On Consistency for Bulk-Bitwise Processing-in-Memory

Ben Perach Ronny Ronen Shahar Kvatinsky
The Andrew and Erna Viterbi Faculty of Electrical & Computer Engineering
Technion – Israel Institute of Technology
Haifa, Israel
benperach@campus.technion.ac.il ronny.ronen@ef.technion.ac.il shahar@ee.technion.ac.il

***Abstract*—Processing-in-memory (PIM) architectures allow software to explicitly initiate computation in the memory. This effectively makes PIM operations a new class of memory operations, alongside standard memory operations (*e.g.*, load, store). For software correctness, it is crucial to have ordering rules for a PIM operation with other PIM operations and other memory operations, *i.e.*, a consistency model that takes into account PIM operations is vital. To the best of our knowledge, little attention to PIM operation consistency has been given in existing works. In this paper, we focus on a specific PIM approach, named bulk-bitwise PIM. In bulk-bitwise PIM, large bitwise operations are performed directly and stored in the memory array. We show that previous solutions for the related topic of maintaining coherency of bulk-bitwise PIM have broken the host native consistency model and prevent any guaranteed correctness. As a solution, we propose and evaluate four consistency models for bulk-bitwise PIM, from strict to relaxed. Our designs also preserve coherency between PIM and the host processor. Evaluating the proposed designs' performance with a gem5 simulation, using the YCSB short-range scan benchmark and TPC-H queries, shows that the run time overhead of guaranteeing correctness is at most 6%, and in many cases the run time is even improved. The hardware overhead of our design is less than 0.22%.**

I. Introduction

In recent processing-in-memory (PIM) architectures, where the memory module does not only hold data but also processes it, PIM operations are explicitly initiated by software (*e.g.*, by a dedicated instruction in the host processor [1,5,9,21,25,29]). When initiated, these PIM operations are sent from the host core to the memory, thereby becoming a new class of memory operations alongside standard memory operations such as load and store. These new operations require re-investigation of the consistency model for hosts using PIM operations. Questions about reordering of PIM operations between PIM and other memory operations must be raised. These issues are important, as without clear ordering rules, it is hard to reason about program correctness [22]. Despite its importance, the issue of a consistency model for PIM has been completely ignored in PIM architecture works [1, 5, 8, 9, 12, 25, 27, 29, 31, 33].

PIM techniques can be categorized according to the location of the processing units, the granularity of the PIM operations, and the arbitration of the PIM computations and memory

This work was supported by the European Research Council through the European Union's Horizon 2020 Research and Innovation Programme under Grant 757259 and through the European Union's Horizon Europe Research and Innovation Programme under Grant 101069336.

Example Code:
Write(A) // *step* ❶
MemFence
Write(B) // *step* ❷
MemFence
Flush(A) // *step* ❸
Flush(B) // *step* ❹
MemFence
PIM op // *step* ❻

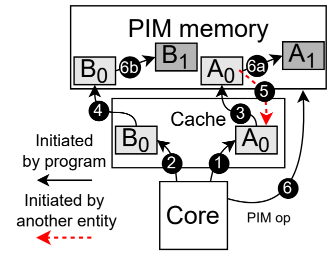

Fig. 1: Example of a code using a PIM operation with a possible reordering of memory operations, resulting in a cyclic order; see the example described in Section I. Execution steps are marked as comments in the code and dark circles. *MemFence* is a standard memory fence instruction, indicating strict order between operations before and after the fence [21,22]. The cyclic order in the example breaks the *MemFence* order guarantees.

accesses [21,28]. Each PIM category exhibits different characteristics, suits different application classes, and requires different system-level support. In this paper, we focus specifically on *bulk-bitwise PIM* [9, 18, 25, 27, 29, 31, 33] and discuss the consistency of such PIM techniques.

In bulk-bitwise PIM, the memory array serves both as the storage medium and as the processing unit, performing the computation on its stored data and writing the result directly to the memory cells within the array. The bulk-bitwise operations performed within the array are bitwise row-to-row or column-to-column logic operations, potentially performed on numerous memory arrays simultaneously. These characteristics have two important implications on the computing system: a) bulk-bitwise PIM operations are immediately committed to the system state during PIM execution, and b) a single bulk-bitwise PIM operation may change a large memory section. Regarding the consistency model and its implementation, these implications suggest that bulk-bitwise PIM operations cannot be speculatively executed and reverted on order violation and that the ordering of PIM operations needs to be enforced on a large address range. These implications do not necessarily apply to other PIM techniques [1, 5, 8, 21, 28], which might require different consistency models and different solutions.

Some prior works refer to the related topic of coherency between the PIM module and the host caches [9, 25, 29]. Consistency and coherency are tightly coupled when PIM is considered, as loads and stores are handled in the host processor caches while PIM operations go to the memory, potentially reordering PIM operations with loads and stores. Designing coherency solutions for PIM should be done with care, as flawed ones can result in problematic behavior. For instance, in [9, 25], PIM operations do not interact with the host processor caches and the coherency between the PIM module and caches is the responsibility of the software (by using cache flushes). For such a PIM operation mechanism, consider the scenario depicted in Fig. 1. A thread, running the example code in Fig. 1, writes value A_0 to address A (Fig. 1❶) and then writes value B_0 to address B (Fig. 1❷). Later, both addresses are flushed to memory (Fig. 1❸❹) before issuing a PIM operation (Fig. 1❻) that will change their values to A_1 and B_1 (Fig. 1❻a❻b). Between the flush and the PIM operation, however, A is read from the memory to a cache (Fig. 1❺) with the value A_0 (*e.g.*, by another thread or by a prefetcher). As a result, read operations to A after the PIM operation will result in a cache hit and the old value of A_0. Although the software is written properly (*i.e.*, all related addresses are flushed from the cache by the software), the result is still incorrect without a proper consistency model.

The above scenario is problematic for an additional reason. A **cyclic ordering** without a well-defined happen-before relation exists in this scenario, allowing the same thread to see different orders of events regardless of the host processor's native consistency and software enforcement. A cyclic ordering can be formed by the following sequence of operations: (1) reading addresses A and B to validate that $Write(A)$ happened before $Write(B)$, (2) reading B twice and getting B_0 and then B_1 to validate that $Write(B)$ is before $PIMop$, and (3) reading B, getting B_1, and then reading A, getting A_0 (from the cache) to validate that $PIMop$ is before $Write(A)$. Hence, $Write(A)$ is before $Write(B)$, $Write(B)$ is before $PIMop$, and $PIMop$ is before $Write(A)$, *i.e.*, $Write(A)$ is before itself, forming a cycle. Furthermore, the ordering of $Write(A)$ and $Write(B)$ is not well-defined, **breaking the ordering rules of the host** specified explicitly by the *MemFence* instructions [22]. Changing the host ordering rules by the additional PIM operation breaks the software correctness relying on these rules.

In the above example, the problem comes from the non-atomicity of the PIM operation and cache flushes. This non-atomicity enables relevant addresses to be brought back to the host caches, making the PIM memory non-coherent with the host caches, and violating the host processor ordering rules for loads and stores. Any ordering guarantees that include PIM operations require the atomicity of PIM operations and cache flushes on the relevant addresses.

In this paper, we propose and evaluate different options for incorporating PIM operations into a consistency model. We offer hardware solutions to enforce the proposed consistency models, including the atomicity of the PIM operations and cache flushes. Our solutions are evaluated using the gem5 simulator environment [3] and workloads for the YCSB [7] and TPC-H [32] benchmarks.

In summary, this paper makes the following contributions:

- We propose four consistency models for bulk-bitwise PIM operations and suggest how to implement them.
- We design a hardware solution to enforce the atomicity of PIM operations and their required cache flushes.
- We evaluate our solutions using a gem5 simulation and database workloads (YCSB and TPC-H), showing that the run time overhead of guaranteeing correctness is at most 6%, and in many cases the run time is even improved.
- We show that in the context of bulk-bitwise PIM, relaxed consistency models do not necessarily execute faster than strict consistency models.

