

Advantages. Performing aggregation in main memory gives three key advantages in terms of energy efficiency. First, it reduces the maximum number of main memory

accesses per aggregation from two (*read-modify-write* in conventional architectures) to one (*aggregation*). This is true even for applications with high data locality since in-memory aggregation allows our cache-conscious aggregation to avoid loading the target data from main memory.

Second, it improves DRAM row buffer locality, which is a critical factor in energy efficiency of DRAM. In conventional systems, one aggregation operation usually incurs two row activations because writeback caches delay the write of the aggregation operation until the cache block eviction, which degrades temporal locality between the read and the write of the aggregation operation. On the other hand, AIM implements in-memory aggregation as a single DRAM command, thereby guaranteeing at most one row activation per aggregation.

Third, it reduces switching activities of off-chip memory channels, which leads to lower off-chip I/O energy consumption. This is because aggregation operands tend to have smaller values (i.e., lower entropy in bits) than the original data.

Implications. Supporting in-memory aggregation has implications for some DRAM features. For example, it does not support ECC DIMMs, similar to other PIM architectures based on standard memory modules [Farmahini-Farahani et al. 2015; Seshadri et al. 2015]. Also, the maximum size of an aggregation operand is limited by the DIMM organization (e.g., up to 4 bytes with $\times 4$ DRAM, 8 bytes with $\times 8$ DRAM, etc.).

However, it should be noted that such implementation issues are due to the underlying main memory technology (i.e., DDR3) rather than our host architecture design (which is our main contribution). Thus, using AIM with other main memory technologies/organizations (e.g., 3D-stacked DRAM, in-DRAM ECC, etc.) can alleviate such issues. For example, the Hybrid Memory Cube 2.0 standard [HMC 2014] includes in-memory addition and bitwise commands, which can be seen as a concrete implementation of in-memory aggregation without sacrificing the error correction capability. In our future work, we will explore the impact of other memory technologies on AIM.

3.6. Potential Generalization Opportunities

Although we designed AIM for commutative and associative read-modify-write operations (i.e., aggregation), slight modifications to our architecture can potentially support other types of read-modify-write operations as well. Let us assume that there is a pair of operations “ $*$ ” and “ $\star$ ” that satisfies the following property for any a , b , and c (e.g., the following expression is satisfied if “ $*$ ” is division and “ $\star$ ” is multiplication):

$$a * b * c = a * (b \star c). \quad (1)$$

Then, even if “ $*$ ” does not satisfy commutativity and/or associativity, “ $*$ ” can be implemented as aggregation in our architecture by using “ $\star$ ” to merge two aggregation operands. More precisely, the following summarizes how aggregation that supports the preceding property can be implemented in cache-conscious aggregation:

- When an aggregation operation is performed on an aggregated block, the result is calculated by using “ $\star$ ” operation (instead of “ $*$ ”).
- When an aggregation operation is performed on an ordinary block or an aggregated block is upgraded to an ordinary block, the result is calculated by using “ $*$ ” operation.
- When an aggregated block is evicted, an aggregation operation of type “ $*$ ” is sent to the next level of memory.

For example, when $a = a \div b \div c$ is performed, if a is stored in main memory, we can update a by (1) *multiplying* b and c using cache-conscious aggregation and (2) dividing a by $b \times c$ using in-memory aggregation. Although our evaluations do not adopt such extension because our target workloads do not benefit from it (see Section 4 for the list of

```

1    %1 = load i64* %a, align 8
2    %2 = icmp gt i64 %1, %b
3    %3 = select i1 %2, i64 %b, i64 %1
4    store i64 %3, i64* %a, align 8

```

Fig. 6. Example LLVM language code for `imin64`.

aggregation operations used in our workloads), this could be useful for broadening the applicability of our architecture to applications that extensively use noncommutative and/or nonassociative read-modify-write operations.

4. COMPILER SUPPORT

As described in Section 3.2, our architecture and its programming model simplify the compiler development for our system. In this section, we demonstrate this advantage by actually developing a compiler for our architecture based on the LLVM compilation framework [Lattner and Adve 2004]. We focus on supporting the following four types of aggregation instructions as they are sufficient to cover our target applications (see Section 6), but other types of instructions can also be implemented as long as they satisfy the definition of aggregation operations.

- `iadd64`: 64-bit integer add;
- `imin64`: 64-bit integer min;
- `fadd32`: single-precision floating-point add; and
- `fadd64`: double-precision floating-point add.

Our compiler finds a set of instructions whose semantics match one of the aggregation instructions and *unconditionally* replaces it with the corresponding aggregation instruction. For example, if a `select` instruction (from Figure 6) satisfies the following criteria, the compiler always replaces the `select` and its associated instructions (mentioned next) with `imin64`.