II. Background

A. Bulk-Bitwise PIM

Bulk-bitwise PIM is a PIM technique characterized by the location of the PIM processing elements and their basic operation capabilities. The processing elements in bulk-bitwise PIM are the memory cells themselves and their periphery circuits (*e.g.*, voltage drivers, sense amplifiers, decoders). Several technologies have been suggested for implementing such memory arrays, including DRAM [9, 29] and emerging resistive technologies [4, 11, 16, 20, 27] (often referred to as memristive stateful logic). All such technologies execute simple logic operations (*e.g.*, AND, NOR, NOT) between one or more cells in the memory array and write the result into a memory cell, as shown in Fig. 2. These logic operations have the restriction that the input and output cells have to be on the same row or column. The same logic operation, however, can be performed in parallel on numerous cells that are aligned on the same columns or rows. Additionally, as a memory chip comprises many memory arrays, and a memory card comprises several memory chips, a memory card with such PIM capabilities can concurrently operate on many arrays, achieving substantial computational throughput, *i.e.*, bulk-bitwise operations.

To perform more complex operations (*e.g.*, addition, multiplication), the basic array logic operations are performed multiple times, during which the array cannot be accessed, and requires additional memory cells to hold intermediate values [2, 9]. Hence, performing a complex operation may implicitly change more cells than just the designated output cells. To support these complex operations at the memory array level, an additional control logic is added to generate the sequence of basic operations [2, 9, 25, 33] and manage the additional cells. Due to area and control constraints, the control logic is often shared between several arrays, and performs the same operation in parallel on all shared arrays.

Since bulk-bitwise PIM operations change memory cell values, using bulk-bitwise PIM modules as the main memory inherently changes the visible state of the system. Therefore, when a command to perform a bulk-bitwise PIM operation is sent from a host processor, the host must issue the command only when it reaches the pipeline commit stage. In that respect, bulk-bitwise PIM operations are similar to store instructions. Nevertheless, unlike a store instruction, a PIM operation is sent directly to the memory and not to the cache hierarchy.

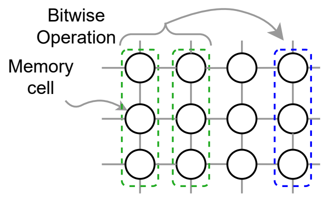

Fig. 2: Illustration of a 3×4 memory array performing a simple bitwise column logic operation (*e.g.*, NOR) per row. In each row, the operation inputs and output are the two left column cell values and the right column cell, marked by green and blue frames, respectively. Bitwise row operations are performed similarly.

Also unlike a store command, since intermediate results are stored within the memory cells, from the host point of view, a PIM operation may change unexpected memory regions. Furthermore, the fact that the main memory is a different module, separated from the host chip, allows different PIM modules with different technologies and instruction sets to connect to the same host. Different PIM modules might mean different encoding for a PIM operation, different complex operations, and different implicit changes of memory cells due to these complex operations. Hence, for a host supporting multiple different PIM modules, the host hardware cannot easily know what addresses are affected by a PIM operation.

Supporting virtual memory is possible with bulk-bitwise PIM. Recent work [25] has shown that bulk-bitwise PIM operations can be restricted to work within the boundaries of a single memory huge-page [24]. This allows user-level code to issue PIM operation commands over virtual memory using the standard memory translation procedure. PIM huge-pages can be allocated to the extent that the PIM memory capacity allows, where the computations within different huge-pages are independent. To perform the same computation on several pages, a PIM operation should be sent to each page separately.

B. Consistency Models

In a shared memory system, where multiple threads concurrently access the same memory region, the order of these accesses *must* be well defined [22]. Using the ordering of the memory accesses, a multi-threaded program can be proven to correctly or incorrectly execute the desired task; otherwise, the correct execution of a program cannot be guaranteed. For instance, if a read and a write to the same memory address can be performed in any order, we cannot know if the read result will be the value before or after the write operation. The possible ordering of memory accesses, or the memory ordering rules for a system, is referred to as the *consistency model* of that system [22].

When reordering of memory operations is discussed, the reference is to the difference between two order relations, the *program order* relation of the issuing thread and the *memory order* relation of another thread, each defining a happen-before relation. A thread's program order is the order of memory operations appearing in the program code of the thread. A thread's memory order is the order of all memory operations in the system as seen by the thread. Memory operations are reordered if (1) they appear in the program order of thread 1, and (2) they appear in a different order in the memory order of thread 2.

Numerous consistency models exist in modern processors [22]. These can be ranked by how strict or relaxed their ordering rules are. Whereas stricter models define more rules that must be obeyed, relaxed models allow more reordering possibilities. To enforce order where reorder is possible, consistency models supply additional operations, *e.g.*, atomic operations (also referred to as read-modify-write operations [26]) and memory fences. Atomic operations read a value from memory, modify it, and write it back to the memory such that these operations are consecutive and cannot be reordered with any other memory operation. Memory fence operations ensure that memory operations before and after the fence (in program order) cannot be reordered. Due to the possible reordering of memory operations, some subtle and unexpected behaviors can occur when consistency models become more relaxed [6], making it hard to program with relaxed models. The motivation for having more relaxed consistency models is to make the hardware more performant by allowing concurrent operations of more memory requests.

III. Consistency Models For Bulk-Bitwise PIM

This section presents four possible consistency models for PIM operations, from the strictest to the most relaxed. These models extend, without violating, the existing host processor consistency model by specifying the ordering rules involving PIM operations. Ordering rules not involving PIM operations of existing models are not modified. Implementations are discussed in Section V.

Before we present the proposed consistency models, we define our PIM operation model. A PIM operation, in the context of this paper, referred to as a *PIM op*, is a memory operation issued by a thread to a specific memory address range. We call this range of addresses the *scope* of the PIM op. The PIM op does not necessarily use or modify all addresses in its scope; it is, however, limited to operating only within these addresses. Furthermore, the PIM memory is partitioned into a fixed set of scopes, each with a fixed address range with no addresses overlapping between scopes. The fixed set of scopes and scope sizes (which can range from a cache-line [1] to the entire memory [29, 31]) are architectural values, defined by the system being used. PIM ops can only be issued to scopes from this fixed set. If data in multiple scopes require the same processing, the required PIM ops should be duplicated for each scope. When a PIM op is issued, its scope identity must be available to the host hardware (*e.g.*, the address of the PIM op), providing the host information where to route the PIM op and what addresses might be affected by the PIM op. The scope identification is implementation depended, for example, in [25], 1GB huge-pages are used as scopes and are identified by the PIM op address. Knowing the scope of a PIM op, not the exact addresses that are read and written by the operation, forms an abstraction for the host hardware.

The scope abstraction enables the same host to connect with different PIM modules, each possibly utilizing different PIM instruction sets.

We now describe four potential consistency models from the strictest to the most relaxed.

Atomic model. For our first proposed consistency model, we observe that bulk-bitwise PIM ops, when reaching the memory, are performed atomically. By atomically, we mean in the sense that once the PIM op starts execution, the memory array is occupied until the operation is complete, not allowing other PIM ops, reads, or writes to be executed. As a PIM op can be modeled by many loads and stores, it seems intuitive to model the behavior of a PIM op as multiple loads and stores performed atomically, *i.e.*, an atomic read-modify-write operation [22] for a large address range. If we view a PIM op as an atomic operation for its scope, we get a consistency model for PIM ops where no memory operation from the same thread may be reordered with a PIM op. We refer to this model as the *atomic model*.

Store model. Another possible view of PIM ops in terms of existing memory operations can be as memory stores [25]. This follows the intuition that a PIM op does not read any data to the host processor. The PIM op only atomically writes to many addresses. Hence, PIM ops should have the same ordering rules as the host has for store operations. This view is more relaxed than the atomic model since store operations usually allow some reordering with other memory operations [22,30]. One important difference between PIM and store operations is the address range they affect. Stores usually affect several bytes while PIM operations may affect an entire scope (up to the entire memory [29,31]). As programs expect to read the last value written by the program order, memory operations to overlapping address ranges are not allowed to reorder. Hence, PIM ops must not reorder with memory operations to the same scope. We refer to this model as the *store model*.

Scope model. Both atomic and store models assign familiar ordering rules to PIM ops, which might make PIM ops more intuitive to use as they will behave similarly to known operations. These ordering rules, however, ignore the unique characteristics of PIM ops. Treating PIM ops as their own class might produce a model that performs better and is better suited for PIM. We note that bulk-bitwise PIM ops are usually used because of their high throughput [9,12,25,29,33], performing numerous instances of the same computation concurrently, where the exact order of all the instances does not matter. These computations are performed on large data sets spanning many scopes, allowing the different scopes to process in any order. Hence, for the third consistency model, we allow PIM ops to reorder with any memory operation that is not assigned to their scope. Nevertheless, as with the store model, PIM ops are strictly ordered with operations to the same scope. To enforce the PIM ops order with other memory operations to other scopes, a dedicated fence should be used [21]. We refer to this model as the *scope model*.

Scope-relaxed model. For the fourth model, we note that PIM ops usually do not operate on all the addresses within their scope [9,25,29,31], and it might be safe to reorder some PIM ops with other memory operations to the same scope. This knowledge – which PIM ops and memory operations are safe to reorder – is available to the software initiating the PIM ops. Hence, we take an example from existing relaxed consistency models [22] and allow the software to indicate whether or not PIM ops and memory operations of the same scope can be reordered. In this model, PIM ops can be reordered with memory operations from other scopes and the same scope. To allow the software to enforce order between PIM ops and memory operations to the same scope, we include a new fence, termed the *scope-fence*, which guarantees order only within a single scope. To enforce order between PIM ops and operations from other scopes, a dedicated fence (as in the scope model) is used. We refer to this model as the *scope-relaxed model*.

To summarize, we propose four possible consistency models for PIM ops (see also Table I):

- Atomic model: PIM ops are not allowed to reorder with any memory operation.
- Store model: PIM ops take the same ordering rules as store operations in the host's consistency model.
- Scope model: PIM ops can be reordered with any operations to other scopes, but not to the same scope.
- Scope-relaxed model: PIM ops can be reordered with operations to other scopes and the same scope. Order is enforced by dedicated fence instructions.

IV. Supporting Coherency

As identified in Section I, to support any ordering rules with PIM ops, the cache flushes of affected addresses must be atomic with the PIM ops. The most straightforward solution is to mark PIM memory regions as uncacheable, as suggested for PIM techniques other than bulk-bitwise PIM [15,23]. For bulk-bitwise PIM, however, reading the PIM results using memory loads is the most time-consuming step in the PIM computation [25]. Because these loads utilize spatial locality in the cache, making the PIM-operated memory region uncacheable will degrade performance substantially for bulk-bitwise PIM. Fig. 3 compares the uncacheable approach and the software flush approach (described in Section I) to a naive approach without any coherency solution (*i.e.*, issuing PIM ops without a correctness guarantee); for details, see Section VI. As the PIM data set size increases, the PIM result size increases, requiring more data to be read from the PIM. Hence, the uncacheable approach run time becomes 2.57× worse than the naive approach, while the software flush approach run time is only 1.09× worse.

Rather than making the PIM-operated memory region uncacheable, we offer to support PIM coherency by enabling the PIM ops, on their way to the main memory, to flush all addresses from their scope. The scope flush might include data that is not used by the PIM op, an overhead of the scope abstraction (Section III). As the handling of PIM ops is a small part of the total execution time [25], coupling the cache

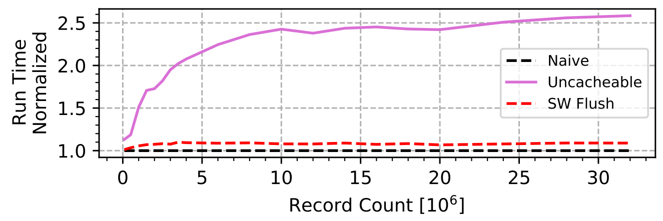

Fig. 3: Run time of the YCSB workload on the Naive baseline (no coherency solution), the uncacheable approach (the Naive baseline where PIM-enabled scopes are uncacheable), and the Software Flush approach (flushing cache-lines from PIM-enabled scopes). The system and YCSB workload are detailed in Section VI.

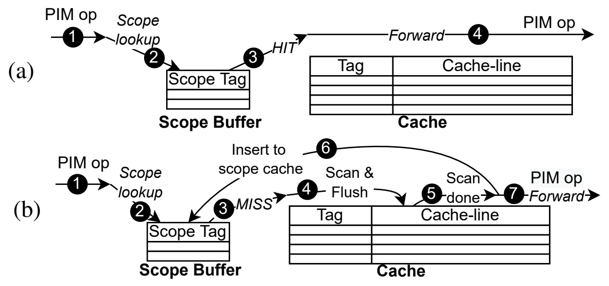

Fig. 4: Operation of PIM ops arriving at a cache with a scope buffer. Numbers in dark circles indicate the event sequence. (a) Hit in the scope buffer. (b) Miss in the scope buffer.

flushes with PIM ops is more promising than coupling them with the loads and stores (*i.e.*, making data uncacheable).

The challenge in supporting cache flushes by PIM ops is the need to identify all cache-lines belonging to the PIM op scope. This is a challenge since scopes can be large, scattering cache-lines in all cache sets. Additionally, the cache structure does not inherently support a search for memory regions bigger than a cache-line. Thus, each PIM op is required to scan all cache sets one by one, and find the relevant cache-lines. This scan may take thousands of clock cycles, blocking the cache for the entire scan period and delaying other memory operations. To alleviate the cache scan overhead, two hardware techniques are suggested, the *scope buffer* and the *scope bit-vector*.

A. Scope Buffer

We note that usually bulk-bitwise PIM has fine-grained instruction sets (*e.g.*, AND, OR, NOT, ADD, MUL) [9, 18, 25, 29, 31], requiring multiple PIM ops to perform a full computation. This suggests that if a PIM op for a scope is issued, then other PIM ops to the same scope will most likely follow, without intermediate loads or stores to the scope. This implies that PIM op scopes exhibit temporal locality and a single cache scan and flush can be performed for several consecutive PIM ops. To take advantage of this temporal locality, we propose adding a structure called the *scope buffer* to the cache. The scope buffer's structure is similar to that of a cache [24] indexed by scope addresses and holding entries for scopes that were recently flushed from the cache. When a PIM op arrives at the cache, a lookup for its scope in the scope buffer is made. If the scope is found (Fig. 4a❶❷❸), the PIM op is forwarded towards the main memory without performing a cache scan (Fig. 4a❹). If the scope is not found in the scope buffer (Fig. 4b❶❷❸), a cache scan must be performed set-by-set, flushing all cache-lines of that scope (Fig. 4b❹). After the scan is finished (Fig. 4b❺), the scope is inserted into the scope buffer (Fig. 4b❻) and the PIM op is forwarded (Fig. 4b❼). If the scope buffer is full and a new scope needs to be inserted into it, the new scope simply overwrites an old scope according to a replacement policy (*e.g.*, LRU) with no additional action. Thus, when a stream of PIM ops to a single scope is sent to the memory, the first PIM op will perform a full cache scan and flush on the scope, and the rest of the PIM ops will pass through the cache without triggering further action.

When a cache-line is inserted into the cache, a lookup request for the scope of this cache-line is made in the scope buffer. If found in the scope buffer, this scope is erased from the scope buffer with no additional action. If the scope is not found, nothing is done. Note that the interaction with the scope buffer for loads and stores takes place in parallel to the operations in the cache and off the critical path; thus, the cache operation latency is not affected.

To determine the required scope buffer size, we noted the following. First, after each PIM computation the results are read from all involved scopes, invalidating these scopes from the scope buffer before the next PIM computation. Second, during a single PIM computation, a scope in the scope buffer is useful only during the issuing of PIM ops to that scope. Therefore, after PIM ops to a scope are done issuing, the scope entry in the scope buffer is not useful anymore. Hence, the scope buffer is only required to hold the scopes that are currently being issued with PIM ops. The number of such scopes depends on the processor configuration and software code. Similarly to choosing a cache size [10], the scope buffer size can be set according to the processor configuration. We have seen that even a small-sized scope buffer is sufficient to achieve close to the maximum possible hit rate for all data set sizes (see Section VII).

B. Scope bit-vector (SBV)

Although the scope buffer reduces the number of cache scans, the remaining cache scans' latency can reach thousands of clock cycles. We note that for bulk-bitwise PIM, the results of a computation are stored on multiple crossbar arrays at the same specific crossbar columns and rows [9, 25, 33], giving a regular non-continuous address range for the result because of the address mapping of crossbar columns and rows [9, 25]. The consequence of PIM results having regular non-continuous address ranges is that a PIM result tends to cluster in a subset of cache sets. The cache sets in this subset depend on the workload and architecture, potentially including any cache set, but tending to be less than all cache sets.