- (1) One of the source operands (`%1`) of the `select` instruction is from a `load` instruction (line 1).
- (2) The definition of the `select` instruction (`%3`) is used as a value to be stored in a `store` instruction (line 4).
- (3) The `load` and the `store` instructions trivially have the same target address aligned to 8 bytes (`%a`).
- (4) The condition variable (`%2`) of the `select` instruction is from an `icmp` instruction (line 2) that has the same source operands (`%1` and `%b`) as those of the `select` instruction.
- (5) The condition code (`gt`) of the `icmp` instruction is set in a way that the `select` instruction chooses the smaller value between the two source operands.
- (6) All uses of intermediate definitions (`%1`, `%2`, and `%3`) are restricted to the instructions mentioned previously.
- (7) All instructions belong to the same basic block and there are no other memory write instructions between the `load` and the `store` instructions.

Through our evaluations, we will show that this simple approach *without compile-time data locality estimation* (which could be tricky or even impractical) works nicely due to our cache-conscious aggregation, thereby simplifying the toolchain development for PIM systems.

5. CONTRIBUTIONS OVER PRIOR ART

In this section, we summarize the contributions of this work over the state of the art.

Fig. 7. Exploiting data locality in (a) PEI and (b) AIM.

5.1. PIM-Enabled Instructions

The following compares AIM with the state-of-the-art PIM architecture called PIM-Enabled Instructions (PEI) [Ahn et al. 2015b], which proposes an instruction-style programming model and locality-aware PIM execution.

Programming Model. Compared to PEI, AIM provides a much more intuitive programming model, which enables *zero modifications to existing software*. First, contrary to PEI which needs explicit synchronization between PIM instructions and normal instructions by issuing pfence, cache-conscious aggregation of AIM eliminates such a requirement. Second, PEI exposes separate operand storage (called operand buffer) to software, whereas AIM directly uses host processor registers as operands of aggregation instructions. Consequently, our aggregation instructions serve as *drop-in* replacements for the equivalent normal instructions, which simplifies the compiler design (e.g., no need for compilers to identify the places to insert pfence). Although AIM does not currently support nonaggregation operations, it can be used together with PEI to make AIM-compatible instructions more energy efficient while supporting other PIM-enabled instructions as well.

In addition, AIM provides a fully automated compilation flow, whereas PEI is evaluated with handwritten code only.

Locality-Aware Execution. While PEI offloads PIM operations to either the host processor or main memory, AIM is able to execute aggregation operations at any level of the memory hierarchy, thereby enabling better adaptation to data locality. For instance, if multiple aggregation operations access the same cache block (see Figure 7), AIM accumulates their operands inside the block and sends it to main memory with in-memory aggregation, whereas PEI lets the host processor execute all of them after fetching the block from main memory and then writes the result back to main memory (thereby generating more main memory accesses than AIM).

Moreover, PEI has higher chances to mispredict data locality than AIM due to its separate locality monitor structure. For example, since the locality monitor can check locality only *after* a PEI is executed, the first PEI to each cache block is always offloaded to main memory regardless of the locality of the block. Note that, unlike our cache-conscious aggregation, the host-/memory-side PEI execution mechanism inevitably requires a separate structure for locality monitoring.

Coherence Support. Unlike PEI which needs to send back-invalidation (or back-writeback) to the last-level cache for coherence management, AIM supports cache coherence without such requests as cache-resident data is always up to date.

Main Memory. While PEI is evaluated based on HMCs, AIM uses the widespread DDR3 memory modules by default. Although both PEI and AIM do not rely on a particular memory technology, implementing a locality-aware PIM system with DDR3 main memory is more challenging because the lack of support for fine-grained (e.g., 8-byte) memory accesses increases the memory bandwidth cost of locality misprediction.

(e.g., using PIM for high-locality data). This requires more accurate locality adaptation, which is why AIM is necessary.

Energy Efficiency. Our work finds that processing aggregation inside the memory hierarchy reduces row activations, main memory accesses, and off-chip channel bit-flips, thereby improving the energy efficiency of the memory hierarchy. This has not been evaluated before.

5.2. Parallel Reduction in Caches

The concept of merging aggregation in caches has also been used to improve the scalability of parallel reduction. Although such a benefit is completely orthogonal to our work, we compare AIM against two parallel in-cache reduction mechanisms: PCLR [Garzarán et al. 2001] and Coup [Zhang et al. 2015]. Both techniques alleviate communication overheads of reduction by letting each core locally perform reduction on its private cache blocks, but unlike our work, they do not explore the energy perspective of reduction.