To take advantage of the fact that cache-lines from PIM-enabled scopes tend to cluster in a subset of the cache sets, we propose an additional structure that keeps track of these cache sets. We name this structure the *scope bit-vector* (SBV). As the name suggests, the SBV is a bit-vector with a single bit for each cache set. A bit in the SBV is set high if its corresponding cache set contains a cache-line from some PIM-enabled scope; otherwise, the bit is kept low. Using the SBV, a cache scan for a PIM op is required to check only the sets that have a high bit in the SBV. To enable the use of the SBV, a cache-line from a PIM-enabled scope must be marked as such. This marking can be implemented, for example, by defining a memory page as PIM-enabled, encoding this information in the translation table page entry, attaching it to every memory request for this page (similar to marking a page as uncacheable [10]), and adding this information to each cache-line's meta-data. In this way, when a cache-line from a PIM-enabled scope is inserted into the cache, the relevant bit in the SBV is turned to high. When a cache-line from a PIM-enabled scope is evicted, all remaining cache-lines in the same set are checked to see whether at least one belongs to some PIM-enabled scope and the SBV bit is updated accordingly.

To summarize the above, to maintain coherency between the host caches and the PIM module, we suggest that PIM ops flush cache-lines from their scope on their way to the memory. When a PIM op arrives at the cache, a lookup for the scope of the PIM op is made in the scope buffer. The scope buffer lookup answer indicates if the PIM op can be forwarded or if a cache scan is required to find and flush all cache-lines from the scope of the PIM op. If a scan is required, the SBV is used to identify the cache sets to scan.

V. Supporting PIM Consistency Models

Supporting coherence is not enough to enforce the consistency models suggested in Section III. How the host cores, memory subsystem, and the PIM module handle PIM ops must also be specified. Therefore, in this section, we present an implementation for each consistency model. These implementations require only a minimal hardware overhead while still enabling the reordering allowed by each consistency model. Table I summarizes the implementation for each model.

A. Base Host Processor and PIM Module

For the host of the PIM module, we take a multicore system illustrated in Fig. 5, as is commonly used for bulk-bitwise PIM architectures [9, 25, 31]. Other hosts (*e.g.*, GPU [21]) can be used similarly. To keep our implementation general and not rely on host-specific attributes, we assume the host has the following common capabilities: (1) The host instruction set contains a dedicated instruction to issue a PIM op (similarly to a store instruction issuing a memory write) [1, 5, 9, 21, 25, 29]. (2) Host cores can execute out-of-order and issue non-PIM memory operations according to the host's native (non-PIM) consistency model [22]. (3) The host memory subsystem can reorder operations passing through it, *e.g.*, by a multi-path network-on-chip [14], virtual channels [14], or non-FIFO buffers [22]. (4) The host memory subsystem consists of an inclusive, shared last-level-cache (LLC), as is prevalent

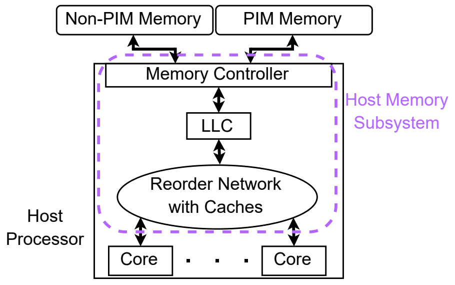

Fig. 5: The host and system used to demonstrate the proposed bulk-bitwise PIM consistency models.

in modern processors [10]. The rest of the cache levels are possibly private or shared. (5) The host memory controller can reorder operations, but does not violate data dependencies between operations. For example, two writes to the same address will be executed in the order of arrival to the memory controller, but writes to different addresses can be reordered. Similarly, the memory controller will not reorder a PIM op with other memory operations that address the same scope.

As the PIM module itself may contain additional routing and logic to handle memory operations [25, 31], it is assumed that the PIM module, like the host's memory controller, does not violate data dependencies between operations. Hence, when a PIM op arrives at the host memory controller, it cannot reorder with other memory operations. To understand why, consider the scenario where the memory controller received PIM op A, and had not yet forwarded it when another memory operation B is issued from another thread, not necessarily arriving at the memory controller. To see if B happened before A, a thread needs to see that B happened and A has not. To do this, another memory operation C has to be sent to A's scope (*e.g.*, a read) after B; however, then C will arrive at the memory controller after A, and as A and C operate on the same scope, the memory controller and the PIM module will keep their order, making C operate after A, and see its results. Consequently, no thread can see B happening before A. This means that when a PIM op arrives at the memory controller, it is safe for the issuing thread to continue issuing other memory operations without the risk of violating memory ordering with that PIM op. Therefore, to enforce the required order of PIM ops and other memory operations, it is sufficient to enforce it only between the host cores and the memory controller, *i.e.*, enforcing the order in the cores, caches, and the on-chip network.

B. Atomic Model

To support the atomic model with minimal modifications to the host, we add a scope buffer and an SBV only to the LLC. Since the LLC is inclusive, flushing cache-lines from the LLC will automatically flush the cache-lines from all cache levels, saving the need to flush each cache level in turn. Additionally, the LLC is the largest cache level, making the added hardware and complexity overheads more reasonable.

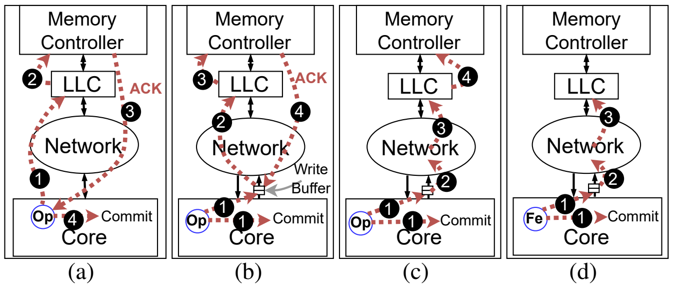

Fig. 6: Consistency model implementations. Op indicates a PIM op. Fe indicates a scope-fence. (a) Atomic model's PIM op issuing process. (b) Store and Scope models' PIM op issuing process. (c) Scope-relaxed model's PIM op issuing process. (d) Scope-relaxed model's scope-fence issue process. See Section V for details.

Model	PIM Op Allowed Reordering	Additional Fence Required	Scope Buffer & SBV
Atomic	None	No	Only LLC
Store	Same as store operations	No	Only LLC
Scope	All operations to other scopes	Ordering between scopes	Only LLC
Scope-Relaxed	All operations except fences	(1) Ordering within scope and (2) between scopes	All caches

TABLE I: Consistency model definitions and implementations.

The process of issuing a PIM op in the atomic model is shown in Fig. 6a. Here, a core treats a PIM op instruction as a memory operation with a memory fence before and after it, not allowing any memory operation to reorder with the PIM op. When it is safe to commit the PIM op, it is issued to the host's memory subsystem (Fig. 6a❶). The core, however, does not yet commit the PIM op, thereby keeping the thread from issuing other memory operations and preventing their reordering with the PIM op at the memory subsystem. The PIM op is forwarded to the LLC without affecting the lower cache levels. When the PIM op arrives at the LLC, the LLC flushes all cache-lines from the scope of the PIM op to the memory as explained in Section IV. When the flush is complete, the LLC forwards the PIM op to the memory controller (Fig. 6a❷). Once the PIM op arrives at the memory controller and ensures the order of the PIM op, the memory controller sends an ACK to the issuing core (Fig. 6a❸). Upon receiving the ACK, the core commits the PIM op instruction (Fig. 6a❹).

C. Store Model

The store model is supported similarly to the atomic model. The difference is that a core treats a PIM op as if it were a store operation, possibly allowing reordering with operations outside its scope (depending on the host's reordering rules for stores). Additionally, between the time a PIM op is issued to the memory subsystem and the time its ACK is received, the core can still issue operations that are allowed to bypass PIM ops. For example, in an X86-TSO core [30], a load can bypass a store to a different address. Therefore, a load to a scope different than the PIM op's scope can still be issued to memory while the PIM op instruction waits for its ACK.