Energy Efficiency. Since the purpose of PCLR and Coup is to reduce the cost of parallel updates in multiprocessor systems, both of them merge reduction operands with their original data *in the closest shared memory to the processors*, that is, shared caches in modern computer systems. Thus, when the target cache block of reduction is not present in shared caches, they have to load the original data from main memory as in conventional architectures. Considering that our in-memory aggregation reduces main memory accesses, row activations, and off-chip channel bit-flips by *avoiding loading the original data of aggregation from main memory*, PCLR and Coup cannot improve the energy efficiency of main memory unlike AIM (see Section 8.1 for a quantitative evaluation of this aspect). Note that extending PCLR or Coup to support in-memory aggregation is not trivial as the shared cache should be able to distinguish reduction data from normal data and propagate reduction operands from shared caches to main memory, which requires our cache-conscious aggregation.

Applicability. PCLR and Coup are effective only if (1) multiple cores competitively access the reduction data and (2) the reduction data fits in the shared last-level cache (since they have to fetch the original data on shared cache misses). On the contrary, AIM becomes more energy efficient with a larger volume of data and does not depend on whether the data is shared or not. Hence, AIM has broader use cases than PCLR and Coup, considering that the amount of data handled by emerging data-intensive workloads is already far beyond the last-level cache capacity and is still explosively increasing.

Effort for Integration. Compared to AIM, PCLR and Coup require more effort to be integrated into existing systems. First, PCLR is not general enough to be used beyond parallel reduction (e.g., unable to share data between reduction and normal instructions, manual cache block flushes required at the end of reduction loops, etc.). Second, Coup introduces significant changes to the cache coherence protocol, which increases the design cost of verifying the coherence protocol and, particularly, its implementation. Third, PCLR and Coup assume manual modification of existing software, whereas AIM eliminates such an obstacle by providing a fully automated compiler.

5.3. Other Related Work

Processing-in-Memory. The PIM concept was extensively studied in the 1990s [Kogge 1994; Oskin et al. 1998; Patterson et al. 1997; Elliott et al. 1999; Hall et al. 1999; Kang et al. 1999; Draper et al. 2002; Kim et al. 2002; Sterling and Zima 2002], which was, however, not commercialized due to the lack of manufacturing technologies for cost-effective integration of logic and memory. Nowadays, this challenge can be practically

solved by 3D integration technologies, which has motivated the recent development of new PIM architectures based on 3D stacking [Zhu et al. 2013; Pugsley et al. 2014; Zhang et al. 2014; Nair et al. 2015; Farmahini-Farahani et al. 2015; Ahn et al. 2015a, 2015b].

The remaining challenge toward commercial PIM systems is on *the ease of programmability*. Most of the proposals, including recent ones, require significant efforts to utilize their PIM systems due to the unconventional programming models of in-memory accelerators, the lack of interoperability with cache coherence and virtual memory, and insufficient support for coordination between host-side and memory-side execution. AIM addresses these problems by interfacing simple aggregation operations as host processor instructions and developing the cache-conscious aggregation mechanism.

Row Buffer Locality of DRAM Writes. Since conventional writeback caches degrade the row buffer hit ratio of DRAM writes, there have been many techniques to improve the row buffer locality of DRAM writes by issuing last-level cache writebacks earlier than cache block evictions [Lee et al. 2010; Stuecheli et al. 2010; Seshadri et al. 2014; Volos et al. 2014; Wang et al. 2012]. Compared to such techniques, AIM offers the following two benefits. First, in-memory aggregation *guarantees* the write of an aggregation operation to hit in the row buffer. Second, AIM does not generate extra writebacks since it *delays the read* of an aggregation operation until the corresponding write, instead of proactively issuing extra writebacks. We will quantitatively analyze these two aspects in Section 8.3.

6. TARGET APPLICATIONS

In this section, we describe our target multithreaded workloads from both standard benchmarks and emerging data-intensive applications. These workloads are important in that they are often performance bottlenecks of many important applications (e.g., graph algorithms in social network analysis, backpropagation in deep learning, sparse matrix-vector multiplication in scientific computing, etc.). All workloads are compiled by our compiler with no software modifications.

Average Teenage Follower (AT) [Hong et al. 2014] is an example kernel of social network analysis. For each teenager vertex, the follower counts of its successors are incremented by using `iadd64`, which calculates the number of teenage followers.

Backpropagation (BP) [Che et al. 2009] is a widely used algorithm for training neural networks. AIM uses `fadd32` to subtract a ratio of gradients from the weights (i.e., backward propagation of the error between the prediction and the expected outcome).

Breadth-First Search (BF) [Hong et al. 2014] is a graph traversal algorithm. We evaluate the level-synchronous BFS [Hong et al. 2011, 2014], in which each vertex is equipped with a “level” field to indicate the breadth of the vertex. For each iteration, vertices in the current level update the levels of their successors with the `min` function (`imin64`) to ignore already visited vertices.

Histogram (HG) [Ahn et al. 2005] calculates the distribution of data by counting items that are mapped to each bin. AIM increments the bin counter for each data item by using `iadd64`.

HotSpot (HS) [Che et al. 2009] performs 2D transient thermal simulation of VLSI systems. For each iteration, the temperature of each grid is adjusted by estimating the effect of adjacent grids. For this computation, AIM uses `fadd64` to add the contribution from nearby grids to the temperature of the current grid.

PageRank (PR) [Brin and Page 1998; Malewicz et al. 2010; Hong et al. 2012, 2014] computes the importance of each vertex in a graph from the relationship between vertices. AIM uses `fadd64` for updating the rank of successor vertices.

Table I. Baseline Simulation Configuration

Component	Configuration
Core	16 out-of-order cores, 4GHz, 4-issue, 128-entry instruction window
L1 I/D-Cache	Private, 32KB, 4/8-way, 64B blocks, 16 MSHRs
L2 Cache	Private, 256KB, 8-way, 64B blocks, 16 MSHRs
L3 Cache	Shared, 16MB, 16-way, 64B blocks, 64 MSHRs
DRAM Controller	32-entry read/write queue, FR-FCFS scheduling, drain-when-full write buffer policy Lee et al. 2010
DRAM	16GB, DDR3-1600 11-11-11 Micron Technology 2009, four channels, one rank/channel, eight banks/rank, 8KB rows

RabbitCT (RC) [Rohkohl et al. 2009] provides a multicore implementation of the FDK algorithm [Feldkamp et al. 1984], which performs backprojection for CT (Computed Tomography) image reconstruction. In this workload, AIM uses `fadd32` for overlaying 2D measurements into the corresponding 3D space to reconstruct a 3D image.

Sparse Matrix-Vector Multiplication (SM) [Sadd 2003] multiplies a sparse matrix A and a vector x , which is an important kernel of many scientific applications. It uses the Compressed Sparse Column (CSC) format, which generates random memory additions (`fadd64` in AIM) to the output vector.

Single-Source Shortest Path (SP) [Malewicz et al. 2010; Hong et al. 2014] is the parallel Bellman-Ford algorithm for finding the shortest paths from a source vertex to all other vertices in a graph. In this algorithm, AIM uses `imin64` for edge relaxation, in which a vertex u updates the distance d_v of its successor v to $\min(d_v, d_u + w_{uv})$.

Weakly-Connected Components (WC) [Kang et al. 2009] is the HCC algorithm [Kang et al. 2009] for finding weakly connected components in a graph. It initializes each vertex with a unique integer label and, for each iteration, collapses the labels of adjacent vertices into the smallest one among them by using `imin64`.

7. EVALUATION METHODOLOGY

7.1. Simulation Configuration

We evaluate our architecture based on our $\times 86-64$ simulator whose frontend is Pin [Luk et al. 2005]. Our simulator has cycle-level timing models of out-of-order cores considering register/structural dependency and limited instruction window, three-level inclusive multibank caches with Miss Status Holding Registers (MSHRs), the MESI cache coherence protocol, on-chip crossbar networks, multichannel memory controllers, and DDR3-based main memory. The simulation configuration of the baseline system is summarized in Table I. We assume that the host processor of the baseline also implements normal instruction versions of `iadd64`, `imin64`, `fadd32`, and `fadd64` so that all systems execute the same number of instructions for the same amount of work.

For energy evaluation, we use Micron's DDR3 SDRAM System Power Calculator [Micron Technology 2007, 2009] to estimate the energy consumption of DRAM and utilize CACTI-3DD [Chen et al. 2012] to further break down the energy inside a DRAM chip, similar to previous work [Zhang et al. 2014]. Also, the energy consumption of on-chip caches and memory controllers is modeled by CACTI 6.5 [Muralimanohar et al. 2009] and McPAT 1.2 [Li et al. 2009], respectively.

For in-memory aggregation, each DRAM bank incorporates one 64-bit ACU, which includes one 64-bit integer adder (`iadd64`), one 64-bit integer comparator/multiplexer (`imin64`), two single-precision floating-point adders (`fadd32`), and one double-precision floating-point adder (`fadd64`). In-memory ACUs are assumed to have 5ns of delay, or one-cycle delay at the DRAM core clock, with no pipelining.

Table II. Input Sets of Workloads

Application	Small Input	Large Input
AT, BF, PR, SP, and WC	soc-Slashdot0811 [Leskovec and Krevl 2014]	soc-LiveJournal1 [Leskovec and Krevl 2014]
BP	$65536 \times 16 \times 1$ [Che et al. 2009]	$4096 \times 4096 \times 1000$ [Krizhevsky et al. 2012]
HG	1,024 bins	4,194,304 bins
HS	256×256 grids	$1,024 \times 1,024$ grids
RC	256^3 voxels [Rohkohl et al. 2009]	$1,024^3$ voxels [Rohkohl et al. 2009]
SM	AMD/G2.circuit [Davis and Hu 2011]	Freescape/FullChip [Davis and Hu 2011]

Also, on-chip caches are equipped with eight-way 64-bit ACUs to compute the result of aggregation for a 64-byte cache block. In our configuration, each L1 cache bank has one ACU, while eight/four L2/L3 cache banks share one ACU.⁴ We design the on-chip ACU to have one-/five-cycle delay at 4GHz with no pipelining for an integer/floating-point operation.