Fig. 6b shows the process of the store model for PIM op issuing. When it is safe for the core to commit a PIM op, it is both issued to the host's memory subsystem and committed (Fig. 6b❶). To prevent reordering by the memory subsystem, the entry point to the memory subsystem (*e.g.*, the write buffer [10,22]) enforces the ordering rules. After sending a PIM op (Fig. 6b❷), the entry point must prevent relevant memory operations from entering the memory subsystem. Similarly to the atomic model, a PIM op flushes only the LLC and is forwarded to the memory controller afterwords (Fig. 6b❸). The memory controller then sends an ACK to the memory subsystem entry point (Fig. 6b❹), indicating that it is safe to issue the withheld memory operations. This process of PIM op issuing is similar to a store operation's issuing process with a write buffer [22] – only in our case, the host memory controller, rather than the L1 cache, indicates completion.

D. Scope Model

The scope model is supported similarly to the store model and as shown in Fig. 6b. The difference is that we want to allow reordering for PIM ops with memory operations to other scopes. Hence, the memory subsystem entry point is required to hold back only operations to a scope of an ongoing PIM op. Other operations can enter the memory subsystem without waiting for the PIM op's ACK (*i.e.*, a non-FIFO write buffer). To enforce ordering between PIM ops of different scopes, the dedicated fence from [21] is used. By allowing more memory operations to clear the memory subsystem entry point, the entry point can receive more operations from the core, preventing the core from stalling on resource contention. Allowing the memory subsystem to handle more operations concurrently, while the core continues execution, potentially leads to better system utilization and performance.

E. Scope-Relaxed Model

The scope-relaxed model allows PIM ops to reorder with memory operations to the same scope, hence cores can reorder PIM ops with all other memory operations. Thus, the scope-relaxed model's process of PIM op issuing, shown in Fig. 6c, allows the core to issue a PIM op at commit (Fig. 6c❶), and also does not require the host's memory subsystem entry point to hold back any memory operation when forwarding a PIM op (Fig. 6c❷). This enables multiple PIM ops, from the same and different scopes, to be inserted into the memory subsystem at the same time and cleared from the entry point, increasing system utilization compared to the previous models. As with the previous models, to preserve the atomicity of PIM ops and cache flushes, PIM ops must flush their scope from the LLC (Fig. 6c❸) before being forwarded to the memory controller (Fig. 6c❹). Unlike the previous models, however, PIM ops must pass through all cache levels on their way to the LLC (Fig. 6c❷) without flushing them (reason given below) and do not require the memory controller to return an ACK.

Evaluation System			
Processor Cores	6 cores, X86, OoO, 3.6GHz	Main Memory	32GB DRAM, DDR4-2400
L1 cache	Private, 16KB, 64B block, 4-way	L2 cache	Shared, 2MB, 64B block, 16-way
L1 scope buffer (if exists)	16 sets, 1-way	L2 scope buffer	64 sets, 4-way
Coherency protocol	MESI	PIM modules	1 (spec as in [25])
Scope	2MB huge page	Max. database records per scope	32K

TABLE II: Architecture and system configuration to capture the consistency model behavior.

To enforce ordering between PIM ops and memory operations from the same scope, a scope-fence, implemented as in [21], is used (Section III). The scope-fence's issuing process is similar to that of a PIM op's, shown in Fig. 6d. The scope-fence is issued from the core at commit time (Fig. 6d❶). Also, the scope fence passes through all cache levels on its way to the LLC (Fig. 6d❷). Unlike a PIM op, however, the scope-fence propagates through all optional paths to a destination [21], *i.e.,* the next cache level, duplicating the scope-fence packet as necessary. On the way to a destination, all memory operations to the scope-fence's scope are not allowed to reorder around the scope-fence in any path. Once all copies arrive at the destination, the scope-fence is forwarded to the next destination (through all optional paths). The scope-fence is terminated at the LLC (Fig. 6d❸). Also, unlike a PIM op, a scope-fence must flush its scope in all caches on its path. Otherwise, the following may happen. Consider a scenario where a PIM op, a scope-fence, and a load, all from the same thread and to the same scope, are issued in that order from a core. Since the issued PIM op and scope-fence do not block the load in the scope-relaxed model, the load can be issued. The load may hit in a lower level cache (skipped by the scope-fence) before the PIM op has reached the LLC and flushed the load's data from the caches, view the pre-PIM value, and effectively be ordered before the PIM op. This, of course, breaks the ordering guarantee of the scope-fence.

Hence, all caches in this implementation include a scope buffer and an SBV. PIM ops perform a cache scan only at the LLC, while scope-fences perform a cache scan in all caches on the path between the issuing core and the memory controller. Note that without a scope-fence, the scope-relaxed model allows the load in the previous scenario to reorder with the PIM op, so the PIM op is not required to perform a scan on all caches. PIM ops, however, do need to be routed through all cache levels on their way to the LLC, so they might be ordered by a scope-fence. To enforce order between PIM ops from different scopes, an additional fence with the same implementation of [21] is used (as in the scope model), affecting all memory operations from all scopes.

Number of Operations	1000
Scan Operation Percentage	95%
Insert Operation Percentage	5%
Number of Fields per Record	5
Field Length	10B
Records in Scan Results	Uniform dist. [1,100]
Scan Base Record	Zipfian dist.

TABLE III: YCSB [7] workload summary. The number of records varies with experiments.

Query	# Scopes	PIM section	Query	# Scopes	PIM section
q1	1832	Full-query	q12	1832	Filter only
q2	66	Filter only	q14	1832	Filter only
q3	2336	Filter only	q15	1832	Filter only
q4	2290	Filter only	q16	62	Filter only
q5	508	Filter only	q17	62	Filter only
q6	1832	Full-query	q19	1894	Filter only
q7	1882	Filter only	q20	2294	Filter only
q8	566	Filter only	21	1832	Filter only
q10	2290	Filter only	q22	46	Full sub-query
q11	4	Filter only			

TABLE IV: TPC-H [32] query summary. Queries 9, 13, and 18 do not have a PIM section [25] and are, therefore, not evaluated.

VI. Methodology

A. System Simulation

To compare the performance and behavior of our consistency models and coherence solutions, we developed a bulk-bitwise PIM simulator[1] based on the gem5 simulator [3]. We adopted the solution of [25] as our PIM module, since, to the best of our knowledge, this is the only available bulk-bitwise PIM solution using virtual memory. All consistency models were implemented as described in Section V with a six-core multicore and a two-level cache hierarchy (L2 is the LLC) with a MESI coherency protocol [22]. All simulations were performed on the gem5 full-system mode, running a Linux kernel, and using the gem5 Ruby memory system. The details of the simulated system are listed in Table II. Estimating the scope-buffer and SBV hardware overhead for the L2 cache, using the Synopsys 28nm library, we observe a 0.092% area overhead. The scope-relaxed model requires a scope-buffer and an SBV for all caches, reaching a total of 0.22% area overhead.

B. Workload

Previous works suggest that database workloads, specifically scanning operations, are a promising application for bulk-bitwise PIM [9, 25, 29, 31]. Hence, as a representative workload for bulk-bitwise PIM, we take the YCSB short-range workload [7], a key-value database workload divided into 95% scan operations and 5% record insertions, and queries from the TPC-H [32], an online analytical database benchmark. Other workloads from the YCSB benchmark were not used as they do not involve bulk-bitwise PIM operations.

In the YCSB benchmark, a database scan operation searches for a set of records from a single database relation (organized

[1]Available at:
https://github.com/benperach/gem5_bulkbitwise_PIM_consistency.

as in [25]) and extracts a certain text field from each found record. For each database scan, the number of result records and the field to extract are randomly generated. To measure this workload on the suggested consistency models, we used runs of 1000 operations (95% scans, 5% insertions, randomly ordered) on a varying number of records. All database scans, reads, and insertions were performed within the PIM module memory. The parameters of the workloads are summarized in Table III. Insertions were performed using standard stores, and scans were performed by: (1) Dividing the scopes of the database evenly among four threads, (2) having each thread issue PIM ops to perform the scan on each of its assigned scopes, and (3) having each thread read the scan result and the required record fields from the database contained in its assigned scopes using standard loads.

To show the performance and behavior trends of our four models, we ran the YCSB workload on a range of database sizes, from 0.1×10^6 to 32×10^6 records, occupying from 4 to 977 scopes. For all scope counts and all models, the same sequence of scans and insertions was measured. Records are randomly distributed in the database, making the scan result evenly distributed across the scopes. The range of scope counts was chosen to capture the behavior of our models. As shown in Section VII, the measurements reached a steady trend on the high end of the scope count range, allowing us to focus on the above-mentioned scope count range.

For the TPC-H queries, each query was run ten times consecutively. Each query run was performed as in [25], executing only the PIM section of the query and reading the results. The PIM section of the query is either only filtering the involved database relations or performing the entire query when a single relation is involved. Queries 9, 13, and 18 were not performed since they do not include any PIM section. Table IV lists the TPC-H queries key parameter; see [25] for details.

An important difference between the TPC-H and YCSB workloads is in the number of reads performed after each PIM computation. For the TPC-H, only the PIM computation result is read, resulting in a regular read pattern that is mapped to a limited group of cache sets (see Section IV-B). For the YSCB workload, the result of the PIM computation indicates what other data from the PIM memory have to be read. The latter data have a different access pattern and are mapped to a different group of cache sets, creating more work for the PIM coherence mechanism (*e.g.*, software flushes, cache scans).

C. Comparison Baselines

We compared the proposed consistency models and their implementations to two baselines. The first baseline is the software flush approach presented in Section I and used in previous bulk-bitwise works [9, 25]. In this approach, the software running on the host is responsible for maintaining coherency by explicitly issuing cache-line flushes. Host cores issue PIM ops to the memory subsystem on commit. Thereafter, PIM ops are forwarded directly to the memory controller without performing any operation in the memory subsystem.

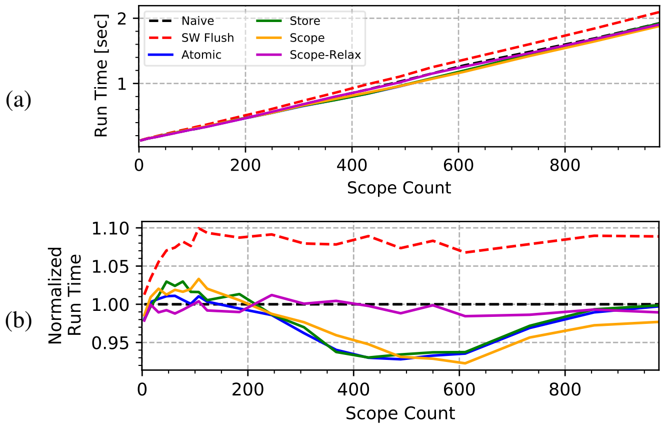

Fig. 7: YCSB benchmark. (a) Absolute run time. (b) Run time normalized to the Naive baseline. The Naive and SW Flush baselines (dashed lines) do not guarantee correct execution.

As discussed in Section I, such an approach cannot guarantee correctness. We refer to this baseline as *SW Flush*.

The second baseline is a naive approach where the cache is not flushed at all, *i.e.*, the program is executed as in the SW Flush baseline, only without the software flushes. Cache-lines are evicted only by normal operation during loads and stores. This baseline does not provide correct execution; it serves to show the performance overhead of the different consistency models. We refer to this baseline as *Naive*.

VII. Evaluation

We start by comparing the four proposed models and the two baselines. Fig. 7 and Fig. 8 show the run times of the YCSB and TPC-H workloads, respectively, for all models and baselines. Fig. 9 and Fig. 10 show some system statistics.

The YCSB workload (Fig. 7) shows several regions of interest. In the first region, up to 100 scopes, the run time of the four models and the SW Flush baseline increase relative to the Naive baseline. In this region, the number of used PIM-enabled scopes is relatively low, resulting in a lightly loaded system and a low number of load operations to memory to retrieve the PIM results. This makes the overhead of managing the PIM ops more prominent, as shown by the increasing performance benefits of the Naive baseline.

For more than 100 scopes, the relative overhead of the SW Flush is constant. As the number of flush operations is proportional to the workload size, the flush relative overhead becomes constant when the workload size-dependent execution (*e.g*, PIM computation, result read) becomes the dominant part of the run time.

Additionally, from approximately 100 scopes, the performance of the four models improves relative to the Naive baseline. The reason for this is that with the increasing scope count, more PIM ops are required. As PIM ops have a long PIM execution time, the host cores, in all models, issue PIM ops at a higher rate than the PIM module can process them. As a result, the PIM module buffer is filled (Fig. 10a), back-

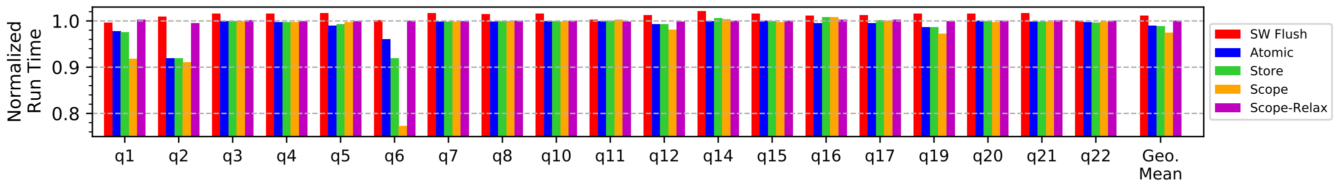

Fig. 8: Run time for TPC-H queries normalized to the Naive baseline. The Naive and SW Flush baselines do not guarantee correct execution.

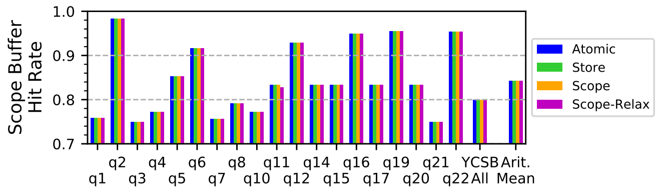

Fig. 9: Scope buffer hit rate for TPC-H and YCSB.

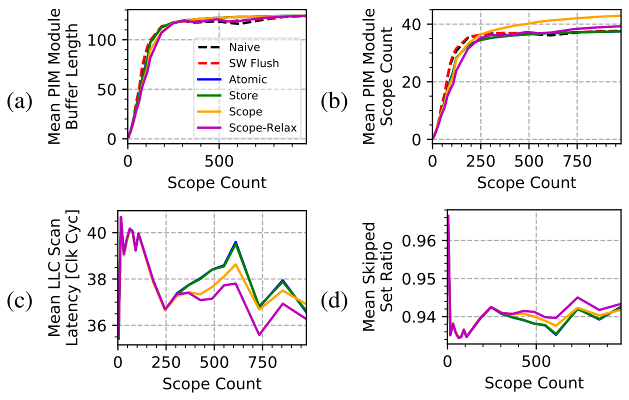

Fig. 10: System statistics for the YCSB workload. (a) Mean PIM module buffer size on PIM op arrival. (b) Mean number of unique scopes at the PIM module buffer on PIM op arrival. (c) Mean LLC scan latency. PIM ops not requiring any scan (scope buffer hit) are counted as zero clock cycle scans. (d) SBV mean ratio of LLC sets skipped out of all LLC sets during an LLC scan.

pressuring the host memory subsystem and saturating it. In the Naive and SW Flush baselines, cores issue PIM ops at a fast rate since there are no constraints on PIM ops. In the four consistency models, however, the ordering mechanisms throttle the cores' PIM ops issue rate, resulting in a lighter load on the host's memory subsystem and allowing other memory operations (*e.g.,* reads from other threads) to execute quicker. The scope-relaxed model behaves similarly to the Naive baseline since it too allows a high PIM op issue rate, also saturating the memory subsystem. We investigate the effect of the PIM module buffer further below.

For more than 600 scopes, the improvements in the models' run time start to diminish. The benefit from lighter loads in the memory subsystem is relevant when one thread issues PIM ops while another thread issues read operations. The lighter load allow the threads to execute concurrently by interleaving their operations. This overlap, however, shortens, relative to the whole run time, as the workload size increases. Thus, when the workload size increases, the run time differences depend mostly on the execution time when all threads issue PIM ops. When all threads issue PIM ops, the scope model, due to its inherent interleaving of PIM ops from different scopes, has the best run time. In the scope model, the core's write-buffer holds back PIM ops that have an ongoing PIM op to their scope, but allows other PIM ops to continue. This results in PIM ops from different scopes arriving at the PIM module's buffer in an interleaved manner, shown in Fig. 10b as the increased number of unique scopes in the PIM module's buffer. Having PIM ops to more scopes, the PIM module can concurrently execute more PIM ops, increasing parallelism and shortening total execution time. The other models issue PIM ops in program order, resulting in PIM ops from the same scope reaching the PIM module together and executing serially.