7.2. Hardware Overhead

The following explains the energy/area overheads of hardware modifications introduced by AIM and our methodology to obtain such estimates. The experimental results shown in this article take these energy overheads into account.

In-Memory ACUs. To estimate the energy/area overheads of in-memory ACUs, we synthesize our Verilog implementation of the ACU design by using Synopsys Design Compiler. Since proprietary DRAM processes are not available due to confidentiality, we conduct conservative estimation by using the TSMC 130nm technology, considering that DRAM processes fall behind logic processes by multiple generations in implementing logic circuits [Zhang et al. 2014]. According to the synthesis result, eight 64-bit ACUs (one per bank) take only $0.42mm^2$ of die area, which is negligible compared to large modern DRAM die sizes (e.g., $30.9mm^2$ for a 23-nm 4-Gb DRAM die [Lim et al. 2012]). Due to this wide area gap between ACUs and DRAM dies, although the exact amount of discrepancy between DRAM processes and logic processes can vary, ACUs implemented by DRAM processes are still expected to have a small area overhead. If such extra cost is unaffordable, in-memory ACUs can also be implemented on the existing logic die of 3D-stacked DRAM (e.g., in-memory atomics of HMCs [HMC 2014]) as our mechanism is not limited to a particular main memory technology.

On-Chip ACUs. To estimate the energy/area overheads of on-chip ACUs, we synthesize the ACU design with our timing constraint by using the TSMC 45-nm technology library and compare the result against the cache area modeled by CACTI. The area overhead of on-chip ACUs is 3% of the on-chip cache hierarchy area.

Aggregation Type Fields in Cache Tags. AIM adds only three extra bits per tag across all caches to store one of the five possible aggregation types (i.e., ordinary, iadd64, imin64, fadd32, and fadd64). This introduces negligible area overheads to on-chip caches (e.g., less than 1% of storage overhead).

7.3. Workloads

We evaluate 10 data-intensive applications explained in Section 6. Each workload has two input sets with different sizes (shown in Table II) to demonstrate the impact of the working set size. All workloads are simulated for up to one billion instructions after

⁴We performed a parameter sweep of the number of ACUs per cache and found that putting one dedicated ACU to every L2/L3 cache bank shows less than 1% energy reduction compared to our shared ACU configuration.

Fig. 8. Comparison of the energy consumption of DRAM (top), the energy consumption of on-chip caches and memory controllers (middle), and the system performance (bottom).

skipping initialization phases. In these workloads, aggregation instructions account for 4% of the dynamic instruction count on average; however, they are responsible for 50% of the total DRAM accesses in large-input workloads. Since our target applications are mostly memory-bandwidth bound, efficient aggregation execution is very important in our workloads.

8. EVALUATION RESULTS

8.1. Energy Consumption and Performance

Figure 8 compares the energy consumption and system performance of the following five configurations. All results are normalized to the baseline and the rightmost sets of bars labeled as “AVG” indicate the geometric mean of the results.

- Baseline* is the traditional system shown in Table I.
- PIM-Only* executes all aggregation operations in main memory after flushing the target block from on-chip caches (no cache-conscious aggregation).
- PEI* represents PIM-enabled instructions [Ahn et al. 2015b] (see Section 5.1). Contrary to the original paper whose evaluation is based on HMC, we use the main memory architecture of AIM (based on DDR3) as in-memory PIM operation implementation for a fair comparison.
- AIM-Private* enables cache-conscious aggregation only for private caches (i.e., L1/L2 caches). Thus, when an aggregation operation causes an L3 cache miss, the L3 cache always fetches the target data from main memory.
- AIM* is the proposed architecture supporting both cache-conscious aggregation and in-memory aggregation.

AIM vs. Baseline/PIM-Only. From Figure 8, we draw three conclusions. First, AIM reduces the energy consumption of main memory by 15%/28% in small-/large-input workloads (see Section 8.2 for detailed analysis). Note that simply using in-memory aggregation without proper consideration of data locality (i.e., PIM-Only) *increases* the average main memory energy consumption by $5.4\times$ in small-input workloads. This is because PIM-Only does not utilize on-chip caches for aggregation operations, and thus, generates $8.8\times$ as many main memory accesses as the baseline does in small-input workloads. The same holds even for some large-input workloads with good data locality (e.g., HS, RC, and SM).

Second, AIM consumes 13%/19% less on-chip memory hierarchy energy in small-/large-input workloads, respectively. This comes mostly from the performance improvement of AIM (see the next paragraph). Although our cache-conscious aggregation may perform more computation than the baseline (e.g., upgrades from aggregated blocks to ordinary ones), its impact is negligible as discussed in footnote 3.