The relative run time of the models on the TPC-H queries is shown in Fig. 8. The models show little run time difference on most queries. When the difference between the models is significant, the scope model has the best run time, followed by the store and atomic models; the scope-relaxed model is always similar to the Naive baseline. The same effects described above for the YCSB workload also apply to the TPC-H queries. Queries q1, q2, q6, q12, q14, q15, q19, and q20 are the only queries that reach a substantial PIM module buffer occupancy, due to a combination of the number of scopes, PIM ops per scope, and PIM ops latency (not shown). Queries q14, q15, and q20 have a few PIM ops per scope and a relatively short PIM execution time per scope, easing the back-pressure created by the full buffer to the point that all models perform similarly. For q1, q2, q6, q12, and q19 we see a visible difference between the models. Queries q2, q12, and q19 have more and longer PIM ops per scope relative to other filter-only queries. Furthermore, q2 has few scopes, reducing the resulting read phase's execution time. For q1 and q6, being full-queries, the PIM section is substantially longer and there are fewer results to read [25]. For these reasons, queries q1, q2, q6, q12, and q19 have a more substantial PIM section and thus a performance difference between the models is visible.

LLC Scan: Fig. 9 and Fig. 10 shows statistics of LLC scans and the performance of the scope-buffer and SBV at the LLC. Since the scope buffer is large enough to hold all concurrently issued scopes in all models, the first PIM op to a scope misses in the scope buffer while all the other PIM ops to that scope hit. This results in the same hit rate for

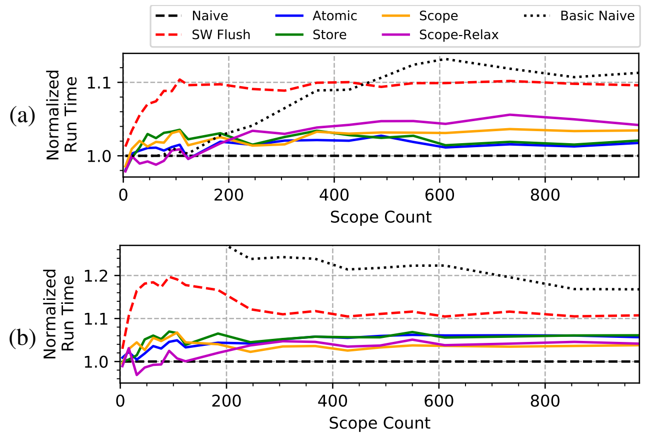

Fig. 11: Normalized run time for the YCSB workload and a PIM module with (a) an unbounded buffer and (b) zero PIM operation latency. Both (a) and (b) are normalized to the Naive baseline and also show the Naive baseline with the basic configuration (marked as *Basic Naive*).

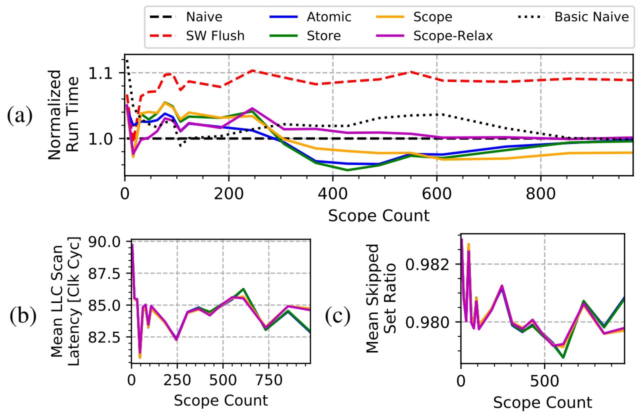

Fig. 12: Experiment result for configuration with an 8*MB* LLC for the YCSB workload. (a) Run time normalized to the Naive baseline, including the 2*MB* LLC Naive baseline (marked as *Basic Naive*). (b) LLC scan mean cycle count. (c) SBV mean rate of LLC sets skipped during an LLC scan. In (b) and (c), store overlaps atomic, and scope-relax overlaps scope.

all models, as shown in Fig. 9. The mean latency of LLC scans for the YCSB workload (Fig. 10c) is approximately 38 clock cycles. This latency is much lower than the number of LLC sets (2*K* sets), resulting from the scope-buffer and SBV operations. To see the effectiveness of the SBV, Fig 10d shows that the mean rate of skipped sets during a scan range is close to 94% for the YCSB workload. For all TPC-H queries with all models, the mean SBV skipped sets ratio is above 99% (not shown). In both Fig. 10c and Fig. 10d, we see that our four models perform the same up to approximately 250 scopes and then start to diverge. This divergence occurs when the back-pressure from the PIM module reaches the host memory controller and fills its buffers. At that point, PIM ops cannot enter the memory controller and the host cores must lower their PIM op issue rate. The lower issue rate allows PIM reads from other cores to interleave with the PIM ops, marking more cache sets in the SBV that are not skipped during an LLC scan. The stricter the model, the lower the issue rate becomes and the more interleaving it allows.

Unbounded Buffer: The limited buffer of the PIM module creates back-pressure and blurs the difference between the consistency models. To eliminate this effect, Fig. 11a shows the normalized run time for the YCSB workload when the PIM module buffer is unbounded. This measurement evaluates the system performance for the scenario where the PIM module buffer is sufficiently large for the used application. The unbounded buffer allows all PIM ops to reach the PIM module. Thus, operations to different scopes can be executed concurrently as soon as they reach the PIM module. Nevertheless, there is still no significant behavior difference between the consistency models because of the execution latency of the PIM module. The latter takes numerous cycles [25] and allows all consistency models to issue all PIM ops relatively quickly from the cores.

With an unbounded buffer, the Naive approach achieves the best run time. As PIM ops arrive to the memory at a faster rate, more parallelism in the PIM execution is uncovered. This benefit is small (less than 6% relative to the four models) as the difference is in the PIM op management while the bulk of execution time is used for the PIM execution and read latency. Note that the scope and scope-relaxed models, though more relaxed and with the faster PIM op core issue rate, perform slightly worse (2%) than the atomic and store models. The fast issue rate congests the host memory subsystem, hurting the execution of other memory operations as they pass through the same network and caches. The stricter models wait until a PIM op reaches the memory controller to release the next PIM op, with the side benefit being keeping the host memory subsystem clear.

Zero Logic: To remove the effect of the PIM execution latency and account for a best-case bulk-bitwise PIM logic, we changed the basic configuration (Section VI) to have a bulk-bitwise PIM logic execution time of zero. The workload still issues the same PIM ops as before, but all PIM ops have a PIM execution time of zero. The experiment results for the YCSB workload, presented in Fig. 11b, show that the four models still have a maximum 6% overhead compared to the Naive baseline. Additionally, the more relaxed consistency models, scope and scope-relaxed, achieve better performances. As the PIM execution time is zero, the system's management of PIM ops is the dominant factor for the PIM part of the workload; the faster issue rate of the relaxed models enables better performance.

LLC Size: An interesting aspect is the effect of the LLC size on run time. A bigger LLC requires additional LLC scan latency and can increase the run time. Fig. 12 shows the performance of our designs with an 8*MB* LLC for the YCSB workload. The behavior with the 8*MB* LLC (Fig. 12a) is similar to that with a 2*MB* LLC (Fig. 7b), with the difference being a performance degradation relative to the

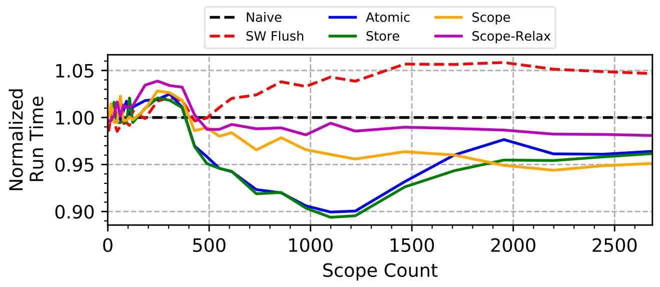

Fig. 13: Normalized run time for the YCSB workload using eight threads on a 16-core host processor. All models are normalized to the Naive baseline.