Third, AIM also improves average system performance by 21%/29% for small-/large-input workloads due to two major factors. First, fewer main memory accesses in AIM alleviate the off-chip memory bandwidth bottleneck. Second, AIM can move main memory accesses for aggregation operations out of the critical path of program execution. In conventional architectures, when the target data of aggregation is stored in main memory, the host processor has to wait until the data is loaded from the main memory. On the other hand, AIM simply initializes the L1 cache block with identity elements and executes the aggregation operation on it. This shortens the processor stall time caused by aggregation operations.

AIM vs. PEI. We also compare AIM with PEI [Ahn et al. 2015b]. In small-input workloads, AIM achieves 19% lower main memory energy consumption and 31% higher performance than PEI. This is because (1) AIM is able to utilize PIM even under high data locality unlike PEI and (2) the locality prediction of PEI is less accurate than AIM (see Section 5.1). Although the original paper [Ahn et al. 2015b] showed that PEI matches the performance and energy consumption of host-side execution in small-input workloads, this does not happen under our configuration since our DDR3-based system imposes higher memory bandwidth overheads to locality misprediction than the original HMC-based system (discussed in Section 5.1).

In large-input workloads, AIM reduces the main memory energy consumption by 21% and improves the performance by 15% compared to PEI. Ideally, PEI and AIM should perform almost identically under zero data locality as both will always utilize PIM. However, in reality, even large-input workloads have some degree of locality, which makes AIM outperform PEI as in small-input workloads. This is especially noticeable in HG, RC, and SM because they exhibit higher data locality (i.e., 14% higher average cache hit ratio in the baseline) than the rest of the large-input workloads.

AIM vs. AIM-Private. Lastly, we evaluate AIM against AIM-Private. Since AIM-Private merges aggregation operands with their original data *in the shared last-level cache* (rather than DRAM as in AIM) just as Coup [Zhang et al. 2015] does, comparing AIM against AIM-Private shows the advantages of our cache-conscious aggregation over parallel in-cache reduction techniques such as PCLR and Coup (see Section 5.2). Note that, although Coup improves the scalability of parallel reduction unlike AIM-Private, it is orthogonal to the benefit of AIM.

According to the experimental results, AIM-Private is not effective at all when the working set size exceeds the cache capacity. In large-input workloads, AIM-Private shows almost the same DRAM energy consumption as the baseline, whereas AIM achieves 28% reduction (they show comparable energy efficiency in small-input workloads). This is because AIM-Private issues plain DRAM reads/writes to handle

Fig. 9. DRAM dynamic energy breakdown.

aggregation operations for memory-resident data just as conventional architectures do. From this result, we can conclude that (1) the energy efficiency benefit of parallel in-cache reduction techniques is *not scalable*, in that they are limited to the case when the working set size does not exceed the last-level cache capacity,⁵ and (2) our new architectural design overcomes this limitation by exposing aggregation to all levels of the memory hierarchy, which we call a *processing-in-memory-hierarchy* paradigm.

8.2. Dynamic Energy Breakdown

Figure 9 shows DRAM dynamic energy breakdown in large-input workloads. We omit background energy of DRAM (39% of the DRAM energy in the baseline) as it is simply proportional to execution time. Also, results for small-input workloads are not shown here because, under high data locality, on-chip caches coalesce most of the aggregation operations, and thus, DRAM energy analysis in such cases provides little insight into the benefit of in-memory aggregation.

Most noticeably, AIM reduces the average energy consumption of row activations by 50%. This is contributed by the fact that AIM guarantees at most one row activation per aggregation operation, contrary to the baseline where one aggregation operation incurs up to two row activations. Due to this, AIM achieves a 30% higher average row buffer hit ratio than the baseline, which indicates fewer row activations.

Moreover, AIM saves 50% of the off-chip I/O energy consumption on average due to two reasons. First, it eliminates 36% of main memory accesses by reducing the maximum number of main memory accesses per aggregation operation from two (read and write) to one. Second, it reduces off-chip channel bit-flips per transfer by 48% since aggregation operands usually have smaller values than the original data.

Lastly, AIM reduces the read/write energy consumption by 26% on average because one aggregation command consumes less energy than a combination of one read command followed by one write command (i.e., 33% less energy according to our model based on CACTI-3DD [Chen et al. 2012]). The reason for this is that (1) the data read by an aggregation command does not need to be transferred outside the bank and (2) in-memory aggregation sends one less command to the bank.

All these benefits are achieved with very low implementation overheads. On average, in-memory ACUs contribute only 1.7% of the DRAM energy consumption.