Naive baseline. This degradation is due to the added LLC scan latency (Fig. 12b), in spite of the increased SBV efficiency (Fig. 12c), making the LLC busier and the host's memory subsystem more congested.

Additional Threads: Another interesting aspect is the effect of additional threads on the relative run time of the models. More threads might increase the load on the memory subsystem and increase the difference between the models. Fig. 13 shows the performance of our designs for the YCSB workload where the scopes are divided between eight threads (instead of four). To allow all threads to run concurrently, we also increased the number of host cores to 16. The behavior for eight threads shows the same trends as with four threads (shown earlier in Fig. 7b), except that eight threads require scaling the scope count to achieve a similar scope count per thread. Additionally, as there are more threads and a greater load on the memory subsystem, the difference between the models and between the Naive baseline is even greater compared to with four threads, especially for the stricter atomic and store models.

The above experiments show that even with the additional activities required to enforce order in our consistency models, PIM ops execution encounters greater bottlenecks: buffer congestion, PIM execution latency, and the result reads. The buffer congestion might be alleviated with a better design (*e.g.*, larger PIM module buffer, better scheduling), but the PIM execution latency and the result read appear as inherent attributes of bulk-bitwise PIM [9, 25, 29, 31], unrelated to the consistency model being used.

VIII. Related Work

Previous work on bulk-bitwise PIM has mostly disregarded consistency, with some works addressing coherency. Perach *et al.* [25] briefly mentioned that cores treat PIM operations as store operations (similar to our store model) but did not address the required support. In [9,25], support for coherency is through explicit software flushes, which break ordering guarantees as described in Section I. Seshadri *et al.* [29] used the memory controller to flush cache-lines before PIM operations. To reduce the flush overhead, the memory controller flushes only cache-lines used by the PIM operation. Such a solution, as opposed to our coherency solution, assumes that the memory controller knows what cache-lines the PIM operations use, making the host and PIM module tightly integrated, which can be impractical in a general scenario.

Prior work on PIM consistency models and coherency for near-memory architecture [13] does exist [1,5,17,19,21]. Near-memory locates processing units on the same die as the memory arrays, but the processing units and memory arrays are distinct and separated modules. As near-memory computing and bulk-bitwise PIM differ substantially in implementation and supported operations, they are used in different ways and suited to different computations. Hence, these solutions for near-memory are mostly not appropriate for bulk-bitwise PIM. In a work on near-memory computation that is relevant for bulk-bitwise PIM, Nag *et al.* [21], showed that standard memory fence operations are insufficient to enforce order among PIM operations. They suggested a new fence mechanism for that goal and demonstrated it on a near-memory architecture with a GPU host. Nag *et al.*, however, did not discuss the order of PIM operations concerning other memory operations and thus did not provide a consistency model. We use their fence mechanism as our fence operation for PIM ops in our proposed models (Section V).

IX. Conclusions

In this paper, we showed the importance of addressing the consistency and coherency of bulk-bitwise PIM systems. We proposed four bulk-bitwise PIM consistency models, from strict to relaxed, and discussed the implementation to support these models, including a low hardware overhead solution for coherency (the scope buffer and the SBV). These consistency models were evaluated on representative database workloads. Our evaluation showed that strict and relaxed models for bulk-bitwise PIM can have similar run times, which are also similar to the run time of a system with no order guarantees. This is because the bulk-bitwise PIM bottlenecks overshadow the overheads associated with the consistency models. Nevertheless, a consistency model and implementation that take into account these bottlenecks – such as our scope model that inherently interleaves PIM operations and lightly loads the host memory subsystem – can reach better performance in some cases and might be a better choice for bulk-bitwise PIM.

References

[1] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," *SIGARCH Comput. Archit. News*, vol. 43, no. 3S, p. 336–348, jun 2015. [Online]. Available: https://doi.org/10.1145/2872887.2750385
[2] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2020.
[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718
[4] J. Borghetti, G. Snider, P. Kuekes, J. Yang, D. Stewart, and R. Williams, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 1476-4687, pp. 873–876, April 2010.

[5] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, “LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory,” *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2017.

[6] S. Chakraborty and V. Vafeiadis, “Grounding Thin-Air Reads with Event Structures,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290383

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[8] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, “Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions,” 2018. [Online]. Available: https://arxiv.org/abs/1802.00320

[9] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. G. Luna, and O. Mutlu, “SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM,” 2021.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[11] B. Hoffer, V. Rana, S. Menzel, R. Waser, and S. Kvatinsky, “Experimental Demonstration of Memristor-Aided Logic (MAGIC) Using Valence Change Memory (VCM),” *IEEE Transactions on Electron Devices*, vol. 67, no. 8, pp. 3115–3122, 2020.

[12] M. Imani, S. Gupta, Y. Kim, and T. Rosing, “FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 802–815.

[13] J. Jeddeloh and B. Keeth, “Hybrid memory cube new DRAM architecture increases density and performance,” in *2012 Symposium on VLSI Technology (VLSIT)*, 2012, pp. 87–88.

[14] N. E. Jerger, T. Krishna, L.-S. Peh, and M. Martonosi, *On-Chip Networks: Second Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[15] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, “RecNMP: Accelerating Personalized Recommendation with near-Memory Processing,” in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA ’20. IEEE Press, 2020, p. 790–803. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00070

[16] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, “MAGIC—Memristor-Aided Logic,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[17] J. H. Lee, J. Sim, and H. Kim, “BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 241–252.

[18] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[19] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, “Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 655–668.

[20] A. Lyle, J. Harms, S. Patil, X. Yao, D. J. Lilja, and J.-P. Wang, “Direct communication between magnetic tunnel junctions for nonvolatile logic fan-out architecture,” *Applied Physics Letters*, vol. 97, no. 15, p. 152504, 2010. [Online]. Available: https://doi.org/10.1063/1.3499427

[21] A. Nag and R. Balasubramonian, *OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations*. New York, NY, USA: Association for Computing Machinery, 2021, p. 298–310. [Online]. Available: https://doi.org/10.1145/3466752.3480103

[22] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, “A Primer on Memory Consistency and Cache Coherence, Second Edition,” *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, 2020. [Online]. Available: https://doi.org/10.2200/S00962ED2V01Y201910CAC049

[23] J. Park, B. Kim, S. Yun, E. Lee, M. Rhu, and J. H. Ahn, “TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 268–281. [Online]. Available: https://doi.org/10.1145/3466752.3480080

[24] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[25] B. Perach, R. Ronen, B. Kimelfeld, and S. Kvatinsky, “PIMDB: Understanding Bulk-Bitwise Processing In-Memory Through Database Analytics,” 2022. [Online]. Available: https://arxiv.org/abs/2203.10486

[26] B. Rajaram, V. Nagarajan, S. Sarkar, and M. Elver, “Fast RMWs for TSO: Semantics and Implementation,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 61–72. [Online]. Available: https://doi.org/10.1145/2491956.2462196

[27] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, H. Cilasun, J. P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, “MOUSE: Inference In Non-volatile Memory for Energy Harvesting Applications,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 400–414.

[28] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. H. Ali, P.-E. Gaillardon, and S. Kvatinsky, “Memristive logic: A framework for evaluation and comparison,” in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[29] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology,” in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 273–287.

[30] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “X86-TSO: A Rigorous and Usable Programmer’s Model for X86 Multiprocessors,” *Commun. ACM*, vol. 53, no. 7, p. 89–97, jul 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[31] N. Talati, H. Ha, B. Perach, R. Ronen, and S. Kvatinsky, “CONCEPT: A Column-Oriented Memory Controller for Efficient Memory and PIM Operations in RRAM,” *IEEE Micro*, vol. 39, no. 1, pp. 33–43, 2019.

[32] “TPC benchmark H standard specification revision 3.0.0,” http://tpc.org/tpch/, Transaction Processing Performance Council, February 2021.

[33] M. S. Q. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, “RACER: Bit-Pipelined Processing Using Resistive Memory,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 100–116. [Online]. Available: https://doi.org/10.1145/3466752.3480071