8.3. Comparison with Aggressive Writeback

As explained in Section 5.3, there have been several techniques that improve the row buffer locality of conventional architectures by proactively issuing bulk writebacks

⁵This is an important limitation because the working set size of our target workloads (and many other big-data workloads) easily exceeds the last-level cache capacity.

Fig. 10. Effect of aggressive writeback on DRAM energy (top) and system performance (bottom).

to cache blocks in the same DRAM row. However, as shown in Figure 10, even after the baseline adopts Aggressive Writeback (AWB) [Seshadri et al. 2014], AIM provides higher energy efficiency than the baseline. This is because, although AWB does shorten the distance between the read and the write of each aggregation operation to some extent, it cannot schedule *every* write of an aggregation operation *right after* the corresponding read. On the other hand, AIM *guarantees* row buffer hits on the writes of aggregation operations, thereby achieving a 13% higher row buffer hit ratio than the baseline with AWB.

Also, AWB sometimes degrades system performance due to two reasons. First, proactively issuing writebacks incurs up to 19% of extra DRAM writes, which is harmful to applications with high memory bandwidth consumption. Second, extra writeback requests increase last-level cache contention, which may block latency-critical reads/writes from the host processor. AIM is free from both drawbacks (e.g., up to 2% increase in DRAM writes), and thus, it does not degrade the performance across all applications evaluated in this article.

8.4. Multiprogrammed Workloads

In order to show the robustness of our cache-conscious aggregation against varying data locality, we evaluate our architecture by using 100 multiprogrammed workloads. Each workload consists of two applications that are randomly selected from our target applications. All results are sorted by the energy consumption of PIM-Only. We omit the normalized energy consumption higher than two for better visibility.

As shown in Figure 11, AIM consistently outperforms *both* the baseline and PIM-Only in terms of energy efficiency and performance. The results also confirm that always using in-memory aggregation is harmful to both energy consumption (up to 10 $\times$, not shown in the figure) and performance (up to 91% slowdown). We conclude that the adaptivity of our cache-conscious aggregation is robust enough to be used in real-world situations where a single machine dynamically services multiple workloads with diverse data locality.

8.5. Comparison with Intrinsic-Based Code

To demonstrate the quality of our compiler, we compare the energy consumption and performance of the compiler-generated binaries (shown in Figure 8) with those of

Fig. 11. DRAM energy consumption (top) and performance (bottom) in multiprogrammed workloads.

intrinsic-based, handwritten code where aggregation instructions are manually inserted to software. According to our analysis, our compiler-based approach identifies all opportunities to use aggregation instructions in our workloads with no programmer intervention and shows only 0.3% higher DRAM energy consumption and 0.6% lower performance than the intrinsic-based approach on average.

9. CONCLUSION

We proposed AIM, a new style of PIM systems designed from the ground up for energy efficiency. AIM features two key contributions that realize processing-in-memory-hierarchy: (1) in-memory aggregation, which tackles the type of computation that is difficult for the traditional memory hierarchy to handle in an energy-efficient way, and (2) cache-conscious aggregation, which leverages on-chip caches for dynamically coalescing aggregation operations with high data locality before they are sent to main memory. These new ideas are seamlessly integrated into existing systems with an intuitive programming model and hassle-free compiler support. Our extensive evaluations show that AIM greatly improves both energy efficiency and system performance by reducing main memory accesses, improving DRAM row buffer locality, and reducing switching activities of memory channels. We conclude that AIM paves the way for introducing the PIM concept into existing systems in the most energy-efficient and least disruptive way possible.

REFERENCES

- Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoun Choi. 2015a. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the International Symposium on Computer Architecture*. 105–117.
- Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoun Choi. 2015b. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the International Symposium on Computer Architecture*. 336–348.
- Jung Ho Ahn, Mattan Erez, and William J. Dally. 2005. Scatter-add in data parallel architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 132–142.
- Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro* 34, 4 (July 2014), 36–42.

- Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the International World-Wide Web Conference*. 14–18.
- Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 44–54.
- Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Proceedings of the Design, Automation and Test in Europe*. 33–38.
- Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (Dec. 2011), 1:1–1:25.
- Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the International Conference on Supercomputing*. 14–25.
- Duncan G. Elliott, Michael Stumm, W. Martin. Snelgrove, Christian Cojocar, and Robert McKenzie. 1999. Computational RAM: Implementing processors in memory. *IEEE Design Test Comput.* 16, 1 (Jan. 1999), 32–41.
- Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 283–295.
- L. A. Feldkamp, L. C. Davis, and J. W. Kress. 1984. Practical cone-beam algorithm. *J. Opt. Soc. Am. A* 1, 6 (June 1984), 612–619.
- María Jesús Garzarán, Milos Prvulovic, Ye Zhang, Alin Julia, Hao Yu, Lawrence Rauchwerger, and Josep Torrellas. 2001. Architectural support for parallel reductions in scalable shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 243–254.
- Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. 1999. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 57.
- Hybrid Memory Cube Consortium. 2014. *Hybrid Memory Cube Specification 2.0*.
- Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 349–362.
- Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 78–88.
- Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. 2014. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of the International Symposium on Code Generation and Optimization*. 208–218.
- U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proceedings of the International Conference on Data Mining*. 229–238.
- Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. 1999. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the International Conference on Computer Design*. 192–201.
- Daehyun Kim, Mainak Chaudhuri, and Mark Heinrich. 2002. Leveraging cache coherence in active memory systems. In *Proceedings of the International Conference on Supercomputing*. 2–13.
- Peter M. Kogge. 1994. EXECUBE—A new architecture for scaleable MPPs. In *Proceedings of the International Conference on Parallel Processing*. 77–84.
- Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the Neural Information Processing Systems*. 1097–1105.
- Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. 75–86.
- Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2010. *DRAM-aware Last-level Cache Writeback: Reducing Write-caused Interference in Memory Systems*. Technical Report No. TR-HPS-2010-002. The University of Texas at Austin.

- Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. (June 2014). <http://snap.stanford.edu/data>.
- Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture*. 469–480.
- Kyu-Nam Lim, Woong-Ju Jang, Hyung-Sik Won, Kang-Yeol Lee, Hyungsoo Kim, Dong-Whee Kim, Mi-Hyun Cho, Seung-Lo Kim, Jong-Ho Kang, Keun-Woo Park, and Byung-Tae Jeong. 2012. A 1.2V 23nm 6F² 4Gb DDR3 SDRAM with local-bitline sense amplifier, hybrid LIO sense amplifier and dummy-less array architecture. In *International Solid-State Circuits Conference Digest of Technical Papers*. 42–44.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*. 190–200.
- Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data*. 135–146.
- Micron Technology 2007. *Calculating Memory System Power for DDR3*. Micron Technology.
- Micron Technology 2009. *4Gb: $\times 4$, $\times 8$, $\times 16$ DDR3 SDRAM*. Micron Technology.
- Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report No. HPL-2009-85. HP Laboratories.
- Ravi Nair, Samuel F. Antao, Carlo Bertolli, Pradip Bose, Jose R. Brunheroto, Tong Chen, Chen-Yong Cher, Carlos H. A. Costa, Jun Doi, Constantinos Evangelinos, Bruce M. Fleischer, Thomas W. Fox, Diego Sanchez Gallo, Leopold Grinberg, John A. Gunnels, Arpith C. Jacob, Philip Jacob, Hans M. Jacobson, Tejas Karkhanis, C. Kim, Jaime H. Moreno, J. Kevin O'Brien, Martin Ohmacht, Yoonho Park, Daniel A. Prener, Bryan S. Rosenberg, Kyung Dong Ryu, Olivier Sallenave, Mauricio J. Serrano, Patrick D. M. Siegl, Krishnan Sugavanam, and Zehra Sura. 2015. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM J. Res. Dev.* 59, 2/3 (March 2015), 17:1–17:14.
- Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active pages: A computation model for intelligent memory. In *Proceedings of the International Symposium on Computer Architecture*. 192–203.
- David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (March 1997), 34–44.
- Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. 190–200.
- Christopher Rohkohl, Benjamin Keck, Hannes Hofmann, and Joachim Hornegger. 2009. RabbitCT—An open platform for benchmarking 3D cone-beam reconstruction algorithms. *Med. Phys.* 36, 9 (Sept. 2009), 3940–3944.
- Yousef Sadd. 2003. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics.
- Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2014. The dirty-block index. In *Proceeding of the International Symposium on Computer Architecture*. 157–168.
- Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2015. Fast bulk bitwise AND and OR in DRAM. *IEEE Comput. Architect. Lett.* 14, 2 (July-Dec. 2015), 127–131.
- Thomas L. Sterling and Hans P. Zima. 2002. Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 48.
- Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The virtual write queue: Coordinating DRAM and last-level cache policies. In *Proceedings of the International Symposium on Computer Architecture*. 72–82.
- Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. 2014. BuMP: Bulk memory access prediction and streaming. In *Proceedings of the International Symposium on Microarchitecture*. 545–557.
- Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. 2012. Improving writeback efficiency with decoupled last-write prediction. In *Proceedings of the International Symposium on Computer Architecture*. 309–320.
- Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*. 85–98.

- Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the International Symposium on Microarchitecture*.
- Tao Zhang, Ke Chen, Cong Xu, Guangyu Sun, Tao Wang, and Yuan Xie. 2014. Half-DRAM: A high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation. In *Proceedings of the International Symposium on Computer Architecture*. 349–360.
- Qiuling Zhu, Berkin Akin, H. Ekin Sumbul, Fazle Sadi, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2013. A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *Proceedings of the International 3D Systems Integration Conference*. 1–7.

Received May 2016; revised July 2016; accepted August 2016